



Janki Bhimani Biography:

Dr. Janki Bhimani is an Assistant Professor of Computer Science and Director of the Data Management Research Lab (DaMRL) at Florida International University (FIU). Prior to joining FIU in 2019, she worked with Samsung Semiconductors. She obtained her Ph.D. and M.S. degrees in Computer Engineering from Northeastern University, Boston. Dr. Bhimani's expertise spans the areas of system design, storage systems, memory management, computer architecture, electronic design automation (EDA), cloud computing, big data, modeling and simulation, resource management, capacity planning, machine learning (ML), and high-performance computing (HPC). She is the recipient of the NSF CAREER Award, FIU Top Scholar Award, KFSCIS Excellence in Research Award, Grace Hopper Celebration of Women in Computing (GHC) Faculty Scholarship, and Distinguished Reviewer and Best Paper Awards from flagship conferences. Her research has been supported by \$4.9 million in funding from prestigious federal and state agencies such as NSF and Cyber Florida, as well as industries like Samsung Semiconductors, out of which \$1.8 million are her PI funds. Her research has led to publications in high-impact journals such as IEEE TCC, IEEE TVT, ACM TOS, IEEE TC, ACM TOMACS, IEEE TBDATA, and IEEE TMSCS, and publications in highly selective conferences and workshops such as DAC, HPCA, DATE, CLOUD, HPDC, and HotStorage, earning an h-index of 16 and about 1000 citations, as well as 10 top-graded patent awards. Dr. Bhimani's graduated student alumni have gone on to successful research careers in academia and industry. As an educator, Dr. Bhimani is passionate to demonstrate excellence in her teaching and service roles. She has taught various core and effort intensive graduate and undergraduate level courses such as Storage Systems and Data Structures. She has taken on various leadership roles of general chair, program committee chair, track chair, publicity chair, session chair, etc. in organizing conferences such as CCGRID, ACM HotStorage, and HPDC. Within FIU, she has experience serving on the Faculty Hiring Committee, Awards Committee, CEC Faculty Council, Subject Area Coordinator, Graduate Committee, Diversity, Equity, and Inclusion (DEI) Committee, and Seminar Series Coordinator. Beyond her professional pursuits, in her free time, she is a creative visual artist who paints with oils on canvas.

Further latest details on her work can be found at <https://bhimanijanki.github.io/>.

Janki Bhimani

Assistant Professor, Knight Foundation School of Computing and Information Science
Florida International University

jbhimani@fiu.edu • [Linkedin](#) • [Google Scholar](#) • Mobile: +1(857)991-9868
(Please visit my [Website](#) for the most updated information.)

RESEARCH INTERESTS

System Design, Memory Management; Storage Systems; Computer Architecture; Optimization, Modeling, and Prediction; Resource Management; Cloud Computing; Machine Learning; Capacity Planning; High Performance Storage and Computing; Emerging Non-Volatile Memories;

HIGHLIGHTS

Research:

[Grants.] • Grants: Secured **\$4.9 million** in research funding from **NSF, Cyber Florida, and Samsung Semiconductors** with **\$1.8 million as PI**. Received **NSF CAREER** award in 2024. Other **five proposals** under review, including two as PI. Future plans include applying for large grants from DOE, NSF, and AFOSR.

[Awards, Publications, and Patents.] Contributed significantly with **10 high-impact journal articles, 35 peer-reviewed conference papers, 10 patents** as lead inventor, and over **900 citations**. Recognized with Awards including **FIU Top Scholar, KFSCIS Excellence in Applied Research, Distinguished Reviewer, and Best Paper Awards**.

[Student Advising/Mentoring.] **Graduated one Ph.D. student**, who is Assistant Professor, mentored eight Hispanics, one Asian American, and three women in the past five years.

[Collaborations.] Established collaborations with institutions like the University of Maryland, University of Chicago, Argonne National Lab, Syracuse University, and industry leaders Samsung Semiconductors and IBM Research.

Teaching:

Designed and taught **three core courses**, achieving an **average student feedback rating of 4.13/5**. Secured **Quality Matters (QM) certification** for courses. Led curriculum update efforts to design and integrate a Data Structures course taught in multiple programming languages. Implemented **module-based content distribution** with live feedback, and class projects on **Chameleon cloud platform**.

Service:

Contributed to FIU's growth, serving on the **Faculty Hiring Committee, Awards Committee, CEC Faculty Council, Subject area coordinator, Graduate Committee, DEI Committee**, and as **Seminar Series Coordinator**. Led roles such as **General Chair, Publicity Chair** for ACM HotStorage, **Program Committee Track Chair** for CCGRID, and **Poster Chair** for HPDC. Extensive service as a **TPC member** for conferences like USENIX FAST, IPDPS, CLOUD. Volunteering at the **Center for Women and Gender Studies**. Received **Grace Hopper Celebration of Women in Computing (GHC) Faculty Scholarship** for two years and Certificate of Completion from **ASEE DELTA Junior Faculty Institute**. Participated in **STRIDE workshop** for hiring, tenure, and promotion, **Diversity Advocate** workshop, and **Bystander Leadership** workshop.

EDUCATION

- Doctor of Philosophy (Ph.D.), Computer Engineering** Aug 2019
Northeastern University, Boston, MA, USA
Dissertation: Enhancing Efficiency and Endurance of Flash-Based Storage for Big Data Processing on Enterprise Cloud and Datacenter
- Master of Science (M.S.), Computer Engineering** Jan 2016
Northeastern University, Boston, MA, USA
M.S. research: FiM - Fine grained Model to Predict Heterogeneous Computing Platforms Performance
- Bachelor of Technology (B.Tech), Electrical & Electronics Engineering** Aug 2013
GITAM University, Vishakhapatnam, India
Major: Robotics and Programming of Embedded Systems, Minor: Circuit Design, Power Management

EXPERIENCE

- Assistant Professor, Computer Science** Aug 2019 - Current
Knight Foundation School of Computing and Information Science
Florida International University, Miami FL, USA
- Volunteering Affiliated Faculty** Aug 2019 - Current
Center for Women's and Gender Studies (CWGS)
Florida International University, Miami FL, USA
- Instructor, College of Engineering** Sep 2017 - Dec 2017
Northeastern University, Boston MA, USA
- Engineer - Performance Architect** May 2016 - Aug 2018
Samsung Semiconductors Inc. Research Lab, San Jose, CA, USA
- IC Design Intern** Jun 2012 - Jul 2012
Energy Options, Rajkot, India
- NASA STEM Engagement** May 2012 - Jun 2012
NASA's John F. Kennedy Space Center, FL, USA

PUBLICATIONS IN DISCIPLINE

(* Indicates an FIU student supervised by myself.)

Citation counts are taken from my Google Scholar Profile, which lists the following statistics:
Total citations: 937, h-index: 16, i10-index: 20.

[Total 45 - 20 FIU (13 with FIU students supervised by myself) and 25 Pre-FIU]

Selective Refereed Journal Publications

1. Danlin Jia, Li Wang, Natalia Valencia*, Janki Bhimani, Bo Sheng and Ningfang Mi. Learning-based Dynamic Memory Allocation Schemes for Apache Spark Data Processing. *IEEE Transactions on Cloud Computing (TCC)* 2023. Tier 1 Journal with impact factor 11.1.

2. Ajinkya S Bankar, Shi Sha, [Janki Bhimani](#), Vivek Chaturvedi, Gang Quan. Thermal Aware System-Wide Reliability Optimization for Automotive Distributed Computing Applications. *IEEE Transactions on Vehicular Technology (TVT)* 2022. Tier 1 Journal with impact factor 2.243.
3. [Janki Bhimani](#), Zhengyu Yang, Jingpei Yang, Adnan Maruf*, Ningfang Mi, Rajinikanth Pandurangan, Changho Choi, Vijay Balakrishnan. Automatic Stream Identification to Improve Flash Endurance in Data Centers. *ACM Transactions on Storage (TOS)* 2021. Tier 1 Journal with impact factor 1.59.
4. [Janki Bhimani](#), Adnan Maruf*, Ningfang Mi, Rajinikanth Pandurangan, and Vijay Balakrishnan. Auto-Tuning Parameters for Emerging Multi-Stream Flash-Based Storage Drives Through New I/O Pattern Generations. *IEEE Transactions on Computers (TC)* 2020. Tier 1 Journal with impact factor 3.131.
5. [Janki Bhimani](#), Ningfang Mi, Miriam Leeser, and Zhengyu Yang, New Performance Modeling Methods for Parallel Data Processing Applications, *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 2019. DOI 10.1145/3309684. Tier 1 Journal with impact factor 1.380.
6. Zhengyu Yang, Manu Awasthi, Mrinmoy Ghosh, [Janki Bhimani](#), and Ningfang Mi, I/O Workload Management for All-Flash Datacenter Storage Systems Based on Total Cost of Ownership, *IEEE Transactions on Big Data (TBDATA)*, Special Issue on the Integration of Extreme Scale Computing and Big Data Management and Analytics, 2018. DOI 10.1109/TBDATA.2018.2871114. Tier 1 Journal with impact factor 2.
7. [Janki Bhimani](#), Zhengyu Yang, Ningfang Mi, Jingpei Yang, Qiumin Xu, Manu Awasthi, Rajinikanth Pandurangan, and Vijay Balakrishnan, Docker Container Scheduler for I/O Intensive Applications running on NVMe SSDs, *IEEE Transactions on Multi-Scale Computing Systems (TMSCS)*, 2018. DOI: 10.1109/TMSCS.2018.2801281. Tier 1 Journal with impact factor 2.065.
8. Zhengyu Yang, [Janki Bhimani](#), Yi Yao, Cho-Hsien Lin, Jiayin Wang, Ningfang Mi, and Bo Sheng, AutoAdmin: Admission Control in YARN Clusters Based on Dynamic Resource Reservation, *Scalable Computing: Practice and Experience*, Special Issue on Advances in Emerging Wireless Communications and Networking (SCPE), 2018. Volume 19, Number 1, pp. 53–67.
9. Zhengyu Yang, Yufeng Wang, [Janki Bhimani](#), Chiu C. Tan, and Ningfang Mi, EAD: Elasticity Aware Deduplication Manager for Datacenters with Multi-tier Storage Systems, *Cluster Computing (CC)*, 2018. <https://doi.org/10.1007/s10586-018-2141-z>.
10. Zhengyu Yang, [Janki Bhimani](#), Jiayin Wang, David Evans, and Ningfang Mi, Automatic and Scalable Data Replication Manager in Distributed Computation and Storage Infrastructure of Cyber-Physical Systems, *Scalable Computing: Practice and Experience*, Special Issue on Communication, Computing, and Networking in Cyber-Physical Systems (SCPE), 2018. Volume 18, Number 4, pp. 291–311.

Highly Selective Peer Reviewed Conference Publications

Acceptance rates below 30%

11. [Manoj Saha*](#), [Danlin Jia](#), [Janki Bhimani](#) and [Ningfang Mi](#), MoKE: Modular Key-value Emulator for Realistic Studies on Emerging Storage Devices, 2023 *IEEE International*

- Conference on Cloud Computing (CLOUD'23), Hybrid Event, Chicago, IL, 2023.
12. Ziyang Jiao, Janki Bhimani, Bryan S. Kim, Wear Leveling in SSDs Considered Harmful, 2022 ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22), Virtual. (*Best Paper Award*)
 13. Adnan Maruf*, Sashri Brahmakshatriya*, Baolin Li, Devesh Tiwari, Gang Quan and [Janki Bhimani](#), Do Temperature and Humidity Exposures Hurt or Benefit Your SSDs?, 2022 Design, Automation and Test in Europe Conference. The European Event for Electronic System Design and Test (DATE'22), Virtual. Acceptance Rate: 25%. (*Best Paper Award Nomination*)
 14. Adnan Maruf*, Ashikee Ghosh*, [Janki Bhimani](#), Daniel Campello, Andy Rudoff, Raju Rangaswami, MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems, 2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA'22), Seoul, South Korea, 2022. Acceptance Rate: 30%.
 15. Adnan Maruf*, Zhengyu Yang, Bridget Davis, Daniel Kim, Jeffrey Wong, Matthew Durand, and [Janki Bhimani](#), Understanding Flash-Based Storage I/O Behavior of Games, 2021 IEEE International Conference on Cloud Computing (CLOUD'21), Online Virtual Congress, 2021. Acceptance Rate: 23.8%.
 16. [Janki Bhimani](#), Jingpei Yang, Ningfang Mi, Changho Choi, and Manoj Pravakar Saha*, Fine-grained Control of Concurrency within KV-SSDs, 2021 14th ACM International Systems and Storage Conference (SYSTOR'21), Virtual. Acceptance Rate: 29.9%.
 17. Manoj Pravakar Saha*, Bryan Kim, and [Janki Bhimani](#), KV-SSD: What is it Good For?, 2021 Design Automation Conference (DAC'21), San Francisco, CA, 2021. Acceptance Rate: 23%.
 18. Danlin Jia, Manoj Pravakar Saha*, [Janki Bhimani](#), and Ningfang Mi, Performance and Consistency Analysis for Distributed Deep Learning Applications, 2020 International Performance Computing and Communications Conference (IPCCC'20), Virtual using Zoom, 2020. Acceptance Rate: 29.3%.
 19. [Janki Bhimani](#), Rajinikanth Pandurangan, Ningfang Mi, and Vijay Balakrishnan, Emulate Processing of Assorted Database Server Applications on Flash-Based Storage in Datacenter Infrastructures, 2019 International Performance Computing and Communications Conference (IPCCC'19), London, UK, 2019. Acceptance Rate: 29.2%.
 20. Danlin Jia, [Janki Bhimani](#), Son Nam Nguyen, Bo Sheng, and Ningfang Mi, ATuMm: Auto-tuning Memory Manager in Apache Spark, 2019 International Performance Computing and Communications Conference (IPCCC'19), London, UK, 2019. Acceptance Rate: 29.2%.
 21. [Janki Bhimani](#), Tirthak Patel, Ningfang Mi, and Devesh Tiwari, "What does Vibration do to Your SSD?", 2019 Design Automation Conference (DAC'19), Las Vegas, NV, 2019. Acceptance Rate: 24.3%.
 22. [Janki Bhimani](#), Ningfang Mi, Zhengyu Yang, Jingpei Yang, Rajinikanth Pandurangan, Changho Choi and Vijay Balakrishnan, "FIOS: Feature Based I/O Stream Identification for Improving Endurance of Multi-Stream SSDs", 2018 IEEE International Conference on Cloud Computing (CLOUD'18), San Francisco, CA, 2018. Acceptance Rate: 15%. (*Best Paper Award*)

23. [Janki Bhimani](#), Ningfang Mi, and Bo Sheng, “BloomStream: Data Temperature Identification for Flash Based Memory Storage Using Bloom Filters”, 2018 IEEE International Conference on Cloud Computing (CLOUD’18), San Francisco, CA, 2018. Acceptance Rate: 15%.
24. Zhengyu Yang, Morteza Hoseinzadeh, Ping Wong, John Artoux, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, [Janki Bhimani](#), Ningfang Mi, and Steven Swanson, “H-NVMe: A Hybrid Framework of NVMe-based Storage System in Cloud Computing Environment”, IEEE International Performance Computing and Communications Conference (IPCCC’17), San Diego, CA, 2017. (*Best Paper Award*)
25. Zhengyu Yang, Morteza Hoseinzadeh, Allen Andrews, Clay Mayers, David Thomas Evans, Rory Thomas Bolt, [Janki Bhimani](#), Ningfang Mi, and Steven Swanson, “AutoTiering: Automatic Data Placement Manager in Multi-Tier All-Flash Datacenter”, IEEE International Performance Computing and Communications Conference (IPCCC’17), San Diego, CA, 2017.
26. [Janki Bhimani](#), Ningfang Mi, Miriam Leeser, and Zhengyu Yang, “FiM: Performance Prediction Model for Parallel Computation in Iterative Data Processing Applications”, IEEE International Conference on Cloud Computing (CLOUD’17), Honolulu, HI, 2017. Acceptance Rate: 18%.
27. Han Gao, Zhengyu Yang, [Janki Bhimani](#), Teng Wang, Jiayin Wang, Ningfang Mi, and Bo Sheng, “AutoPath: Harnessing Parallel Execution Paths for Efficient Resource Allocation in Multi-Stage Big Data Frameworks”, International Conference on Computer Communications and Networks (ICCCN’17), Vancouver, Canada, 2017. Acceptance Rate: 25%.
28. Qiumin Xu, Manu Awasthi, Krishna T. Malladi, [Janki Bhimani](#), Jingpei Yang, and Murali Annavaram. “Performance analysis of containerized applications on local and remote storage” International Conference on Massive Storage Systems and Technology (MSST’17), Santa Clara, CA, 2017.
29. [Janki Bhimani](#), Jingpei Yang, Zhengyu Yang, Ningfang Mi, Qiumin Xu, Manu Awasthi, Rajinikanth Pandurangan, and Vijay Balakrishnan, “Understanding Performance of I/O Intensive Containerized Applications for NVMe SSDs”, IEEE International Performance Computing and Communications Conference (IPCCC’16), Las Vegas, NV, 2016. Acceptance Rate: 25.50%.
30. Zhengyu Yang, Jianzhe Tai, [Janki Bhimani](#), Jiayin Wang, Ningfang Mi, and Bo Sheng, “GREM: Dynamic SSD Resource Allocation in Virtualized Storage Systems with Heterogeneous VMs”, IEEE International Performance Computing and Communications Conference (IPCCC’16), Las Vegas, NV, 2016. Acceptance Rate: 25.50%.

Other Peer Reviewed Conference and Workshop Publications

Acceptance rates provided when known

31. [Manoj P. Saha*](#), [Omkar Desai](#), [Bryan S. Kim](#), [Janki Bhimani](#). “Leveraging Keys In Key-Value SSD for Production Workloads” The International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC’23), Orlando, FL, 2023. (Short Paper)
32. [Adnan Maruf*](#), [Daniel Carlson*](#), [Ashikee Ghosh*](#), [Manoj Saha*](#), [Janki Bhimani](#), [Raju Rangaswami](#). “Allocation Policies Matter for Hybrid Memory Systems” The International

ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'23), Orlando, FL, 2023. (Short Paper)

33. Manoj P. Saha*, Bryan S. Kim, Haryadi S. Gunawi, [Janki Bhimani](#). “RHIK - Re-configurable Hash-based Indexing for KVSSD” The International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'23), Orlando, FL, 2023. (Short Paper)
34. Mahsa Bayati, [Janki Bhimani](#), Ronald Lee, Ningfang Mi. “Exploring Benefits of NVMe SSDs for BigData Processing in Enterprise Data Centers” International Conference on Big Data Computing and Communication (BIGCOM'19), Qingdao, China, 2019.
35. [Janki Bhimani](#), Jingpei Yang, Zhengyu Yang, Ningfang Mi, NHV Krishna Giri, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. “Enhancing SSDs with multi-stream: What? why? how?” IEEE International Performance Computing and Communications Conference (IPCCC'17), San Diego, CA, 2017. (Short Paper)
36. [Janki Bhimani](#), Zhengyu Yang, Miriam Leeser, and Ningfang Mi, “Accelerating Big Data Applications Using Lightweight Virtualization Framework on Enterprise Cloud”, IEEE High Performance Extreme Computing Conference (HPEC'17), Waltham, MA, 2017.
37. Qiumin Xu, Manu Awasthi, Krishna T. Malladi, [Janki Bhimani](#), Jingpei Yang, Murali Annavaram, “Docker Characterization on High Performance SSDs”, IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS'17), Santa Rosa, CA, 2017. (Short Paper)
38. Liu Chao, [Janki Bhimani](#), and Miriam Leeser, “Using High Level GPU Tasks to Explore Memory and Communications Options on Heterogeneous Platforms” ACM Workshop on Software Engineering Methods for Parallel and High Performance Applications (SEM4HPC), Washington, D.C., 2017.
39. Liu Chao, [Janki Bhimani](#), and Miriam Leeser, “Exploring Memory Options for Data Transfer on Heterogeneous Platforms”, The International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'17), Washington, D.C., 2017. (Short Paper)
40. [Janki Bhimani](#), Miriam Leeser, and Ningfang Mi, “Performance Prediction Techniques for Scalable Large Data Processing in Distributed MPI Systems”, IEEE International Performance Computing and Communications Conference (IPCCC'16), Las Vegas, NV, 2016. Acceptance Rate: 12%. (Short Paper)
41. [Janki Bhimani](#), Miriam Leeser, and Ningfang Mi, “Design Space Exploration of GPU Accelerated Cluster Systems for Optimal Data Transfer Using PCIe Bus”, IEEE High Performance Extreme Computing Conference (HPEC'16), Waltham, MA, 2016.
42. [Janki Bhimani](#), Miriam Leeser, and Ningfang Mi, “Accelerating K-Means Clustering with Parallel Implementations and GPU Computing”, IEEE High Performance Extreme Computing Conference (HPEC'15), Waltham, MA, 2015.
43. [Janki Bhimani](#), Miriam Leeser and Ningfang Mi, “Predicting the Performance of Machine Learning Algorithms running on Heterogeneous Computing Platforms” Women in Machine Learning Workshop (WiML'14), Montréal, Canada, 2014. (Short Paper)
44. Baiyu Chen, Zhengyu Yang, Siyu Huang, Xianzhi Du, Zhiwei Cui, [Janki Bhimani](#), Xin Xie, and Ningfang Mi, “Cyber-Physical System Enabled Nearby Traffic Flow Modelling

for Autonomous Vehicles”, IEEE International Performance Computing and Communications Conference, Special Session on Cyber Physical Systems: Security, Computing, and Performance (IPCCC CPS’17), San Diego, CA, 2017.

45. Xianfei Xia, Hongru Xiao, Zhengyu Yang, Xin Xie, and Janki Bhimani, Pelletization Characteristics of the Hydrothermal Pretreated Rice Straw with Added Binders. *Arabian Journal for Science and Engineering* 43, no. 9 (2018): 4811-4820.

Books N/A

Chapters in Books N/A

Government Reports or Monographs N/A

Book Reviews N/A

PRESENTED PAPERS, AND LECTURES

1. Guest Speaker: Picking Research as Career, Women in CS (WiCS) Student Chapter, Miami, FL, April 17, 2023.
2. Guest Speaker: Research Towards Data Storage and Management, Presentation Request for Flit-Path Scholars, Miami, FL, February 11, 2022.
3. Invited Speaker: Emerging Technologies Moving Forward, Entrepreneurs’ Organization (EO), Miami, FL, February 10, 2022.
4. Guest Lecture: Towards Designing Intelligent Storage Devices, IBM Research, Almaden, San Jose, CA, May 12, 2021.
5. Guest Lecture: Challenges of the Evolving Memory and Storage Technologies, Memory Solutions Lab, Samsung, San Jose, CA, October 23, 2020.
6. Guest Speaker: Ph.D. in Computer Science – from the lens of a Girl who likes pink, FIU Women in Cybersecurity (WiCys) Student Chapter, Miami, FL, October 22, 2020.
7. Guest Lecture: New Techniques for Data Management in Evolving Storage Technologies, Florida International University, Miami, FL, November 22, 2019.
8. Guest Lecture: New Storage Technologies for Big Data Processing on Cloud and Datacenter Infrastructures, Colorado State University, Fort Collins, CO, March 27, 2019.
9. Paper Presentation Talk: FIOS: Feature Based I/O Stream Identification for Improving Endurance of Multi-Stream SSDs, 2018 IEEE International Conference on Cloud Computing (CLOUD’18), San Francisco, CA, 2018.
10. Paper Presentation Talk: BloomStream: Data Temperature Identification for Flash Based Memory Storage Using Bloom Filters, 2018 IEEE International Conference on Cloud Computing (CLOUD’18), San Francisco, CA, 2018.
11. Paper Presentation Talk: FiM: Performance Prediction Model for Parallel Computation in Iterative Data Processing Applications, IEEE International Conference on Cloud Computing (CLOUD’17), Honolulu, HI, 2017.
12. Paper Presentation Talk: Understanding Performance of I/O Intensive Containerized Applications for NVMe SSDs, IEEE International Performance Computing and Communications Conference (IPCCC’16), Las Vegas, NV, 2016.

13. Paper Presentation Talk: Accelerating Big Data Applications Using Lightweight Virtualization Framework on Enterprise Cloud, IEEE High Performance Extreme Computing Conference (HPEC'17), Waltham, MA, 2017.
14. Paper Presentation Talk: Design Space Exploration of GPU Accelerated Cluster Systems for Optimal Data Transfer Using PCIe Bus, IEEE High Performance Extreme Computing Conference (HPEC'16), Waltham, MA, 2016.
15. Paper Presentation Talk: Accelerating K-Means Clustering with Parallel Implementations and GPU Computing, IEEE High Performance Extreme Computing Conference (HPEC'15), Waltham, MA, 2015.

CREATIVE WORK

1. Conducting a SWOT Analysis as a strategic planning technique has been a key initiative to enhance the overall learning experience of the class.
2. Taking the lead in designing and developing the website for the VOCES project, which can be accessed at <https://voces.fiu.edu/>, has been a significant contribution.
3. Creating an efficient workflow that covers the entire lifecycle of multimedia posts, including stages such as concept brainstorming, assignment of responsibilities, post design using tools like Canva, feedback collection and review, post enhancements, and final staging, has streamlined the process.
4. Determining the optimal posting frequency and content distribution across various social media platforms, such as LinkedIn, Instagram, and others, with the aim of organically boosting engagement and cultivating student interest in becoming VOCEROs, has been a strategic focus.
5. Creating a pioneering teaching method by incorporating surveys and polls to assess student engagement and pinpoint challenging topics has been a transformative approach. Initially breaking down module topics into more digestible sub-topics, prompting students to assess their difficulty through pre-class polls, and then revisiting the sub-topics that most students found challenging in the following class using real-time feedback, employing personalized teaching techniques to enhance understanding, and facilitating an open forum for students to anonymously ask questions has significantly improved the learning experience.

WORKS IN PROGRESS

Publications

1. Ali Bin Omer Qureshi*, Lie Pan, Dheeraj Gandhi, Yifan Guo and Janki Bhimani, Optimizing Data Transfers in Edge Computing
2. Ali Bin Omer Qureshi*, Lie Pan, Dheeraj Gandhi, Yifan Guo and Janki Bhimani, Data Spilling Reduction in PrestoDB
3. Manoj Pravakar Saha*, Bryan Kim, and Janki Bhimani, Optimizing In-Storage Indexing
4. Omkar Desai, Daniel Carlson*, Janki Bhimani, Bryan S. Kim, Conch: Caching System for Concurrent DNN Training
5. Pratik Poudel*, Jason Liu, Janki Bhimani, Distributed Workload Characterization Optimizer

6. Manoj Pravakar Saha*, Bryan Kim, and Janki Bhimani, DeePaM: Distributed Deep-Learning Page Cache Management for Disaggregated Memory System
7. Manoj Pravakar Saha*, Raju Rangaswami, Yanzhao Wu, and Janki Bhimani, MiPiCheck: Mixed Pipeline Checkpointing
8. Manoj Pravakar Saha*, Ashikee Ghosh*, Raju Rangaswami, Yanzhao Wu, and Janki Bhimani, LATTICE: Looking Beyond File I/O-based DNN Checkpointing
9. Omkar Desai, Adnan Maruf*, Janki Bhimani, Bryan S. Kim, Modeling the space-time trade-off for the DSI pipeline in ML training.
10. Adnan Maruf*, Ashikee Ghosh*, Janki Bhimani, and Raju Rangaswami, RUMINANT-Adaptive Tiering for Hybrid Memory Systems.
11. Daniel Carlson*, Adnan Maruf*, Ashikee Ghosh*, Janki Bhimani, and Raju Rangaswami, Understanding Hybrid Memory Allocations.
12. Adnan Maruf*, Daniel Carlson*, Janki Bhimani, Persistent Memory To Better Manage File-backed Pages of Gaming Workloads.
13. Adnan Maruf*, Dwaraka Prasath Mohen Babu*, Christopher Meadows*, Ningfang Mi, Bo Sheng, and Janki Bhimani, Data Structure For Low-Overhead Stream Identification.
14. Manoj Pravakar Saha*, Bryan Kim, and Janki Bhimani, Flash Isolation Without Instrumentation Overhead.
15. Manoj Pravakar Saha*, Omkar Desai, Bryan Kim, and Janki Bhimani, Multi-Tenant Key Value Device Indexing.
16. Manoj Pravakar Saha*, Aris Duani Rojas*, and Janki Bhimani, Parallel Data Access Within Key Value Storage Devices.

Proposals Submitted For Review and Under Preparation

[2 PI + 3 co-PI]

1. NSF AI Institute on Neuro-Symbolic AI, under preparation.
2. CAHSI-Google Institutional Research Program (IRP): Optimizing Data Workflows in ML, PI, Oct 2024 - Sep 2025, \$100,000.
3. CSR: Medium: DISCO: Disciplined Data Science Framework for Storage I/O Management, PI, Oct, 2024 - Jun 2028, \$1,200,000, My share: \$475,000 (37.5%) PI team: Haryadi S. Gunawi, University of Chicago and Sandeep R. Madireddy, Argonne National Laboratory.
4. Samsung GRO: New Techniques for Managing Accelerator Compute and CXL Memory to Improve Performance and Scalability of AI Inference, PI, Oct 2024 - Sep 2026, \$299,852, PI team: Ningfang Mi, Northeastern University.

FUNDED RESEARCH GRANTS

[9 PI + 3 co-PI/SI = \$4,892,218; Total PI funds = \$1,792,232; My share of PI funds to FIU = \$1,497,303; Co-PI/SI funds to FIU = \$95,750.79]

1. **2024-2029 NSF CAREER (Only PI)**
[“CAREER-2338457 - Towards Efficient In-storage Indexing”](#)

- Total Value: \$615,528 (Direct+Indirect) My share: \$615,528 (100%)
Start date: Jul 1, 2024 Expiration date: Jun 30, 2029
2. **2024-2026 NSF REU Supplement (Only PI)**
“CSR-2406069 - REU: Learning and Management in Tiered Memory Systems”
Total Value: \$16,000 (Direct+Indirect) My share: \$16,000 (100%)
Start date: Jan 1, 2024 Expiration date: Dec 31, 2024
Project ID: 800020019
 3. **2023-2026 NSF CISE Core (Only PI)**
“CSR-2323100 NSF Core: CSR: Small: Learning and Management in Tiered Memory Systems”
Total Value: \$514,704 (Direct+Indirect) My share: \$514,704 (100%)
Start date: Oct 1, 2023 Expiration date: Sep 30, 2026
Project ID: 800018181
 4. **2023-2024 Samsung Global Research Outreach (GRO) Award (PI)**
“Leveraging Disaggregated Servers for Large Scale AI Training Acceleration”
Co-PI: Bryan Kim, Syracuse University
Total Value: \$50,000 (Direct+Indirect) My share: \$25,000 (50%)
Start date: Sep 1, 2023 Expiration date: Feb 29, 2024
 5. **2022-2027 NSF HSI (SI)**
“HRD-2225201 - HSI Institutional Transformation Project Voces (Voices for Organizing Change in Educational Systems)”
PI team: Yesim Darici, Stephen Secules, Rocio Benabentos, Laird Kramer, Jaroslava Miksovska, Monica Cardella, FIU
Total Value: \$2,999,986 (Direct+Indirect) My share: \$69,340 (2.3%)
Start date: Sep 15, 2022 Expiration date: Aug 31, 2027
 6. **2022-2023 Samsung Memory Solutions Lab (MSL) Research Award (PI)**
“Leveraging Disaggregated Servers for Large Scale AI Training Acceleration”
Co-PI: Bryan Kim, Syracuse University
Total Value: \$50,000 (Direct+Indirect) My share: \$25,000 (50%)
Start date: Mar 1, 2023 Expiration date: Aug 31, 2023
 7. **2021 Samsung Semiconductor Inc. Equipment Grant (Only PI)**
“Parallel Data Access with Key-Value SSDs”
Total Value: \$10,000 (Direct+Indirect) My share: \$10,000 (100%)
Start date: Oct 1, 2021 Expiration date: Jun 30, 2022
 8. **2021-2022 NSF REU Supplement (Only PI)**
“CNS-2122987 - REU: New Techniques for I/O Behavior Modeling and Persistent Storage Device Configuration”
Total Value: \$16,000 (Direct+Indirect) My share: \$16,000 (100%)
Start date: May 1, 2021 Expiration date: Apr 30, 2022
Project ID: 800014726

9. **2020-2023 NSF CISE Core (Lead PI)**
 “CNS-2008324 - Small: New Techniques for I/O Behavior Modeling and Persistent Storage Device Configuration”
 Co-PI: Ningfang Mi, Northeastern University
 Total Value: \$500,000 (Direct+Indirect) My share: \$255,071 (51%)
 Start date: May 1, 2020 Expiration date: Apr 30, 2023
 Project ID: 800012359
10. **2020-2022 Cyber Florida (Co-PI)**
 “RumorHunt: A Next-Generation Online Scalable Streaming System”
 PI team: Liting Hu, FIU and Zhishan Guo, University of Central Florida
 Total Value: \$75,000 (Direct+Indirect) My share: \$21,410.79 (28%)
 Start date: Aug 1, 2020 Expiration date: May 30, 2022
 Project ID: 800012574
11. **2019-2020 FIU Faculty Grantsmanship Development Program (Co-PI)**
 “Design, Development and Testing of Distributed Computing Framework for globally coordinated data submission and accessibility of Mass Spectrometry Data”
 PI team: Fahad Saeed, Alex Afanasyev, Hadi Amini, FIU
 Total Value: \$25,000 (Direct+Indirect) My share: \$5,000 (20%)
 Start date: Nov 1, 2019 Expiration date: May 30, 2020
12. **2019 Samsung Semiconductor Inc. Equipment Grant (Only PI)**
 “Exploring Vulnerabilities of Key-Value SSDs”
 Total Value: \$20,000 (Direct+Indirect) My share: \$20,000 (100%)
 Start date: Oct 1, 2019 Expiration date: Sep 30, 2021

PROPOSALS SUBMITTED BUT NOT FUNDED

1. Pre-Application for ASCR-RENEW (DE-FOA-0002942): Establishing a Sustainable Pathway for Hispanic Students in AI and High-Performance and Scientific Computing Careers at Florida International University (FIU) and Sandia National Laboratories (SNL), Co-PI, PI team: Jason Liu, Raju Rangaswami, Cuong Nguyen, Wenqian Dong, Yanzhao Wu, FIU, Ron Oldfield, Jay Lofstead, Andrew Younge, Patricia Gharagozloo, Sandia National Laboratories (SNL).
2. NSF Core: Collaborative Research: CSR: Medium: TensorHeap: Cross-Stack Memory Management for Machine Learning, Co-PI, Aug 2023 - Jul 2027, \$546,523, PI team: Raju Rangaswami, Jason Liu, FIU, Ming Zhao, Jia Zou, Arizona State University, and Wujie Wen, Lehigh University.
3. NSF CISE Core Medium: LISSA: Learning in the Storage Stack, PI, Jul 2022 - Jun 2026, \$1,200,000, PI team: Haryadi S. Gunawi, University of Chicago and Sandeep R. Madireddy, Argonne National Laboratory.
4. NSF CAREER: Ameliorate In-storage Indexing, Single PI, Jan 2023 - Dec 2028, \$599,789
5. DOE CAREER: Techniques to Leverage Emerging Persistent Memories to Accelerate Parallel Data Accesses, Aug 2022 - Jul 2027, \$750,000
6. NSF CISE: CNS Core: Small: Towards an Adaptive, Multi-Indexed, and Distributed Key-Value Based Flash Storage, PI, Aug 2022 - Jul 2025, \$600,000, PI team: Bryan Kim,

Syracuse University.

7. NSF PPOSS large: PPOSS: LARGE: TensorHeap: Cross-Stack Memory Management for Machine Learning, Co-PI, Aug 2022 - Jul 2027, \$2,011,614, PI team: Raju Rangaswami, Jason Liu, FIU, Ming Zhao, Jia Zou, Arizona State University, and Wujie Wen, Lehigh University.
8. NSF STC: Science and Technology Center for Decentralized Autonomous Organizations, Key Personnel, Jan 2023 - Dec 2028, \$30,000,000, PI team: Yesim Darici, Kemal Akkaya, Hebin Li, Raju Rangaswami, Selcuk Uluagac, Jason Liu, Sukumar Ganapati, Aslihan Akkaya, Laird Kramer, FIU, Ismail Guvenc, North Carolina State University, Ming Zhao, Arizona State University, Shengwang Du, University of Texas at Dallas, Yufei Ding, University of California at Santa Barbara, and Michael Titze, Sandia National Labs.
9. NSF Collaborative Research: PPOSS: Planning: Cross-Stack Memory Management for Machine Learning, Co-PI, Aug 2021 - Jul 2023, \$110,000, PI team: Raju Rangaswami, Jason Liu, FIU, Ming Zhao, Jia Zou, Arizona State University, and Wujie Wen, Lehigh University.
10. NSF INCLUDES Alliance: Tomorrow's Women in STEM Today (TWIST), Co-PI, Aug 2021 - Jul 2026, \$9,257,973, PI team: Yesim Darici, Jessy Abouarab, Mireya Mayor, FIU.
11. NSF CAREER: Leveraging Persistent Key-Value SSDs, Single PI, Jan 2021 - Dec 2026, \$542,600
12. NSF Distributed Infrastructure for Making Mass Spectrometry Data Findable and Accessible, Co-PI, Aug 2021 - Sep 2024, \$803,269, PI team: Fahad Saeed and Hadi Amini, FIU.
13. NSF NSF INCLUDES: Planning Grant FIU TWIST, Co-PI, Aug 2020 - Jul 2022, \$99,998, PI team: Yesim Darici, Jessy Abouarab, Mireya Mayor, FIU.
14. NSF CIBR: Distributed Infrastructure for Big Mass Spectrometry Data, Co-PI, Oct 2020 - Sep 2023, \$1,048,558, PI team: Fahad Saeed and Hadi Amini, FIU.
15. NSF Collaborative Research: PPOSS: Planning: Leveraging Persistent Memory for Trustworthy ML, Co-PI, Aug 2020 - Jul 2022, \$146,262, PI team: Raju Rangaswami, FIU, Ming Zhao, Jia Zou, Arizona State University, and Wujie Wen, Lehigh University.
16. Facebook Faculty Research Award for Distributed Systems Research: Distributed Systems for Deep-Learning with Heterogeneous Persistent Storage, Single PI, Aug 2020 - Jul 2022, \$100,000
17. Facebook Faculty Research Award for Systems for Machine Learning: System Memory and Storage Management for Deep Learning, Single PI, Aug 2020 - Jul 2022, \$100,000
18. Microsoft Faculty Award: Efficient Resource Management for Distributed Deep-Learning with Flash-Based Persistent Storage, Single PI, Aug 2020 - Jul 2022, \$200,000
19. FIU Faculty Grantsmanship Development Program: Visualizing Women's Health in Miami-Dade County, Co-PI, Jan 2019 - Dec 2020, \$25,000, PI team: Yesim Darici, Jessy Abouarab.
20. FIU Faculty Grantsmanship Development Program: Evaluating Effective Pipeline Strategies for Women in STEM Success Using GIS: What works and what doesn't work in FL Schools, Co-PI, Jan 2020 - Dec 2020, \$25,000, PI team: Yesim Darici, Jessy Abouarab.

21. Bill & Melinda Gates Foundation: Re-Examining the Patterns of and Motivations for Traditional Contraception Method Use in India and Sub-Saharan Africa: A Mixed-Method Approach, Key Personnel, Aug 2020 - Jul 2025, \$1,977,500, PI team: Yesim Darici, Jessy Abouarab, Haiying Long, Sarah Stumbar, Jessica Meadows, Stephany Alvarez-Ventura, FIU and Rahman Tauhidur, Arizona State University.
22. NSF CRII: CSR: System Support for Evolving Flash-Based Persistent Storage to Accelerate Parallel Applications, Single PI, May 2020 - April 2022, \$174,915.99

PATENT DISCLOSURES, APPLICATIONS, AND AWARDS

(Content in blue color are items since arriving at FIU.)

- Daniel Carlson*, Adnan Maruf*, Raju Rangaswami, and [Janki Bhimani](#), inventors; "Techniques to Dynamically Allocate Pages within CXL Memory Systems", Application.
2. [Manoj Pravakar Saha*](#), Yanzhao Wu, Raju Rangaswami, and [Janki Bhimani](#), inventors; "Methods to Efficiently Checkpoint Deep-Learning Model on Persistent Memories", Application.
 3. [Manoj Pravakar Saha*](#), [Janki Bhimani](#), inventors; "Flexible and Efficient Data Management Techniques Within Key Value Storage", US 17/340,573.
 4. [Adnan Maruf*](#), [Ashikee Ghosh*](#), Raju Rangaswami, and [Janki Bhimani](#), inventors; "ML based Tiered Memory", US 17/344,449.
 5. [Janki Bhimani](#), Jingpei Yang, Changho Choi, inventors; Samsung Electronics Co Ltd, assignee. "Parallel key value based multi-thread machine learning exploiting KV-SSDs" US 16/528,492.
 6. [Janki Bhimani](#), Rajinikanth Pandurangan, Changho Choi, Vijay Balakrishnan, inventors; Samsung Electronics Co Ltd, assignee. "System and method for identifying hot data and stream in a solid-state drive" US 15/895797.
 7. [Janki Bhimani](#), Rajinikanth Pandurangan, Vijay Balakrishnan, Changho Choi, inventors; Samsung Electronics Co Ltd, assignee. "Methods and systems for testing storage devices via a representative I/O generator" United States patent application US 15/853419.
 8. [Janki Bhimani](#), Anand Subramanian, Vijay Balakrishnan, and Jingpei Yang, inventors; Samsung Electronics Co Ltd, assignee. "Container workload scheduler and methods of scheduling container workloads" United States patent application US15/820856.
 9. [Janki Bhimani](#), Jingpei Yang, Changho Choi, Jianjian Huo, inventors; Samsung Electronics Co Ltd, assignee. "Smart I/O stream detection based on multiple attributes" United States patent application US 15/344,422.
 10. [Janki Bhimani](#), Hingkwon Huen, Jingpei Yang, Manu Awasthi, Vijay Balakrishnan, Jason Martineau, inventors; Samsung Electronics Co Ltd, assignee. "Intelligent controller for containerized applications" United States patent application US 15/379,327.

PROFESSIONAL ACHIEVEMENTS, HONORS, AWARDS, AND FELLOWSHIPS

(Content in blue color are items since arriving at FIU.)

1. 2024 - Received NSF CAREER Award.
2. 2023 - Received FIU Top Scholar Award in the category of the Research, Creative Activities, and Award-Winning Publications.
3. 2023 Quality Matters Certification for Online Course - CIS5346 Storage System
4. 2022 - Received Outstanding Applied Research Award by Knight Foundation School of Computing and Information Science (KFSCIS), FIU.
5. 2022 The Best Paper Award at 14th ACM Workshop on Hot Topics in Storage and Filesystem (HotStorage'22).
6. 2022 The Best Paper Award Nomination at Design, Automation and Test in Europe Conference. The European Event for Electronic System Design and Test (DATE'22)
7. 2022 Quality Matters Certification for Online Course - COP3530 Data Structures
8. 2021 Awarded Certificate of Completion from ASEE DELTA Junior Faculty Institute
9. 2021 Grace Hopper Celebration of Women in Computing (GHC) Faculty Scholarship
10. 2021 Recognized as Distinguished Reviewer Award, 13th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '21)
11. 2020 Received Certification for Hybrid Course - COP3530 Data Structures
12. 2020 Grace Hopper Celebration of Women in Computing (GHC) Faculty Scholarship
13. 2019 Outstanding Graduate Research Award, Northeastern University
14. 2018 The Best Paper Award at 11th IEEE International Conference on Cloud Computing (IEEE CLOUD)
15. 2017 The Best Paper Award at 36th IEEE International Performance Computing and Communications Conference (IPCCC)
16. 2014 Double Husky Scholarship, Northeastern University
17. 2012 The Best Budget Robot Award at 3rd Lunabotics International Mining Competition, NASA, Kennedy Space Center, FL
18. 2012 The Best Working Model Award in Junk Yard Wars at Conscientia, Indian Institute of Space Science Technology (IIST)
19. 2012 The Best Paper Award at Aagama National Level Technical Paper Contest
20. 2011 The Best Working Model in Junk Yard Wars during Technozion at National Institute Of Technology (NIT)
21. 2011 The Outstanding Debate Performance Award by Institute of Engineers India (IEI)
22. 2010 The Impromptu Speaker Award by International Society for Technology in Education (ISTE)
23. 2010 - 2013 University Merit Scholarship, GITAM University

ACADEMIC SUPERVISION

Doctoral Students (Thesis Advisees)

[10 Ph.D. Thesis Advisees (1 graduated + 1 proposal defended + 3 candidate + 3 discontinued + 2 hired applicant starting next semester), and 10 Ph.D. Committee Member (4 graduated + 3 proposal defended + 3 qualifying passed)]

Graduated Ph.D. Students

1. Adnan Maruf, Ph.D. student
Dissertation topic: Improving the performance and reliability of systems with emerging memory and storage devices
Graduated in Apr. 2023
Tenure Track Assistant Professor, Missouri State University

Current Ph.D. Students (Thesis Advisees)

2. Manoj Pravakar Saha, Ph.D. student
Dissertation topic: Enhancing the in-storage indexing and ML checkpointing
Dissertation plan: Spring 2025
3. Ali Bin Omer Qureshi, Ph.D. student
Dissertation topic: Optimizing Data Spilling in Distributed Query Engine and Memory Management
Dissertation plan: Fall 2029
4. Mayur Akewar, Ph.D. student
Dissertation topic: Towards Designing New Techniques for AI Based Data Indexing and Neuro-Symbolic AI
Dissertation plan: Spring 2030
5. Gabriel Rovira, Ph.D. student
Dissertation topic: Improving Data Distribution Pipeline in Heterogeneous Memory System with CXL Devices
Dissertation plan: Fall 2030
(2 new Ph.D. students hired starting next semester)

Discontinued Ph.D. Students

6. Daniel Carlson, Ph.D. student
Dissertation topic: Improving the Performance of Dis-aggregated Memory Systems
Duration advised by me: Fall 2020 - Spring 2024
7. Ashikee Ghosh, Ph.D. student
Dissertation topic: Designing Libraries for Efficient ML Checkpointing

Duration co-advised by me: Spring 2020 - Fall 2023
Software Development Engineer, Amazon

8. Maimuna Begum Kali, Ph.D. student
Dissertation topic: Optimizing Parallel Operations within BigData Processing Platforms
Duration advised by me: Fall 2019 - Fall 2021
Ph.D. student, School of Universal Computing, Construction, and Engineering Education (SUCCEED)

Ph.D. Committee Member

9. Rafael Trujillo
10. Pratik Poudel
11. Pedro Espina
12. Sumesh Kumar
13. Ziyang Jiao (Syracuse University)
14. Omkar Desai (Syracuse University)
15. Liana Valdes Rodriguez (Graduated)
16. Oswaldo Artiles (Graduated)
17. Muhammad Haseeb (Graduated)
18. Danlin Jia (Northeastern University) (Graduated)

M.S. Students (Thesis/Project Advisees)

[5 M.S. Students (3 graduated + 2 current)]

Graduated M.S. Students

19. Dwaraka Prasath Mohen Babu, ESpace Networks
20. Ashikee Ghosh, Amazon
21. Ali Bin Omer Qureshi

Current M.S. Students

22. Shashidhar Reddy Chavula
23. Muttahar Khalid

Research Experience for Undergrad (REU) Students

Graduated Undergrad Students

24. Gabriel Zavala, Dell
25. Daniel Carlson
26. Roberto Martinez, Co-Founder & CPO of GammaSwap Labs
27. Christopher Meadows
28. Aris Duani Rojas, Ph.D. Student
29. Sashri Brahmakshatriya
30. Natalia Valencia, Ph.D. Student
31. Kevin Nordman

Current Undergrad Students

32. Federico Monteverdi
33. Christopher Lukas Kverne
34. Amanda Di Perna

Independent Study

[1 M.S. + 3 Undergraduate]

1. Daniel Carlson, Summer 2022, Topic: Hybrid Memory Management.
2. Dwaraka Prasath Mohen Babu, Spring 2022, Topic: Data Structures to Identify Data Streams.
3. Sashri Brahmakshatriya, Summer 2021, Topic: Analyze Reliability of SSDs.
4. Christopher Meadows, Summer 2021, Topic: Design Data Stream Identifier.

Capstone Mentoring

1. Daniel Carlson
2. Patrick Perez
3. Oscar Barbosa
4. Nazmul Huq
5. Luis Acosta
6. Ettore Mottola
7. Eitan Flor
8. Bryan Camacho

TEACHING ACTIVITIES

Storage Systems (CIS 5346): The most recent offering of this course was in Fall 2023, and it underwent evaluation by students through the Student Perceptions of Teaching Survey (SPOTs), yielding a commendable mean score of 4.64/5. Notable comments from students include, *"The most successful aspect of this course is the detailed information in the lectures about the objectives and executions of each module and great instructor-student interactions."* Another student mentioned, *"The professor's availability and commitment towards every student was commendable."* Additionally, a student highlighted, *"This was the best online course that I have taken so far, expertly crafted, and the pacing of this course was good."* Lastly, a student appreciated the practicality, stating, *"The way we could relate the course to real-life scenarios will definitely help me never forget what I learned. Course material and discussions are thought-provoking and interesting."*

Data Structures (COP 3530): COP 3530 earns its "effort-intensive" label due to students' substantial growth expectations post-course and a high enrollment rate. In Fall 2023, my SPOTs rating was 4.19/5. Students praised COP 3530 for being *"well-structured,"* with one noting, *"The most successful aspect is probably having discussions for each Module."* Another student commended the *"Professor's teaching proficiency is 5+ stars rating, she presents lectures in a clear, understandable way with many opportunities in each module to get clarification."* The "video lectures" were highlighted for their *"easy-to-understand quality."* Overall, the course was recognized for its *"balance, pacing, and real-world applicability, making it a transformative and great learning experience."* The peer evaluation of this course by my colleague Dr. Masoud Milani stated that *"my observations, as well as students' opinions, confirm that Dr. Bhimani is an outstanding teacher who has clearly mastered the art of teaching."*

Graduate Courses Taught

Overall SPOTS rating: 4.26/5

1. CIS 5346: Storage Systems, Fall 2024, Fully-online modality, SPOTs- Number of student: , Response rate: , Overall average: .
2. CIS 7980: Ph.D. Dissertation, Summer 2024, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
3. CIS 5346: Storage Systems, Spring 2024, Fully-online modality, SPOTs- Number of student: 34, Response rate: , Overall average: .
4. CIS 7980: Ph.D. Dissertation, Spring 2024, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
5. CIS 5346: Storage Systems, Fall 2023, Fully-online modality, SPOTs- Number of student: 15, Response rate: 92.9, Overall average: 4.64/5.
6. CIS 7910: Graduate Research, Fall 2023, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
7. CIS 7980: Ph.D. Dissertation, Fall 2023, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
8. CIS 7980: Ph.D. Dissertation, Summer 2023, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.

9. CIS 5346: Storage Systems, Spring 2023, Fully-online modality, SPOTs- Number of student: 27, Response rate: 100%, Overall average: 3.8/5.
10. CIS 7980: Ph.D. Dissertation, Spring 2023, Hybrid modality, SPOTs- Number of student: 2, Response rate: NA, Overall average: NA.
11. CIS 7980: Ph.D. Dissertation, Fall 2022, Hybrid modality, SPOTs- Number of student: 2, Response rate: NA, Overall average: NA.
12. CIS 7980: Ph.D. Dissertation, Summer 2022, Hybrid modality, SPOTs- Number of student: 2, Response rate: NA, Overall average: NA.
13. CIS 5346: Storage Systems, Spring 2022, Fully-online modality, SPOTs- Number of student: 27, Response rate: 100%, Overall average: 4.11/5.
14. CIS 5900: Independent Study, Spring 2022, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
15. CIS 7980: Ph.D. Dissertation, Spring 2022, Hybrid modality, SPOTs- Number of student: 2, Response rate: NA, Overall average: NA.
16. CIS 7980: Ph.D. Dissertation, Fall 2021, Hybrid modality, SPOTs- Number of student: 2, Response rate: NA, Overall average: NA.
17. CIS 7980: Ph.D. Dissertation, Summer 2021, Hybrid modality, SPOTs- Number of student: 2, Response rate: NA, Overall average: NA.
18. CIS 7910: Graduate Research, Spring 2021, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
19. CIS 7980: Ph.D. Dissertation, Fall 2021, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
20. CIS 7910: Graduate Research, Fall 2020, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
21. CIS 7980: Ph.D. Dissertation, Fall 2020, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
22. CIS 7910: Graduate Research, Summer 2020, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
23. CIS 7910: Graduate Research, Spring 2020, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
24. CIS 7980: Ph.D. Dissertation, Spring 2020, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
25. CIS 5346: Storage Systems, Fall 2019, In-person modality, SPOTs- Number of student: 12, Response rate: 75%, Overall average: 4.48/5.

Undergraduate Courses Taught

Overall SPOTS rating: 4.00/5

1. COP 3530: Data Structures, Fully-online modality, Fall 2023, SPOTs- Number of student: 48, Response rate: 83.3, Overall average: 4.19/5.

2. COP 3530: Data Structures, Fully-online modality, Spring 2023, SPOTs- Number of student: 60, Response rate: 85.4%, Overall average: 3.64/5.
3. COP 3530: Data Structures, Fully-online modality, Fall 2022, SPOTs- Number of student: 47, Response rate: 85.4%, Overall average: 3.93/5.
4. CIS 3900 Independent Study, Summer 2022, Hybrid modality, SPOTs- Number of student: 1, Response rate: NA, Overall average: NA.
5. COP 3530: Data Structures, Fully-online modality, Spring 2022, SPOTs- Number of student: 39, Response rate: 69.2%, Overall average: 3.85/5.
6. COP 3530: Data Structures, Fall 2021, Fully-online modality, SPOTs- Number of student: 47, Response rate: 80.9%, Overall average: 4.2/5.
7. CIS 3900 Independent Study, Summer 2021, Hybrid modality, SPOTs- Number of student: 2, Response rate: NA, Overall average: NA.
8. COP 3530: Data Structures, Spring 2021, Fully-online modality, SPOTs- Number of student: 51, Response rate: 72.5%, Overall average: 4.43/5.
9. COP 3530: Data Structures, Fall 2020, Certified hybrid modality, SPOTs- Number of student: 39, Response rate: 84.6%, Overall average: 2.74/5.
10. COP 3530: Data Structures, Spring 2020, Hybrid modality, SPOTs- Number of student: 18, Response rate: 94.4%, Overall average: 4.13/5.
11. EECE 2560: Fundamentals of Engineering Algorithms (Northeastern University), Fall 2017, In-person modality, SPOTs- Number of student: 12, Response rate: 80%, Overall average: 4.8/5.

OTHER PROFESSIONAL ACTIVITIES AND PUBLIC SERVICE

FIU Internal Service

1. 2023-2024: KFSCIS Awards Committee
2. 2023-2024: CEC Faculty Council Representative
3. 2023-2024: KFSCIS Seminar Series Coordinator
4. 2023-2024: CEC Faculty representative at United Nations Women and Girls in Science day annual celebrations
5. 2023-2024: Graduate Council
6. 2022-2023: CEC Faculty Council Representative
7. 2022-2023: KFSCIS Seminar Series Coordinator
8. 2022-2023: Subject area coordinator: BS-CS for Programming: COP-2210, COP-3337, COP-3530, COP-4338, COP-4226, COP-4520
9. 2022-2023: KFSCIS Diversity, Equity & Inclusion (DEI) Committee
10. 2022-2023: CEC Faculty representative at United Nations Women and Girls in Science day annual celebrations

11. 2022-2023: Graduate Council
12. 2021-2022: KFSCIS Tenure Track Faculty Hiring Committee
13. 2021-2022: Subject area coordinator: BS-CS for Programming: COP-2210, COP-3337, COP-3530, COP-4338, COP-4226, COP-4520
14. 2021-2022: Capstone or Senior Projects Supervisor
15. 2021-2022: KFSCIS Diversity Advocate for Faculty Hiring
16. 2021-2022: CEC Faculty representative at United Nations Women and Girls in Science day annual celebrations
17. 2021-2022: Graduate Council
18. 2020-2021: KFSCIS Tenure Track Faculty Hiring Committee
19. 2020-2021: Subject area coordinator: BS-CS for Programming: COP-2210, COP-3337, COP-3530, COP-4338, COP-4226, COP-4520
20. 2020-2021: Capstone or Senior Projects Supervisor
21. 2020-2021: KFSCIS Diversity Advocate for Faculty Hiring
22. 2020-2021: KFSCIS Faculty representative at United Nations Women and Girls in Science day annual celebrations
23. 2020-2021: Graduate Council
24. 2020-2021: KFSCIS Graduate Committee
25. 2019-2020: CEC Faculty representative at United Nations Women and Girls in Science day annual celebrations
26. 2019-2020: Graduate Council
27. 2019-2020: KFSCIS Graduate Committee

FIU Microcredential

Remote Teach Ready Badge Summer 2020

Professional Activities

1. General Chair for ACM Workshop on Hot Topics in Storage and File Systems (HotStorage) 2024 leading the overall operations of the workshop.
2. TPC for USENIX Conference on File and Storage Technologies (USENIX FAST) 2024 with heavy review workload of 15-20 papers.
3. Program Committee Track chair for the track Performance Monitoring, Modeling, Analysis, and Benchmarking (in Cluster, Cloud and Internet Computing) at 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid) 2024.
4. Poster chair for the 33rd ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC) 2024.
5. Poster chair for the 32nd ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC) 2023.

6. Publicity chair and TPC for ACM Workshop on Hot Topics in Storage and File Systems (HotStorage) 2023.
7. TPC and Session chair for ACM Workshop on Hot Topics in Storage and File Systems (HotStorage) 2022 leading a session on ZNS and SSDs, 2022.
8. TPC and Session chair for USENIX Conference on File and Storage Technologies (USENIX FAST) 2021, leading a session on The SSD Revolution Is Not Over.
9. TPC and Session chair for ACM Workshop on Hot Topics in Storage and File Systems (HotStorage) 2022 leading a session on Flash Storage, 2022.
10. TPC and Session chair for IEEE International Symposium on Workload Characterization (IISWC) 2020, leading a session on System Architecture and Applications.
11. TPC for IEEE International Conference on CLOUD Computing, 2022
12. TPC for IEEE International Conference on Distributed Computing Systems (ICDCS), Machine Learning on or for Distributed Systems, 2022
13. TPC for IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid), Performance Modeling, Scheduling and Analysis Track, 2022
14. NSF Panelist for Cyberinfrastructure for Sustained Scientific Innovation (CSSI) program
15. TPC for IEEE International Parallel & Distributed Processing Symposium (IPDPS), 2020
16. TPC for IEEE International Performance Computing and Communications Conference (IPCCC), 2019, 2020

Service as Peer Reviewing

Conferences:

1. IEEE International Conference on Distributed Computing Systems (ICDCS)
2. IEEE/ACM International Symposium on Cluster, Cloud, and Internet Computing (CC-GRID)
3. ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)
4. International Symposium on High-Performance Parallel and Distributed Computing (HPDC)
5. USENIX Conference on File and Storage Technologies (FAST)
6. IEEE International Parallel & Distributed Processing Symposium (IPDPS)
7. IEEE International Conference on Cloud Computing (IEEE CLOUD)
8. IEEE High Performance Extreme Computing Conference (IEEE HPEC)
9. IEEE International Conference on Green Computing and Communications (GreenCom)
10. International Conference on Massive Storage Systems and Technology (MSST)
11. IEEE International Conference on Big Data (BigData)
12. International Conference on Networking, Architecture, and Storage (NAS)
13. International Conference on Parallel and Distributed Systems (ICPADS)

14. Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (IN-
FLOW)
15. International Conference on Performance Engineering (ICPE)
16. ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)
17. IEEE/IFIP International Conference on Dependable Systems and Networks (DNS)
18. Big Data and Cloud Performance Workshop at INFOCOM (DCPerf)
19. International Conference on Autonomic Computing (ICAC)
20. International Conference on Computer Aided Design (ICCAD)
21. International Conference on Cloud Computing Technologies and Applications (CloudTech)
22. Field-Programmable Custom Computing Machines (FCCM)
23. International Conference on Computer. Communication and Networks (ICCCN)
24. IEEE International Performance Computing and Communications Conference (IPCCC)
25. IEEE/ACM International Conference on Utility and Cloud Computing (UCC)

Journals:

1. IEEE Transactions on Cloud Computing (TCC), IEEE Journal
2. ACM Transaction on Storage (TOS), ACM Journal
3. IEEE Transactions on Services Computing (TSC), IEEE Journal
4. Simulation Modelling Practice and Theory (SIMPAT), Elsevier Journal
5. Computers, MDPI Journal
6. Future Generation Computer Systems (FGCS), Elsevier Journal
7. IEEE Transactions on Computers (TC), IEEE Journal
8. ACM Transactions on Modeling and Performance Evaluation of Computing Systems (TOM-
PECS), ACM Journal

Society Memberships

1. Member (2014-present) Association for Computing Machinery (ACM)
2. Member (2014-present) Institute of the Electrical and Electronics Engineers (IEEE)
3. Member (2014-present) The Advanced Computing Systems Association (USENIX)

Community Services

1. Volunteering Affiliated Faculty, Center for Women and Gender Studies (CWGS), Florida
International University, Miami FL, USA

PERSONAL TRAITS

Highly motivated and eager to learn new things.

Strong leadership skills and innovative approaches.

Ability to work as an individual as well as in group.

Research Statement

Janki Bhimani, Assistant Professor,

Knight Foundation School of Computing and Information Science, Florida International University

jbhimani@fiu.edu, bhimanijanki@gmail.com • [Linkedin](#) • [Website](#) • [Google Scholar](#) • Mobile: +1(857)991-9868

(Please visit my [Website](#) for the most updated information.)

The global datasphere is projected to reach 163 zettabytes by 2025, indicating a profound shift in our daily lives driven by data. As datacenters adapt to meet escalating computational and storage demands, the challenge of efficiently managing this abundance of information becomes critical for system architects. My decade-and-a-half-long research focuses on **pioneering simple yet disruptive and efficient solutions for memory management and storage systems to address this challenge**. The goal is to establish robust, parallel, and performant computing systems capable of thriving in our data-intensive landscape. This work not only aligns with current federal funding priorities but is imperative for shaping the future, where innovation and efficiency converge to redefine the boundaries of large-scale data-intensive workloads.

[Publication Decisions.] My work as a faculty member primarily focuses on storage systems, system design and architecture, modeling and simulation, and cloud computing domains. Consequently, I have directed my efforts towards the highest-ranked journals in these disciplines, including **ACM TOS, IEEE TC, ACM TOMACS, IEEE TCC, and IEEE TVT**. For instance, the IEEE Journal of Transactions in Cloud Computing has an impact factor of 11.1, and most of these journals publish less than 10% of submitted manuscripts. Due to the often lengthy time-to-press for journal publications and the time-intensive nature of qualitative research, I frequently target preliminary findings for highly-selective peer-reviewed conferences and workshops such as **HPCA, DAC, DATE, HotStorage, CLOUD, and HPDC**. This approach enables me to disseminate preliminary findings quickly to target audiences while also building a narrative for subsequent journal publications. Currently, I have **45 peer-reviewed publications**, with 20 published after joining FIU in the last five years. I am also the lead inventor of **10 patents**. My work has received multiple **Best Paper Awards** and nominations for the prestigious **IEEE MICRO Top Picks Award** and **In Company of Women Award** in the Science and Technology category, showcasing its impact. Additionally, my research is highly cited, with about **1000 citations**, an h-index of 16, and an i10-index of 20, reflecting broad influence. Recognizing the excellent quality of my research, I am a recipient of the **NSF CAREER Award, FIU Top Scholar Award** in the category of- Research, Creative Activities, and Award-Winning Publications, and the **Outstanding Research Award** by the KFSCIS. I have also been invited to speak on my research by various organizations, including the Entrepreneurs' Organization of Miami and industry research labs such as IBM and Samsung Research.

[Research Funding.] In my research, I have achieved significant milestones by securing a total of **twelve grants** valued at over **\$4.9 million**, with nine of them valued at over **\$1.8 million** as the lead/sole Principal Investigator from federal, state, and industry sources such as **NSF, Cyber Florida, and Samsung Semiconductors**. Notably, I am proud of my recent highly competitive **NSF CAREER Award** towards designing efficient in-storage indexing techniques, and **NSF CISE Core** small award as the **sole PI**, focusing on designing new memory management techniques for in-memory analytic frameworks and databases leveraging machine learning (ML). These funding's underscores the significance of my research work in the national discourse. Currently, I am awaiting decisions on **five proposals**, two of which I lead. Looking forward, my plans involve pursuing grants such as DOE CAREER, NSF Core Medium, and AFOSR to further advance my research.

[Research Collaborations and Advising.] My commitment to advising and mentoring is reflected in the successful graduation of **one Ph.D. student** who is a tenure-track Assistant Professor at Missouri State University. Over the past five years, I have advised a diverse group, including eight Hispanics, one Asian American, and three Women. Currently, I am supervising a dynamic team comprising **six graduate** and **three undergraduate** students and I am in process of **hiring two more graduate students** starting Fall 2024. Establishing fruitful collaborations is integral to my research approach. I have established partnerships with esteemed institutions such as the **University of Maryland, University of Chicago, Argonne National Lab, and Syracuse University**, as well as industry leaders like **Samsung and IBM Research**, contributing to the advancement of our collective research goals.

[Research Experience and Successes.] My research philosophy is centered on advancing "end-to-end system design" to maximize the utilization of available resources and effectively address societal challenges. I am deeply passionate about optimization and emerging technologies. Engaging in interdisciplinary research is a particular interest of mine, and I consider benchmarking and modeling as essential tools for conducting thorough investigations and driving disruptive improvements. At a high level, my research has concentrated on improving two critical components of computer systems: Memory and Storage. Over the last five years as a faculty member, my research contributions can be categorized primarily into the following three directions.

I/O Behavior Modeling, Performance Prediction and Optimization: In the era of "Big Data," where multiple data processing applications coexist in data centers, I/O activities exhibit significant variations. The configuration of existing storage systems, typically done during installation and then permanently maintained, is becoming insufficient. Modern data processing systems present three key challenges. First, simultaneous operations from multiple applications create interference, impacting the performance of solid-state drives (SSDs), unlike hard disk drives (HDDs) with limited bandwidth. Second, while HDD reliability relies on internal mechanical components, SSDs are sensitive to user and

operating system I/O workloads, necessitating optimal configuration for persistent storage. Third, diverse SSD types, including multi-stream SSDs and Key-Value (KV) SSDs, come with distinct internal algorithms and parameters. Selecting and tuning these algorithms based on I/O activities is crucial for optimal performance and flash endurance. Therefore, the research objective is to **model the complex I/O activities of diverse applications, dynamically tuning the internal algorithm parameters of flash-based SSDs for optimal performance and reliability.**

Learning and Management in Tiered Memory Systems: Further, in the past decade, ML has undergone astounding growth, permeating various industries, including storage systems. To tackle the challenges presented by vast amounts of data and optimize memory accesses, tiered memory systems are gaining popularity. These systems employ high-speed memory like DRAM for frequently accessed upper-tier data and slower but larger memories like NVMe NAND flash, 3D-Xpoint, and CXL memories for lower tiers. These systems prove crucial for efficient data management, contributing to improved performance and efficiency, reducing data access times, and lowering overall computing costs. Thus, we **design novel tiered memory management techniques, leveraging ML’s power and addressing its limits and overheads as a versatile solution to enhance various aspects, including parameter tuning, task scheduling, scanning, migration, and allocation, ultimately optimizing performance, Quality of Service (QoS), and resource utilization in in-memory databases and analytic frameworks.**

Towards Efficient In-storage Indexing and Device Endurance: In the domain of in-storage indexing, a concept originating in the 1990s, the practical implementation has encountered obstacles, as exemplified by Seagate’s 2014 endeavor with Kinetic HDDs, constrained by HDD I/O limitations. However, the prospect of efficient in-storage indexing has recently gained momentum with flash-based SSDs. Early efforts, including our preliminary research, demonstrate that the combination of a fast and lightweight KV database or POSIX-compliant file system with a key-value SSD (KVSSD) performing in-storage indexing outperforms traditional block SSDs. The development of Efficient In-Storage Indexing Devices (ISIDs) poses challenges that necessitate attention to ensure optimal performance and functionality. Firstly, the role of storage device models is pivotal in computer systems research, addressing research gaps related to performance analysis, algorithm development, system evaluation, resource management, and realistic simulations. The lack of a low-cost open-source research platform hampers rapid adoption. Secondly, the design of ISIDs for diverse workloads demands meticulous consideration of indexing techniques, query optimization, data access patterns, and data distribution within the constraints of limited device resources. Therefore, we **develop ISID models that capture internal feature dependencies and support dynamic model calibration. This aims to develop new index management techniques efficiently utilizing limited on-device resources while considering flash-specific constraints to optimize endurance and latency for a multi-tenant environment.**

In conjunction with the above, my team has embarked on pursuing various other cutting-edge projects. First, we challenge the assumption that flash-based SSDs are less susceptible to diverse environmental conditions such as vibrations, temperature, and humidity than HDDs. Extensive testing indicates that, even within specified datasheet limits, short-term exposures exhibit lingering effects, and long-term exposure results in over **30%** performance degradation. This raises concerns in datacenter performance, affecting tail latency and Service Level Agreements (SLAs) but has even more crucial implications on automotive operating in harsh conditions, with its transition to use more and more complex electronic control systems (ECS) with integrated flash. Leveraging these insights, with a particular focus on the impact of temperature on the reliability of electronic control systems (ECS), we study automotive applications with multi-core processing architectures, taking into account temperature considerations and system-level reliability. We **optimize system-wide reliability through a mathematical programming model and a genetic algorithm, accurately predicting system-wide mean-time-to-failure (MTTF) with substantial speed-ups, thereby enhancing reliability analysis.**

Second, another initiative involves **critically examining wear leveling in SSDs, addressing challenges, assessing effectiveness, and advocating for capacity variance.** Third, in the space of **Systems-for-ML**, our team is designing **new models to enhance data storage and ingestion pipelines for ML workloads.** We are also investigating **novel approaches to optimize Deep Neural Network (DNN) checkpointing and versioning** beyond traditional file I/O methods and designing models. Forth, in the space of **ML-for-Systems**, we are leveraging **disciplined data science to revolutionize decision-making in storage I/O.** Fifth, we are working towards **optimizing data spilling in distributed query engine.** Lastly, my commitment extends to advocating for positive change in educational systems through the NSF HSI project **VOCES - Voices for Organizing Change in Educational Systems**, leading efforts in the direction of social media impacts and improving course curriculum. *Thus, to conclude, my diverse undertakings showcase my comprehensive engagement in advancing knowledge and addressing pivotal challenges across multiple facets of data storage and computing as well as the education system.*

[Future Plan.] In my envisioned trajectory, I am committed to further solidifying my standing as an expert in memory management and storage systems, with a dedicated focus on emerging technologies. My work will continue to contribute to and influence the national discourse around efficient, reliable, and enduring data management techniques. Each successive phase of my research is strategically crafted to make practical and measurable strides towards optimizing data management. My dedication to solving important research problems and consistently publishing in high-quality venues remains unwavering. I plan to enhance my recruitment strategies, aiming to intentionally diversify my doctoral student cohort. Looking forward, I aspire to leverage the national network I have cultivated to foster collaborations on a broader scale. Initiating endeavors for large-scale, multi-institutional funding is a pivotal goal, intended to amplify the impact of my research endeavors and revolutionize data management practices.

Teaching Statement

Janaki Bhimani, Assistant Professor,

Knight Foundation School of Computing and Information Science, Florida International University

jbhimani@fiu.edu, bhimanijanki@gmail.com • [Linkedin](#) • [Website](#) • [Google Scholar](#) • Mobile: +1(857)991-9868

(Please visit my [Website](#) for the most updated information.)

Designed and taught **three core courses**, achieving an **average student feedback rating of 4.13/5**. Secured **Quality Matters (QM) certification** for all my courses. Led curriculum update efforts to design and integrate a Data Structures course taught in multiple programming languages. Implemented **module-based content distribution** with live feedback, and class projects on **Chameleon cloud platform**.

Teaching Philosophy

As an Assistant Professor, I am thrilled by the opportunity to fully integrate teaching and mentoring into my daily routine. My goal is to motivate students to acquire and apply knowledge, teach them critical thinking skills to solve newly occurring problems in their fields of study, and assist them in building career foundations to succeed after graduation. I emphasize to students that the specifics of all the programming languages, libraries, frameworks, and deployment services that are in vogue at the moment can change at a rapid pace. *All new technologies are easy to learn once students have developed strong foundations.* Apart from the technical depth of content, I also encourage everyone in the classroom to express their thoughts. To bring technical subjects to life, I teach using a combination of various techniques such as whiteboard discussions, PowerPoint presentations, and audio-visual illustrations, such as a demo of sorting algorithms using UNO cards. A sample of my demonstrative lecture can be found [here](#).

I believe that students learn more by thinking than by memorizing. Therefore, before delving into any new topic, I explain the key challenges and questions that initially motivated its exploration and how various ideas evolved over time to nurture problem-solving capabilities. For example, in [this](#) lecture on solid-state drives, I first explain why students should learn about the topic, highlighting its exciting advantages and key challenges that motivated various branches of different research works.

In the growing and ever-changing field of computer science and engineering, I view teaching not only as the transmission of knowledge to students but also as a means of inspiring independent inquiry and learning. *It is a collaborative process that fosters improvement for both students and me.* To illustrate how I embody these principles in my lectures, consider [this](#) lecture on NAND cells within SSDs. As students gain insights into the complexities of NAND cells, I, too, gain a deeper understanding through the process of explanation, fostering a continuous cycle of learning and teaching in the ever-changing landscape of computer science and engineering. Recognizing the uniqueness of each student, I have learned and will continue to tailor my mentoring style to bring out the best in every individual.

Teaching Experience

At FIU I have designed two courses from scratch and taught multiple semesters of graduate and undergraduate courses in different modalities, including in-person, certified hybrid, and certified synchronous and asynchronous online formats. The courses I taught include Storage Systems and Data Structures, Independent Study, Senior Project, and Capstone Project. For all my courses, I have developed module-based content that provides the necessary scaffolding to span the space for better understanding of the concepts learned. With input from peers at the Center for the Advancement of Teaching (CAT), colleagues within our department, teaching assistants, and student feedback, I have taken several measures to enhance the teaching process. These include developing a precursor covering basic concepts before the start of the course, creating a mechanism to incorporate student feedback during the same semester, and crafting a well-structured syllabus with course-level objectives. Additionally, I have outlined module-level objectives and explained their alignment with course-level objectives, improved course content, and designed clear rubrics for all assignments, making it easier for students to understand expectations. Next, I elaborate the content of some of my courses, highlighting my favorite parts.

Storage Systems (CIS 5346): Teaching the advanced graduate-level Storage Systems course has been a stimulating academic pursuit. This course delves into a diverse array of topics, encompassing the introduction to storage systems, storage devices (Hard Disk Drives, Solid State Drives), storage system components, storage architecture, large-scale distributed storage systems, datacenter storage, non-volatile memory (NVM), reliability and fault tolerance (RAID Systems), performance, file-systems, operating systems storage management, memory and storage concepts (Caching, Consistency, and Deduplication), disks and scheduling, and emerging storage technologies and future trends.

One of the key challenge to efficiently teach this course extends beyond disseminating theoretical knowledge; it lies in cultivating a profound curiosity for the rapidly evolving landscape of storage technologies. Through hands-on projects and real-world case studies, my aim is to facilitate a dynamic learning experience reflective of the practical demands within the field. The class structure combines traditional instruction with seminar-style learning, fostering a dynamic educational environment. Each student is tasked with extensively preparing and leading discussions on a research paper, enhancing their ability to critically evaluate and present findings to their peers. This multifaceted approach aims to nurture a cohort of learners equipped not only with theoretical comprehension but also with a problem-solving mindset, preparing them to confront the evolving challenges in the technological landscape.

The most recent offering of this course underwent evaluation by students through the Student Perceptions of Teaching Survey (SPOTs), yielding a commendable mean score of **4.64/5**. Notable comments from students include, "The most successful aspect of this course is the detailed information in the lectures about the objectives and executions of each module and great instructor-student interactions." Another student mentioned, "The professor's availability and commitment towards every student was commendable." Additionally, a student highlighted, "This was the best online course that I have taken so far, expertly crafted, and the pacing of this course was good." Lastly, a student appreciated the practicality, stating, "The way we could relate the course to real-life scenarios will definitely help me never forget what I learned. Course material and discussions are thought-provoking and interesting." The positive feedback reflects the effectiveness of the course structure and content delivery, as well as my commitment to facilitating an engaging and enriching learning experience.

Data Structures (COP 3530): Teaching COP 3530 has been fulfilling, impacting students' job interviews. This course starts with a "Brush-up Your C++" module and progressing to Big-O complexity, stacks, queues, linked lists, searching and sorting algorithms, graph and tree algorithms, recursion and backtracking, and hash tables. The curriculum includes quizzes, discussions, assignments, and a final exam, incorporating real-world applications and practical problem-solving scenarios. With the aim to enhance students' understanding of memory management, particularly beneficial for system-related courses like operating systems and high-performance computing, I designed this course from scratch in C++. Having completed the hybrid certification process, I taught my Data Structures course in a hybrid modality for multiple semesters. Initially, I found it challenging to motivate students for out-of-class components. To address this, I conducted surveys and polls on each module to gauge student understanding of various out-of-class topics. Based on the poll responses, I curated the time I spent on each topic and revised the topics that most students found challenging. This approach significantly improved student interest and performance.

COP 3530 earns its "effort-intensive" label due to students' substantial growth expectations post-course and a high enrollment rate. In Fall 2023, my SPOTs rating was **4.19/5**. Students praised my class for being "well-structured," with one noting, "The most successful aspect is probably having discussions for each Module." Another student commended the "Professor's teaching proficiency" with a "5+ stars" rating, emphasizing, "The professor is great at conducting and organizing an online course." My "video lectures" were highlighted for their "easy-to-understand quality." Overall, the course was recognized for its "balance, pacing, and real-world applicability, making it a transformative and great learning experience."

Independent Study (CIS 5900 and CIS 3900): In my role as the instructor for Independent Study courses CIS 5900 and CIS 3900, my primary responsibility is to facilitate a self-directed and personalized learning journey for each student. Through regular individual meetings, I offer tailored guidance and support, recognizing and addressing the unique academic goals and interests of each participant. The course framework includes assigned readings that lay the groundwork for independent investigations, culminating in students producing comprehensive reports documenting their research findings. The inherent challenges of this instructional role involve instilling a sense of accountability and motivation in students navigating their project independently. Achieving the delicate balance between providing sufficient guidance and allowing for student autonomy is critical to ensuring a fruitful independent study experience. Equally important is maintaining open communication channels and promptly addressing any issues that may arise during the independent learning process. These efforts collectively contribute to fostering a supportive and enriching educational environment within the framework of independent study.

Overall, many students have found success in securing co-op, internship, and full-time opportunities with top industries such as Google, Microsoft, Samsung, etc., after taking my courses and working with me. Witnessing my students' achievements brings me great joy, and I take pride in their success.

Mentoring Experience

In my role as a mentor, I have been deeply committed to fostering an environment where students can not only gain knowledge but also thrive in their research pursuits. I draw inspiration from the supportive mentors who guided me throughout my career. I aim to pass on not just information but also provide a nurturing and engaging space for students to develop their research abilities. Having mentored both undergraduate and graduate students, I understand the importance of effective communication tailored to each student's needs. I recognize that the level of involvement in technical details varies among students, and I strive to strike a balance between offering prompt feedback and allowing for independent exploration. My goal is to nurture students' research independence while creating a collaborative and inclusive atmosphere. In terms of academic supervision, I take pride in the success of my graduated Ph.D. and M.S. students who have secured positions in reputable companies or obtained tenure-track positions. Currently, I am advising three graduate and three undergraduate students, each delving into innovative research topics encompassing storage systems, memory management, ML-for-Systems, Systems-for-ML, and cloud computing. Additionally, I actively participate in Ph.D. committees for students from various universities, contributing to their academic growth. My mentoring extends to independent study projects, where I guide students through topics such as hybrid memory management and SSD reliability. In REU programs, I provide support and guidance to students, fostering a supportive learning environment. Furthermore, I offer direction to students in capstone projects, ensuring their successful completion and preparing them for future challenges. Overall, my mentoring experiences reflect a commitment to students' growth, emphasizing collaboration, communication, and a focus on their individual development as researchers.

Service Statement

Janki Bhimani, Assistant Professor,

Knight Foundation School of Computing and Information Science, Florida International University

jbhimani@fiu.edu, bhimanijanki@gmail.com • [Linkedin](#) • [Website](#) • [Google Scholar](#) • Mobile: +1(857)991-9868

(Please visit my [Website](#) for the most updated information.)

Contributed to FIU's growth, serving on the **Faculty Hiring Committee**, **Awards Committee**, **CEC Faculty Council**, **Subject area coordinator**, **Graduate Committee**, **DEI Committee**, and as **Seminar Series Coordinator**. Led roles such as **General Chair**, **Publicity Chair** for ACM HotStorage, **Program Committee Track Chair** for CCGRID, and **Poster Chair** for HPDC. Extensive service as a **TPC member** for conferences like USENIX FAST, IPDPS, CLOUD. Volunteering at the **Center for Women and Gender Studies**. Received **Grace Hopper Celebration of Women in Computing (GHC) Faculty Scholarship** for two years and Certificate of Completion from **ASEE DELTA Junior Faculty Institute**. Participated in **STRIDE workshop** for hiring, tenure, and promotion, **Diversity Advocate** workshop, and **Bystander Leadership** workshop.

Service Philosophy

Service is the bedrock of our academic community, propelling us toward excellence. Beyond obligation, it embodies our commitment to a collegial environment and self-governance. My primary commitment is active engagement in KFSCIS, CEC, and FIU, fostering positive change. I aim to broaden my impact globally in system design and storage system research, connecting with researchers to bring valuable insights back to our academic home. I view my service commitments not as burdens but as golden opportunities for learning, growth, and the demonstration of my inherent value to the academic community. Each engagement is a chance to refine my leadership skills, to cultivate relationships, and to contribute meaningfully to the success of the University. As a steward of progress, I am dedicated to shaping our academic future through unwavering commitment to service.

Service Roles and Experience

School Service: My commitment to service within the KFSCIS is evident through my extensive involvement in various roles, each contributing significantly to the vibrancy and excellence of our academic community. Serving on the KFSCIS Tenure Track Faculty Hiring Committee for two consecutive years has been a distinguished responsibility, offering me the opportunity to actively shape the trajectory of our department. Throughout these two years, we faced the substantial task of hiring for multiple open tenure track positions, entailing a considerable amount of work. This included efforts to advertise faculty openings in both traditional and non-traditional minority community venues, a meticulous review of over 100 applications, the orchestration of multiple rounds of Zoom interviews for approximately 20 candidates, and frequent committee meetings, often exceeding twice a week. My role extended beyond the virtual interview process to coordinating in-person interviews with candidates. For many faculty interview candidates, I willingly took on various duties, from facilitating morning pickups to guiding them through a series of meetings and talks, accompanying them for lunch, showcasing our wonderful campus, organizing dinner engagements, exploring various areas around Miami based on their living preferences, and ensuring their safe return to the hotel in the evening. The magnitude of this effort was instrumental in the successful hiring of eight new tenure-track faculty members. Highlighting the significance of diversity and equity in the faculty selection process, I actively fostered an inclusive environment, recognizing the value that diverse perspectives bring to our academic community. To prepare for this crucial role, I engaged in workshops and certificate programs, such as ASEE DELTA Junior Faculty Institute, STRIDE workshops for hiring, STRIDE workshops for tenure and promotion, Diversity Advocate workshop, and Bystander Leadership workshop. The noteworthy female faculty additions to our faculty during the two years of my service include Dr. Ruimin Sun, Dr. Agoritsa Polyzou, and Dr. Wenqian Dong, underscoring the success in enriching the diversity of our academic team. In these two years of my intense commitment to the Tenure Track Faculty Hiring Committee, we were successful in hiring eight new tenure track faculty members. Thus, I not only contributed to the growth of our department but also played a pivotal role in cultivating a faculty body that reflects a broad spectrum of expertise and backgrounds essential for academic excellence.

Furthermore, as a member of the KFSCIS Graduate Committee for two years, I actively contributed to the review of Ph.D. student applications and met with many of them to address questions about applying to FIU. Our decisions as members of the graduate committee significantly impact the academic journey of our graduate students, ensuring that we maintain a high standard of education and support for their research endeavors. Additionally, I served on our Diversity, Equity & Inclusion (DEI) Committee for one year, where I led the process of preparing the first draft of the 2025-2028 NSF Broadening Participation (BPC) plan. In my role as the Subject Area Coordinator for the BS-CS Programming track, responsible for courses such as COP-2210, COP-3337, COP-3530, COP-4338, COP-4226, and COP-4520, I consistently aimed to provide comprehensive and timely reports by reviewing course evaluations. During my two years of service in this role, we underwent ABET evaluations, and I dedicated my best efforts to meet with evaluators and provide them with all requested materials to facilitate the process. Additionally, I served as the Seminar Series Coordinator, where coordinating the KFSCIS Seminar Series transcends mere event planning. This responsibility involves intricate logistics and hosting distinguished speakers, including Turing award winners and

members of national academies. The series not only fosters intellectual exploration but also connects our academic community with influential figures in computer science, broadening our collective knowledge and impact. I also served as a member of the KFSCIS Awards Committee, actively participating in the recognition of outstanding achievements within our academic community.

College Service: At the college level, my commitment to service has manifested through my engagement in various roles over the past five years. Serving as the CEC Faculty Council Representative, I actively participated in discussions and decision-making processes, representing the voice and concerns of the faculty within the College of Engineering and Computing (CEC). This role has provided me with valuable insights into the broader academic landscape of the college, allowing me to contribute to collegial governance and the enhancement of our academic programs. Simultaneously, my involvement on the Graduate Council has allowed me to play a pivotal role in shaping the policies and initiatives related to graduate education within the college. By contributing to discussions and decisions at this level, I have been able to actively advocate for the interests of graduate students and ensure the maintenance of high standards in our graduate programs. As the faculty representative at the United Nations Women and Girls in Science Day annual celebrations, I have extended my commitment to service beyond the confines of the college. Actively participating in events and activities dedicated to promoting the role of women and girls in science, I have contributed to fostering an inclusive and supportive environment within the college and beyond. These roles collectively demonstrate my commitment to service, collegial governance, and fostering an inclusive academic environment.

University Service: My commitment to service extends beyond school and college-level responsibilities, encompassing initiatives that actively contribute to the dynamic and inclusive ethos of the university. I have actively engaged in professional development, completing a hybrid program to become a certified hybrid instructor, embracing innovative teaching methods for a modern academic landscape. In addition, I volunteer as affiliated faculty for the Center for Women and Gender Studies (CWGS), where I dedicate efforts to promote women in computer science. This role allows me to contribute to the broader discourse on gender diversity in STEM fields and advocate for inclusivity within the university community. Each responsibility not only contributes to the betterment of our department, college, and university but also reflects my steadfast commitment to the values of service, collaboration, and excellence.

External Service: In external service, my active engagement across various roles underscores a dedicated commitment to advancing the fields of storage systems, cloud computing, distributed computing, and performance modeling. In my leadership roles, I served as the General Chair for the ACM Workshop on Hot Topics in Storage and File Systems (HotStorage) in 2024, spearheading the workshop's overall operations and contributing to its success. Simultaneously, I served as the Program Committee Track Chair for the Performance Monitoring, Modeling, Analysis, and Benchmarking track at the 22nd IEEE International Symposium on Cluster, Cloud, and Internet Computing (CCGrid) in 2024. My involvement extended to serving as the Poster Chair for the 33rd and 32nd ACM International Symposium on High-Performance Parallel and Distributed Computing (HPDC) in 2024 and 2023, respectively. I played a crucial role as the Publicity Chair and TPC for the ACM Workshop on Hot Topics in Storage and File Systems (HotStorage) in 2023. Additionally, I assumed roles as TPC and Session Chair for both the ACM Workshop on Hot Topics in Storage and File Systems (HotStorage) and the USENIX Conference on File and Storage Technologies (FAST) in 2022, leading sessions on topics such as ZNS and SSDs, and "The SSD Revolution Is Not Over."

Parallel to these conference leadership commitments, my dedication to peer reviewing is evident across prestigious conferences. I took on the demanding role of Technical Program Committee (TPC) member for the USENIX Conference on File and Storage Technologies (FAST) in multiple years, handling a substantial review workload comprising 15-20 papers. I served on the TPC of IEEE International Conference on Distributed Computing Systems (ICDCS), IEEE/ACM International Symposium on Cluster, Cloud, and Internet Computing (CCGRID), ACM Workshop on Hot Topics in Storage and File Systems (HotStorage), International Symposium on High-Performance Parallel and Distributed Computing (HPDC), IEEE International Parallel & Distributed Processing Symposium (IPDPS), and more over the last five years. My contributions also extend to reviewing papers submitted to reputable journals like IEEE Transactions on Cloud Computing (TCC), ACM Transactions on Storage (TOS), IEEE Transactions on Services Computing (TSC), and others. Beyond conferences and journals, I serve as an NSF panelist for the Cyberinfrastructure for Sustained Scientific Innovation (CSSI), Office of Advanced Cyberinfrastructure (OAC), and Computer and Information Science and Engineering (CISE) programs, demonstrating my commitment to broader research initiatives.

Engaging in outreach, I serve as a guest speaker to inspire scholars in programs like the flit-path scholars' program and the Women in Cybersecurity (WiCys) Student Chapter. I have also delivered invited talks on my research at high-impact events hosted by organizations such as the Miami Entrepreneurs' Organization (EO), IBM Research, and Samsung Research. Moreover, I prioritize continuous professional development, completing a hybrid program to become a certified hybrid instructor. Acknowledging the importance of integrating scholarship, teaching, and service, I obtained a certificate of completion from the ASEE DELTA Junior Faculty Institute. Further enhancing my perspective, I have been awarded faculty scholarships to attend the Grace Hopper Celebration of Women in Computing (GHC) for two consecutive years. These experiences collectively enrich my capabilities and contribute to my multifaceted role as an educator and researcher.

[Future Plans:] Moving forward I plan to continue to support endeavors at all university levels and leverage existing relationships to extend my national and international involvement in the research community.

PUBLICATION SAMPLES

Representative Journal Publications

IEEE Transactions on Cloud Computing (TCC)
IEEE Transactions on Vehicular Technology (TVT)
ACM Transactions on Storage (TOS)
IEEE Transactions on Computers (TC)
ACM Transactions on Modeling and Computer Simulation (TOMACS)

Sample of Peer Reviewed Conference and Workshop Proceeding Publications

IEEE International Symposium on High-Performance Computer Architecture (HPCA)
ACM Workshop on Hot Topics in Storage and File Systems (HotStorage)
Design, Automation and Test in Europe Conference (DATE)
IEEE International Conference on Cloud Computing (CLOUD)
ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC)

Learning-Based Dynamic Memory Allocation Schemes for Apache Spark Data Processing

Danlin Jia , Li Wang, Natalia Valencia, Janki Bhimani, Bo Sheng, and Ningfang Mi

Abstract—Apache Spark is an in-memory analytic framework that has been adopted in the industry and research fields. Two memory managers, Static and Unified, are available in Spark to allocate memory for caching Resilient Distributed Datasets (RDDs) and executing tasks. However, we find that the static memory manager (SMM) lacks flexibility, while the unified memory manager (UMM) puts heavy pressure on the garbage collection of the JVM on which Spark resides. To address these issues, we design a learning-based bidirectional usage-bounded memory allocation scheme to support dynamic memory allocation with the consideration of both memory demands and latency introduced by garbage collection. We first develop an auto-tuning memory manager (ATuMm) that adopts an intuitive feedback-based learning solution. However, ATuMm is a slow learner that can only alter the states of Java Virtual Memory (JVM) Heap in a limited range. That is, ATuMm decides to increase or decrease the boundary between the execution and storage memory pools by a fixed portion of JVM Heap size. To overcome this shortcoming, we further develop a new reinforcement learning-based memory manager (Q-ATuMm) that uses a Q-learning intelligent agent to dynamically learn and tune the partition of JVM Heap. We implement our new memory managers in Spark 2.4.0 and evaluate them by conducting experiments in a real Spark cluster. Our experimental results show that our memory manager can reduce the total garbage collection time and thus further improve Spark applications' performance (i.e., reduced latency) compared to the existing Spark memory management solutions. By integrating our machine learning-driven memory manager into Spark, we can further obtain around 1.3x times reduction in the latency.

Index Terms—JVM memory management, distributed data processing, machine learning, Apache Spark, Q-learning.

I. INTRODUCTION

THE unprecedented proliferation of data has triggered a significant development of scalable analytics stacks in recent years. Developers and researchers strive to boost data-processing

Manuscript received 10 April 2023; revised 31 August 2023; accepted 28 September 2023. This work was supported by the National Science Foundation (NSF) under Grants CNS-2008324, CNS-2323100, CNS-1452751, and CNS-2008072. Recommended for acceptance Dr. by J. Zhai. (Corresponding author: Danlin Jia.)

Danlin Jia, Li Wang, and Ningfang Mi are with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115 USA (e-mail: jia.da@northeastern.edu; wang.li4@northeastern.edu; ningfang@ece.neu.edu).

Natalia Valencia and Janki Bhimani are with the School of Computing and Information Sciences, Florida International University, Miami, FL 33199 USA (e-mail: nvale010@fiu.edu; jbhimani@fiu.edu).

Bo Sheng is with the Department of Computer Science, University of Massachusetts Boston, Boston, MA 02125 USA (e-mail: bo.sheng@umb.edu).

Digital Object Identifier 10.1109/TCC.2023.3329129

speed in hardware and software. However, processing a massive volume of data has entirely relied on the performance of computing facilities and the efforts of users and can only achieve a suboptimal performance [1]. Thus, distributed frameworks (e.g., Hadoop [2]) that share computational resources on a cluster have been proposed to handle the overwhelming data. However, it has been noticed that in Apache Hadoop, many I/O requests are generated for accessing the intermediate data. To address this issue, in-memory analytic frameworks (e.g., Apache Spark [3]) have been developed to improve data processing performance.

Apache Spark [3], one of the most successful in-memory analytic frameworks, has been going through a boom in the past few years. Specifically, Apache Spark implements an abstraction of a data structure called Resilient Distributed Datasets (RDD) [4], which can be manipulated in parallel on different executors. Each RDD is created from an input dataset or another RDD and is immutable. Based on these two features, Spark builds a lineage of an application to track each computation stage and recover from faults in a tolerant way. Furthermore, Spark stores intermediate data (i.e., RDDs) in RAM, which reduces communication overhead between Spark executors, especially for some iterative and interactive machine learning applications. In this way, Spark avoids the overhead of I/O operations and improves overall performance. Therefore, one of the most crucial factors in Spark is the management of memory resources. An effective memory management scheme can shrink an application's latency (i.e., the total execution length) and improve performance dramatically. Unfortunately, Apache Spark hides the default scheme in memory management from users, who have few opportunities to monitor and configure the memory space.

In this work, we first investigate two existing Spark memory managers: Static memory manager (SMM) and Unified memory manager (UMM). Specifically, SMM applies predefined configurations to allocate fixed memory partitions for Spark applications, which heavily relies on the user's efforts and knowledge of the application's characteristics for memory optimization. On the other hand, UMM can dynamically allocate memory based on the run-time memory demands. However, UMM introduces heavy Garbage Collection (GC) as it tends to overprovision memory for runtime objects. We further run representative data processing benchmarks to collect the latency of applications under these two memory managers. We find that the Spark performance is significantly affected by the memory partition, which may lead to either long Java garbage collection (GC) or long delay in intermediate data access. Based on the analysis of the defects of the existing memory managers, we design a

learning-based bidirectional usage-bounded memory management scheme that monitors the run-time execution performance and dynamically re-allocates memory space to Spark execution and RDD storage. We first propose a basic version of our new autotuning memory manager, named *ATuMm*, which leverages an intuitive feedback-control solution to improve Spark performance by dynamically adjusting memory pools with a fixed adjustment step.

To obtain an optimal learning speed, the users of *ATuMm* need to tune the adjustment step manually. However, it is not trivial to configure this adjustment step. Significantly when the memory demands of an application vary frequently, an inappropriate adjustment step might limit the benefit of *ATuMm*. To address this issue, we further propose a Q-learning-based Spark memory manager, called *Q-ATuMm*, which aims to develop an intelligent agent to help make decisions of the adjustment step automatically. The goal of *Q-ATuMm* is to utilize a machine learning algorithm (e.g., Q-learning [5]) to adjust memory partitions in Spark dynamically and efficiently. We remark that Q-learning offers several advantages compared to other machine learning algorithms, especially in scenarios involving sequential decision-making and dynamic environments.

The main contributions of this work are as follows.

- *Understanding of two existing memory managers in Spark:* We study the infrastructure of two Apache Spark memory managers to understand how these two managers allocate memory space to the storage and execution pools. We further conduct real experiments to analyze the performance of these two managers.
- *Design and implementation of an auto-tuning memory manager:* We propose a new Spark memory manager, named *ATuMm*, that dynamically tunes the size of storage and execution memory pools based on the performance of current and previous tasks. We implement and evaluate *ATuMm* in Spark 2.4.0 and show that our new memory manager significantly improves the Spark performance.
- *Optimization of memory management by developing an intelligent agent:* We develop an intelligent agent by using the Q-Learning algorithm and integrate the agent in Spark as a new memory manager, named *Q-ATuMm*. We show that *Q-ATuMm* can further improve the performance via our new machine learning agent for both iterative data processing applications and ad-hoc database queries.
- *Analysis of memory usage and GC of Spark memory managers:* We investigate the execution memory usage and garbage collection of all four Spark memory managers (i.e., SMM, UMM, *ATuMm*, and *Q-ATuMm*). We discover that both *ATuMm* and *Q-ATuMm* decrease garbage collection time by preventing overloaded execution memory. Also, we observe that *Q-ATuMm* has lower latency than *ATuMm*.

In the remainder of this paper, we will discuss the issues of two existing memory managers and related work which motivates our design of a new memory management scheme in Section II. In Section III and Section IV, we present the detailed algorithm and the evaluation of our two new memory managers. Conclusion is presented in Section V.

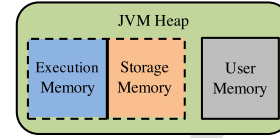


Fig. 1. Memory partition of spark memory managers.

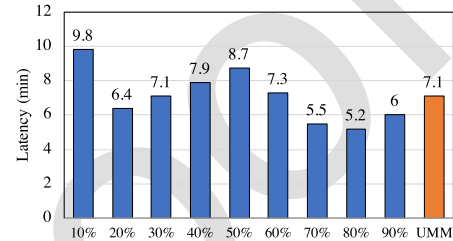


Fig. 2. Latency of application under SMM and UMM. SMM increases storage fraction from 10% to 90%.

II. MOTIVATION AND RELATED WORK

In this section, we study the performance of Spark applications managed by two existing Spark memory managers (i.e., SMM and UMM). In both memory managers, as shown in Fig. 1, a portion of Java heap (i.e., memory in the dashed rectangle) is dedicated for processing Spark applications (called *Accessible Memory*), while the rest of memory is reserved for Java class references and metadata usage (called *User Memory*). Accessible memory is further divided into two partitions, *Storage Memory* and *Execution Memory*. The boundary between the storage memory and execution memory is fixed (i.e., static) in SMM, but flexible in UMM. Storage memory is used for caching RDDs, while execution memory is used for runtime task processing. If storage memory is already fully utilized when a new RDD needs to be cached, some old RDDs will be evicted according to the LRU (Least Recently Used) algorithm. On the other hand, if execution memory is full, all intermediate objects generated at runtime will be serialized and spilled into the disk to release memory space for subsequent task processing.

A. SMM: Static Memory Partition Analysis

To understand how memory partition can affect Spark performance, we conduct a set of experiments in a Spark cluster consisting of four homogeneous workers (see the setup in Section IV-B), with PageRank [6] as a representative benchmark. We set the boundary, which we also refer to as *storage fraction* (i.e., the ratio of storage memory to accessible memory), from 10% to 90% of accessible memory space under SMM. Since the total accessible memory dedicated to Spark applications remains constant, execution memory is decreased when storage memory is increased.

Fig. 2 first illustrates the experiment results for SMM with different storage fractions. We can observe that the Spark performance varies with different memory partitions. Intuitively, if the storage memory is too small to cache RDDs that will be reused in the following computations, the RDD processing time cannot be

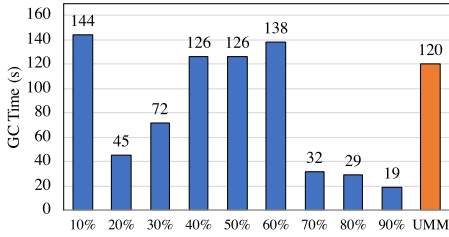


Fig. 3. GC time comparison. SMM increases storage fraction from 10% to 90%.

177 saved. On the other hand, if we assign too much space to storage
 178 memory, then the confined execution memory pool may trigger
 179 a high overhead of I/O communications. However, neither one
 180 of these two effects dominates the other, and the resulting joint
 181 performance depends on the characteristics of the workload. As
 182 shown in Fig. 2, the latency is not a monotonic function of the
 183 storage memory size. Therefore, we conclude that SSM yields
 184 varying performance with different storage fractions and cannot
 185 automatically achieve optimal performance.

186 B. Static VS. Dynamic: Latency Comparison

187 SMM cannot fit all kinds of workloads well because of
 188 its lack of flexibility. Compared with SSM, UMM allocates
 189 memory resources dynamically according to resource demands.
 190 Furthermore, UMM gives a higher priority to execution memory
 191 than to storage memory. Execution memory can force the storage
 192 memory pool to shrink if storage memory exceeds 50% of total
 193 accessible memory, even if it is fully utilized. Based on this
 194 mechanism, UMM guarantees sufficient memory for executing
 195 run-time tasks, which avoids the content of execution memory
 196 from being spilled into the disk to the greatest extent.

197 We find that UMM still cannot consistently achieve the best
 198 performance, although it strives to adjust the storage fraction
 199 based on resource demands dynamically. For example, the last
 200 bar in Fig. 2 further shows the latency of UMM. We can see
 201 that UMM does help improve the performance by obtaining
 202 lower latency than SSM with some storage fractions (e.g., 10%
 203 and 50%). Whereas UMM cannot beat SSM with a storage
 204 fraction of 20% and 70%~90%, and thus cannot achieve optimal
 205 performance.

206 C. UMM Limitation: GC Impact

207 To explore the cause of UMM’s ineffectiveness, we conduct a
 208 set of experiments to investigate the impact of garbage collection
 209 (GC) on Spark application latency. We plot the GC times of
 210 SMM with different storage fractions and that of UMM in Fig. 3.
 211 We observe that SMM has a much lower GC time when storage
 212 fraction is set to 20%, 30%, and $\geq 70\%$. In contrast, the GC
 213 time under UMM is as high as 120 seconds, about six times the
 214 lowest GC time obtained by SMM with a storage fraction of
 215 90%. By combining the results in Figs. 3 and 2, we note that the
 216 GC time has considerable impacts on Spark performance and
 217 UMM’s performance degradation results from such a long GC
 218 time.

We discover that long GCs occur under UMM because UMM
 expands the execution memory pool aggressively, resulting in
 a large amount of intermediate data in execution memory. The
 Java garbage collector then needs to maintain these in-memory
 intermediate data and thus increases the overall GC time. Such
 high GC time finally introduces extra latency to a Spark appli-
 cation’s execution. Besides, there exist no explicit methods to
 eliminate these long GCs by configuring UMM by users. This
 observation motivates us to consider both GC time and execution
 time for dynamically adjusting memory partition. The impact
 of GC on Spark’s performance is also investigated in existing
 works, which will be discussed in Section II-E.

231 D. Need for Learning-Based Solutions

232 The basic version of our new memory manager (ATuMm) is
 233 designed based on an intuitive feedback-control solution, which
 234 uses the current task’s execution as the feedback to decide the
 235 increase or decrease in the boundary between the execution and
 236 storage memory pools with a fixed adjustment step. To obtain
 237 an optimal learning speed, the user must manually configure
 238 the adjustment step, which requires pre-knowledge about the
 239 workload and the system characteristics. Even with an optimal
 240 adjustment step, our ATuMm may not consistently achieve
 241 the best performance. One reason is the fixed adjustment step
 242 that cannot work well for applications with varying memory
 243 demands. Another reason is that ATuMm makes the tuning
 244 decisions heavily depending on the execution status of the
 245 current task. Motivated by the above limitations, we need to
 246 design a more comprehensive learning solution that can have an
 247 intelligent agent to “smartly” calculate rewards for dynamically
 248 tuning the adjustment step and thus optimizing the learning
 249 speed. We select Q-learning algorithm as our intelligent memory
 250 management agent for the following reasons. First, Q-learning is
 251 model-free, meaning it doesn’t require a complete understanding
 252 of the underlying system dynamics. This makes it suitable
 253 for situations where the environment is complex, uncertain,
 254 or difficult to model accurately. Second, Q-learning employs
 255 temporal difference learning, allowing it to learn from each
 256 individual interaction with the environment. This characteristic
 257 makes it well-suited for online learning and environments where
 258 data arrives sequentially. Third, compared to other powerful but
 259 complicated ML/DL models, i.e., convolutional neural networks
 260 and transformers, Q-learning is light to integrate with existing
 261 systems and offers low learning overhead.

262 E. Gap in the Existing Works

263 We summarize existing works in Table I. MEMTUNE
 264 presents an algorithm that adjusts memory allocation based
 265 on the characterizations of tasks (i.e., storage-sensitive or
 266 execution-sensitive). This work considers the impact of JVM on
 267 Spark performance to decide how to balance memory allocation
 268 for obtaining a good performance. But, this work only focuses
 269 on analyzing the sensitivity of tasks and takes different actions,
 270 such as reserving more memory for storage requirements if tasks
 271 are storage-sensitive. Another work DSMM, dynamically sets
 272 the storage fraction by simply comparing the size of the data set

TABLE I
COMPARISON OF EXISTING SPARK MEMORY OPTIMIZATION WORKS

	Optimization Level	Workload Characterizing	Machine Learning	Garbage Collection
MEMETUNE [7]	Memory	Sensitivity Analysis	N/A	N/A
DSMM [8]	Memory	Data Size Analysis	N/A	N/A
SMBSP [9]	Framework	N/A	Artificial Neural Network	N/A
MLAT [10]	Framework	N/A	Regression & Clustering	N/A
PokéMem [11]	Memory	Data Size Analysis	N/A	Considered
MCS [12]	Memory	N/A	N/A	Considered
Q-ATuMm	Memory	Learning-based Analysis	Q-Learning	Considered

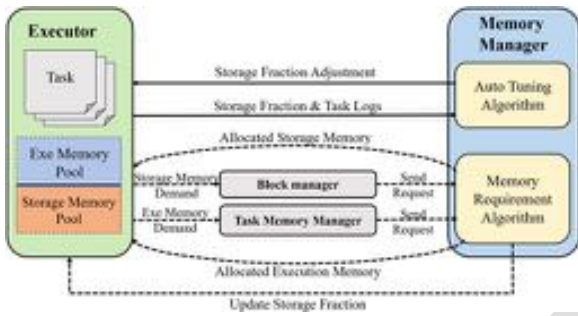


Fig. 4. New memory allocation scheme architecture.

with its memory usage. Compared to our work, these two works fail to track the memory requirement diversity at run-time, which still relies on preknowledge of the application’s characteristics.

SMBSP applies Artificial Neural Network (ANN) to configure Spark’s parameters automatically, including computation, cache, and storage configurations. MLAT is another work that utilizes machine learning to auto-config Spark’s parameters. This work learns proper configurations for different Spark clusters as well. However, these two works optimize Spark’s performance at a coarser level and lack consideration of runtime workload characteristic adjustment compared to our work. We also note that our work contributes to optimizing Spark’s caching logic and can be adapted easily to [9] and [10].

PokéMem and MCS consider the impact of GC on Spark’s performance and strive to optimize memory management via controlling GC. PokéMem focuses on reducing memory pressure by estimating the data size of objects created by third-party libraries. However, the estimation model is data structure- and library-dependent. MCS is close to our work which defines constraints to limit the priority of execution memory. However, it lacks dynamic adjustment of these constraints.

III. NEW LEARNING-BASED MEMORY MANAGER DESIGN

In this section, we present our new learning-based memory allocation scheme, which aims to improve the overall latency for Spark applications by considering both *resource demands* and *garbage collection impact* in dynamic memory resource allocation. Fig. 4 shows the overview of our design and illustrates the overall block diagram of Spark modules on an “Executor”. A Spark cluster often consists of multiple “Executors”. Each “Executor” hosts a set of running tasks and manages their storage and execution memory pools independently. In addition, there are two managers in Spark that are responsible for the memory

requests sent from the “Executor” module. Specifically, the “Block Manager” manages the storage memory requirements, and the “Task Memory Manager” manages the execution memory requirements.

In our memory allocation scheme, we develop two new main modules, called *Auto Tuning Algorithm* (i.e., ATuMm or Q-ATuMm), and *Memory Management Algorithm*, and integrate them with the existing Spark modules, as shown in Fig. 4. The “Executor” periodically calls the “Auto Tuning Algorithm” to adjust the storage fraction and set the limit (or the maximum allowed) of execution memory. The “Memory Management Algorithm” further responds to the memory requirements sent by the “Block Manager” and “Task Memory Manager” modules by considering both free storage/execution memory space and the decision made by the “Auto Tuning Algorithm”. Upon completing each task, the “Auto Tuning Algorithm” receives the runtime logs of the completed task and the previously completed tasks from the “Executor” module. Based on these logs, the algorithm adjusts (1) the boundary between the storage and execution memory pools and (2) the maximum allowed memory space to the execution pool. The adjustment decisions are then passed to the “Executor” for the next task execution. The above adjusting process repeatedly occurs until the last task at the “Executor” completes. Meanwhile, the “Memory Requirement Algorithm” bases on the memory requirements from the “Executor” to allocate the memory space for the RDD cache (i.e., storage memory) and task execution (i.e., execution memory). The storage fraction is then accordingly updated by this algorithm based on runtime memory demands.

A. Memory Requirement Algorithm

The *Memory Management Algorithm* is designed to allocate memory space for RDD caching and task execution. In particular, this algorithm receives the online memory requirements from the “Block Manager” and the “Task Memory Manager” modules. Specifically, our scheme maintains two parameters: “StorageFraction” and “heapStorageMemory”. While the former decides the maximum available memory of the storage memory pool, the latter limits the maximum available memory of the execution memory pool. According to the current storage partition and “heapStorageMemor”, this algorithm allocates available memory to the two manager modules (i.e., “Block Manager” and “Task Memory Manager”) to meet their requirements.

Algorithm 1 describes the main procedures of this memory management mechanism.

Algorithm 1: Memory Requirement Algorithm.

```

1 Procedure acquireExecutionMemory (reqExe)
2   extraNeed=reqExe-freeExecutionMemory
3   if extraNeed>0 then
4     memoryBorrow=min(extraNeeded,storageMemoryPoolSize-
5       heapStorageMemory,freeStorageMemory)
6     decreaseStoragePoolSize(memoryBorrow)
7     increaseExecutionPoolSize(memoryBorrow)
8     acquired = executionMemory-
9       Pool.acquire(freeExecution+memoryBorrow)
10  else
11    acquired=executionMemoryPool.acquire(reqExe)
12  return acquired
13
14 Procedure acquireStorageMemory (reqSto)
15   memoryToFree=max(0, reqSto-freeStorageMemory)
16   if memoryToFree>0 then
17     freeStorageMemory(memoryToFree)
18   acquired = storageMemoryPool.acquire(reqSto)
19   if heapStorageMemory<usedStorageMemory then
20     heapStorageMemory=usedStorageMemory
21   return acquired

```

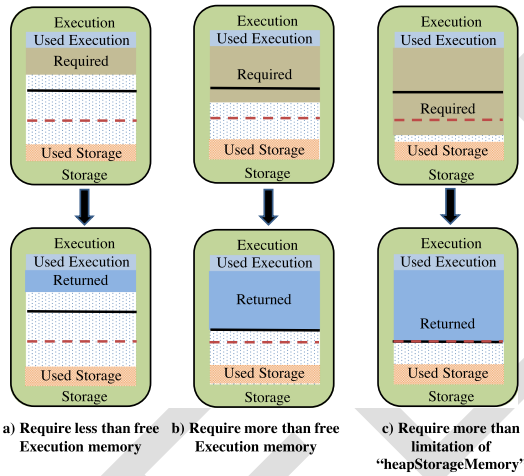


Fig. 5. Execution requirement conditions.

– Procedure *requireExecutionMemory()* takes “reqExe” as the input, which is the execution memory size required by “Task Memory Manager”, and returns the actual allocated execution memory. Specifically, execution memory requirements can be one of the three scenarios shown in Fig. 5. In the figure, we plot the Spark memory pool on an “Executor”, where a solid line represents the potential boundary between execution memory and storage memory. A dashed line represents the value of “heapStorageMemory”, indicating the least reserved space for storage memory. Besides, we also mark the used execution and storage memory space. In the first scenario, the required execution memory is less than the free execution memory, see Fig. 5(a). Then, the procedure allocates all needed memory to “Task Memory Manager”.

The second scenario is shown in Fig. 5(b), where the required execution memory exceeds the free execution memory but not beyond the limit of “heapStorageMemory”. Procedure *requireExecutionMemory()* still allocates all needed memory to “Task Memory Manager” and meanwhile expands the execution memory pool by moving down the boundary bar (see the solid

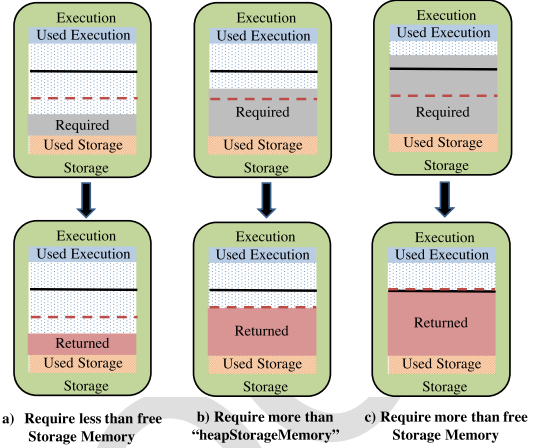


Fig. 6. Storage requirement conditions.

line in the bottom plot of Fig. 5(b)). Finally, suppose the required execution memory exceeds the boundary of “heapStorageMemory”. In that case, the procedure only allocates the memory up to “heapStorageMemory” (see the dashed line in the bottom plot of Fig. 5(c)) and also moves down the boundary bar to “heapStorageMemory”. Our algorithm prevents memory over-allocation for task execution by limiting the memory that can be allocated to execution memory. For example, in both scenarios (b) and (c), the execution memory pool occupies part of storage memory after allocating memory to the execution memory pool. However, in scenario (c), we use “heapStorageMemory” to avoid the execution memory pool invading the storage memory pool. In this way, GC time can be reduced as discussed in Section II.

– Procedure *requireStorageMemory()* receives the required storage memory size (“reqSto”) from the “Block Manager” module for allocating actual memory to cache RRDs. Similarly, we have three possible conditions of storage memory requirements, depicted in Fig. 6. If the required storage memory is less than free storage memory as shown in Fig. 6(a) and (b), then all required memory will be allocated to “Block Manager” (no matter beyond “heapStorageMemory” or not). In contrast, if the required storage memory is more than the free storage memory (see Fig. 6(c)), then only the memory space up to the boundary bar will be allocated to “Block Manager,” and meanwhile, RDD eviction will be triggered to release some memory for caching new RDDs. In both scenarios 2 and 3, we further update the variable “heapStorageMemory” to be equal to the actual storage memory pool size.

It is noticeable that “Memory Management Algorithm” does change the storage fraction under some scenarios, such as the ones shown in Fig. 5(b) and (c). Thus, the storage fraction is jointly determined by both “Memory Management Algorithm” and “Auto Tuning Algorithm”.

B. Auto Tuning Algorithm

Here, we first present the basic version of our auto-tuning algorithm, named ATuMm, which uses a feedback-control way to dynamically adjust the boundary of two memory pools with

Algorithm 2: ATuMm.

```

1 Procedure barChange ( GCTime, executionTime)
2   curRatio=GCtime/executionTime
3   if curRatio=preRatio then
4     return None
5   else if (curRatio<preRatio and preUpOrDown=true) or
6         (curRatio>preRatio and preUpOrDown=false) then
7     update preUpOrDown to ture, update preRatio
8     return (setUp(step))
9   else
10    update preUpOrDown to false, update preRatio
11    return (setDown(step))
12
13 Procedure setUp (step, preStorageFraction)
14   if preStorageFraction+step<100% then
15     curStorageFraction=preStorageFraction+step
16     if usedStoragePoolSize/totalStoragePoolSize>80% then
17       heapStorageMemory=heapStorageMemory+
18       step*accessibleMemory
19     update preStorageFraction
20     return heapStorageMemory, curStorageFraction
21
22 Procedure setDown (step, preStorageFraction)
23   if preStorageFraction-step>0 then
24     curStorageFraction=preStorageFraction-step
25     memoryEvict=memoryUsed-curStorageFraction
26     if memoryEvict>0 then
27       freeStorageMemory(memoryEvict)
28     heapStorageMemory=heapStorageMemory-
29     step*accessibleMemory
30     if heapStorageMemory>=curStorageFraction*accessibleMemory
31       then
32         heapStorageMemory=curStorageFraction*accessibleMemory
33     update preStorageFraction
34   return heapStorageMemory, curStorageFraction

```

407 a fixed adjustment step. Then, we propose a Q-learning-based
408 algorithm, named Q-ATuMm, which uses an intelligent agent
409 to optimize the learning speed by automatically tuning the
410 adjustment step.

411 1) *Basic Version. ATuMm:* When a task on the “Executor”
412 completes, the “Auto Tuning Algorithm” takes the GC time, the
413 execution time of the completed task, and the current storage
414 fraction as inputs and then compares the performance of the
415 completed task (in terms of the ratio of GC time to execution
416 time) with that of the previous tasks to make the adjustment
417 decision. In particular, the “Auto Tuning Algorithm” returns two
418 variables: (1) a new storage fraction (“curStorageFraction”) for
419 the potential memory partition, and (2) a new “heapStorage-
420 Memory” variable to indicate the least memory reserved for
421 storage memory. Using these two variables, ATuMm can adjust
422 the memory partition with a limit on the maximum memory that
423 can be allocated to execution memory. Algorithm 2 shows the
424 pseudo-code of the “Auto Tuning Algorithm”.

425 Both *setUp()* and *setDown()* repartition the accessible mem-
426 ory to the storage and execution pools based on the decision
427 made by *barChange()*. We also remark that the variable “heap-
428 StorageMemory” is new in our design, which plays a critical
429 role in avoiding long GC time resulting from over-allocated
430 execution memory. Later, we present how this variable is used
431 in the “Memory Requirement Algorithm” to control the actual
432 memory space for RRD caching and task execution.

433 – Procedure *barChange()* receives GC time and execution
434 time of the current task from the “Executor” module. We con-
435 sider the ratio of GC time to execution time as a measurement of

Spark performance. A low ratio indicates a “good performance”,
436 vice verse. Then, *barChange()* makes an adjustment decision
437 from one of three possible actions (i.e., keep still, increase
438 storage fraction, and decrease storage fraction). In particular, we
439 use two variables, “preRatio” and “preUpOrDown” to record the
440 ratio of GC time to the execution time of previous tasks and the
441 last adjustment decision, respectively. We compare “curRatio”
442 with “preRatio” to calculate the reward of the last adjustment.
443 If the current task yields a better performance (i.e., “curRatio” is
444 lower than “preRatio”), the boundary-moving decision that we
445 previously made (i.e., “preUpOrDown”) gets a reward. Thus, we
446 decide to keep moving the boundary further in the same direction
447 as the last task. Otherwise, we move the boundary in a direction
448 that is opposite to that of the last adjustment. Besides these two
449 actions, if the Spark performance converges (i.e., the current
450 ratio is equal to the previous ratio), the boundary keeps still.
451 After taking the new action, the storage fraction changes, and
452 two variables (i.e., “preRatio” “preUpOrDown”) are updated for
453 the next decision.

– Procedures *setUp()* and *setDown()* control how to expand
455 or shrink the storage and execution memory pools base on the
456 decision made in *barChange()*. As mentioned in Section II,
457 Spark memory is divided into two pools, i.e., storage memory
458 and execution memory. We thus consider there exists a parti-
459 tion “bar” between storage and execution memory in Spark.
460 Setting the bar up means enlarging the storage memory pool
461 and shrinking the execution memory pool, while setting the bar
462 down means decreasing the storage memory pool and expanding
463 the execution memory pool. In ATuMm, users can configure the
464 percentage of accessible memory (indicated as “step”) that will
465 be increased or decreased in each adjustment.
466

It is challenging to move the partition bar if both storage
467 and execution memory pools are fully utilized. A mechanism
468 is required to determine which objects should be evicted. LRU
469 (Least Recently Used), an existing RDD caching algorithm, is
470 applied by the Spark block manager for storage memory. We
471 adopt this caching algorithm to manage the RDD evictions from
472 storage memory. For execution memory, *barChange()* is called
473 only when a task has finished its computation and released all
474 its occupied memory resources. Thus, there is no need to evict
475 objects from the execution memory pool. This is also one reason
476 we choose to adjust the memory boundary after each task’s
477 completion.
478

Procedure *setUp()* takes “preStorageFraction” and the pre-
479 defined parameter “step” (e.g., 5%) as inputs to determine a
480 new storage fraction (“curStorageFraction”) to repartition the
481 memory and a bound (“heapStorageMemory”) to reserve the
482 least storage memory space. In detail, *setUp()* increases the
483 storage fraction by “step” (see lines 12 and 13 in Algorithm 2)
484 if the new storage memory pool size is less than the overall
485 available memory space. Meanwhile, *setUp()* updates “heap-
486 StorageMemory” only if 80% of the storage memory is used
487 (see lines 14, and 15 in Algorithm 2). The difference between
488 the storage memory pool size and “heapStorageMemory” will
489 be the potential memory space allocated to execution memory.
490

Procedure *setDown()* has the same inputs and outputs
491 as *setUp()* to shrink the storage memory pool. In details,
492

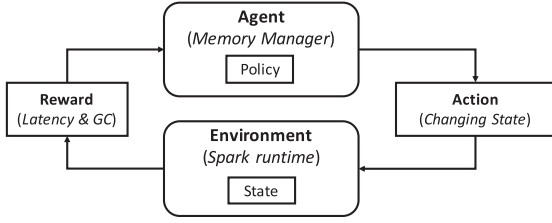


Fig. 7. Reinforcement Learning (Q-Learning) Algorithm in Q-ATuMm. We define 1) agent represents the memory manager, 2) environment is Spark runtime, 3) state represents “StorageFraction” and “heapStorageMemory” that limit the allocation of storage and execution memory, 4) action is changing state, and 5) reward is calculated from latency and GC time.

493 *setDown()* decreases the storage fraction by “step” (see line 20 in
 494 Algorithm 2). However, it needs to consider RDD evictions to
 495 release the reduced storage memory additionally (see lines 21
 496 and 22 in Algorithm 2). For example, if the current storage
 497 memory pool is 5 GB with 4.5 GB used, and the potential
 498 storage memory becomes 4 GB, then the memory space (‘mem-
 499 oryEvict’) that needs to be released is 0.5 GB. *setDown()* then
 500 needs to trigger the caching algorithm to evict cached RDDs
 501 to shrink the storage memory pool. Finally, *setDown()* updates
 502 (or decreases) “heapStorageMemory” by “step” of accessible
 503 memory. If “heapStorageMemory” is more than the new storage
 504 memory, then *setDown()* sets ‘heapStorageMemory’ to be equal
 505 to the new storage memory (see line 25 in Algorithm 2).

506 2) *Q-Learning Based Version: Q-ATuMm*: As discussed in
 507 Section II-D, ATuMm suffers from the inflexibility of the adjust-
 508 ment step. In order to optimize the adjustment speed, we further
 509 refine our auto tuning algorithm by using reinforcement learning
 510 techniques to automatically set the adjustment step for changing
 511 the memory boundaries. On the other hand, Spark applications
 512 process data in batches, possessing consistent memory and com-
 513 putation characteristics, which can be learned by reinforcement
 514 learning efficiently. Q-learning is a specific algorithm within the
 515 broader field of reinforcement learning, which receives feedback
 516 from the objective and makes decisions to optimize the rewards.
 517 As shown in Fig. 7, an *agent* interacts with an *environment* by
 518 taking actions, then the environment returns a reward of the
 519 action to the agent and updates the state of the environment. By
 520 exploiting different actions across all possible states, the agent
 521 can produce an optimal policy to manipulate the states of the
 522 environment.

523 Q-learning maintains a Q-table, where the columns and rows
 524 represent states and actions. The values (i.e., value function) in
 525 the Q-table represent the expectation of benefits of applying an
 526 action, given a state. The agent updates the value function based
 527 on an equation (particularly Bellman equation [13]). Specifi-
 528 cally, Q-learning maintains an exploration-exploitation balance,
 529 ensuring that the agent explores new actions and state-action
 530 pairs while exploiting learned information to make optimal
 531 decisions. Theoretically, an epsilon-greedy exploration strategy,
 532 as used in the Bellman equation, guarantees that all state-action
 533 pairs are visited infinitely often, which is crucial for conver-
 534 gence. Another important factor in Q-learning is the convergence
 535 rate. The convergence rate of Q-learning depends on factors

Algorithm 3: Q-ATuMm

```

1 Procedure initializeAgent ()
2   Initialize stateSpace
3   Initialize actionSpace
4   Initialize QTable
5   Initialize  $\alpha, \epsilon, \gamma$ 
6   Initialize stateIndex, actionIndex
7 Procedure QLearningAgent (GCTime, executionTime, stateIndex,
  actionIndex)
8   reward = taskTime / (GCTime +  $\delta$ )
9   QTable(stateIndex, actionIndex) = updateQTable (reward,
  stateIndex, actionIndex)
10  rnd = random(0, 1.0)
11  if rnd <  $\epsilon$  then
12    actionIndex = random(0, actionSpace.length)
13  else
14    actionIndex = GetIndex(QTable(stateIndex).max)
15  action = actionSpace(actionIndex)
16  state = stateSpace(stateIndex)
17  return action
18 Procedure updateQTable (reward, stateIndex, actionIndex)
19  QValue = QTable(stateIndex, actionIndex)
20  stateValue =  $\gamma$  * (QTable(stateIndex).max - QValue)
21  QValue = QValue +  $\alpha$  * (reward + stateValue)
22  return QValue
  
```

such as the learning rate schedule and the characteristics of the
 environment. In practice, while Q-learning converges asymptoti-
 cally, convergence speed can vary, and certain modifications, like
 learning rate annealing, can influence the convergence rate. We
 evaluate the impact of learning rate and other hyper-parameters
 in Section IV-C4.

In Q-ATuMm, when the “Executor” finishes a task, the agent
 (i.e., memory manager) calculates the reward of the last action
 based on the execution time and GC time of the current task.
 Then Q-ATuMm updates the policy and makes a decision about
 which is the next state. Specifically, *InitializeAgent()* initial-
 izes all parameters before running applications. *QLearningAgent()*
 uses the garbage collection time and execution time of the
 completed task to calculate the reward of the current action and
 calls *UpdateQTable()* to update values of the current state and
 action in Q-table. *QLearningAgent()* then decides the action to
 execute the following task by either exploring a new action or
 exploiting a known action. We note that Q-ATuMm creates a
 two-dimension discrete action space, where each element in
 the action space represents a pair of “StorageFraction” and
 “heapStorage-Memory”, as introduced in Section III-A. We
 define “StorageFraction” and “heapStorage-Memory” as ratios
 of the overall heap size, ranging from 1% to 99%. The status
 space is the same as the action space. Algorithm 3 describes the
 details of Q-ATuMm. Q-ATuMm trains the model on-the-fly.

– Procedure *initializeAgent()* initializes the state space,
 the action space and the Q-table. We denote α as the learning rate,
 representing the length of the step to update the value function.
 ϵ is the exploration ratio, which indicates how much the agent
 prefers to explore unknown actions. We denote γ as a discount
 factor reflecting how much the future rewards contribute to the
 current update.

– Procedure *QLearningAgent()* receives the garbage collec-
 tion and execution time of the task, with the state of current
 “stateIndex” and “stateAction”, which locate the value function
 in the Q-table to update. Because our goal is to minimize garbage

536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571

TABLE II
TESTBED CONFIGURATION

Component	Specs
Host Server	Dell PowerEdge T310
Host Processor Speed	2.93GHz
Host Memory Capacity	16GB DIMM DDR3
Host Memory Data Rate	1333 MHz
Host Storage Device	Western Digital WD20EURS
Host Disk Bandwidth	SATA 3.0Gbps
Host Hypervisor	VMware Workstation 12.5.0
Processor Core Per Node	1 Core
Memory Size Per Node	1 GB
Disk Size Per Node	50 GB

572 collection and reduce the overall latency, $QLearningAgent()$ de-
573 fines the reward as the ratio of the execution time (GC time plus
574 others) to the GC time plus a constant number (i.e., $\delta = 0.01$)
575 to avoid zero denominators (see line 8 in Algorithm 3). $Update-$
576 $QTable()$ is then called to update the value function in the Q-
577 table. $QLearningAgent()$ uses a parameter ϵ to decide to explore
578 a random action or to exploit the action with the largest benefit
579 (see lines 11-14 in Algorithm 3). A larger ϵ means the agent
580 prefers to explore unknown actions. Finally, $QLearningAgent()$
581 returns the action to the “Executor” to execute the following
582 tasks.

583 – Procedure $updateQTable()$ takes the reward as an input
584 to calculate the new value in Q-table based on the Bellman
585 equation [13]. First, $UpdateQTable()$ locates the value in Q-table
586 and then computes the “stateValue” to estimate the reward of the
587 next state. It is worth pointing out that the parameter γ is used
588 to decide how important future decisions are. A larger γ indicates
589 the agent relies more on the future reward than the current one.
590 Finally, $UpdateQTable()$ updates the “Q value” with the current
591 reward and the estimated future reward. The parameter α is used
592 as the learning rate to control how fast the agent learns from
593 the rewards. There is a trade-off between learning speed and
594 accuracy. A larger learning rate can allow the agent to learn and
595 move faster to the optimal solution, but meanwhile, has a higher
596 possibility of causing the agent to be trapped in a locally optimal
597 point.

598 IV. EVALUATION

599 In this section, we discuss the implementation and the evaluation
600 of ATuMm and Q-ATuMm in a real Spark cluster. We aim to
601 investigate the performance in terms of latency, memory usage,
602 and garbage collection at run-time. We use default UMM and
603 SMM mode as our baseline, which is discussed in Section II.

604 A. Testbed

605 We conduct our experiments in a Spark cluster with one driver
606 and four workers that are homogeneous to each other. The cluster
607 is deployed on the Dell PowerEdge T310 and hypervised by
608 VMware Workstation 12.5.0. Each node in the Spark cluster is
609 assigned 1 CPU, 1 GB memory, and 50 GB disk space. Table II
610 summarizes the details of our testbed configuration.

611 We implement ATuMm and Q-ATuMm as new portable mem-
612 ory manager modules, besides SMM and UMM, in Apache

Spark 2.4.0, which contain functions interacting with other 613
Spark modules. It is noticeable that our new memory man- 614
ager can also be integrated into Spark from the version of 615
1.6.0 to 2.4.0. The source code is available on GitHub.¹ The 616
LOC is 2,428 in total. Specifically, we develop functions $ac-$ 617
 $quireStorageMemory()$ and $acquireExecutionMemory()$ to allo- 618
cate storage and execution memory to “Block Manager” and 619
“Task Memory Manager”, respectively. We also integrate a 620
profile collector in the “Executor” module to collect task logs. 621
Specifically, ATuMm applies function $barChange()$ to receive 622
these task logs and calls functions $increaseStorageFraction()$ or 623
 $decreaseStorageFraction()$ to adjust memory partition. Mean- 624
while, Q-ATuMm uses function $updateQTable()$ to maintain 625
the Q-Table for the agent to perform reinforcement learning. 626
Furthermore, we integrate a memory usage analyzer in ATuMm 627
and Q-ATuMm to collect the run-time memory usage informa- 628
tion. Users can replace the existing Spark memory manager to 629
ATuMm or Q-ATuMm by simply setting a configurable param- 630
eter before submitting a Spark application. 631

632 B. ATuMm Evaluation

633 We set the accessible memory and the initial storage fraction 634
of ATuMm as the same as those of UMM (i.e., accessible 635
memory is 60% of JVM heap, and storage memory is initial- 636
ized as 50% of accessible memory). The step to increase or 637
decrease storage fraction in each adjustment is configured as 638
5% of accessible memory by default. Furthermore, the win- 639
dow size representing the number of previous tasks is set as 640
20% of activated tasks by default. Users can pre-configure 641
these parameters in ATuMm before launching any Spark 642
applications.

643 1) *Latency Analysis*: We evaluate and compare the perfor- 644
mance of Spark applications under three memory managers 645
(SMM, UMM, and ATuMm) by conducting experiments with 646
different applications. We choose PageRank and K-means as 647
benchmarks because these two applications are two ubiquitous 648
techniques, which are widely applied in machine learning and 649
data mining applications [6], [14]. Considering the duration of 650
experiments, we report results for a workload of 1 GB input data 651
for applications.

652 Fig. 8(a) and (b) illustrate the latency of PageRank and K- 653
means under different memory managers. We set various storage 654
fraction under SMM manually, and compare the latency of SMM 655
with that of UMM and ATuMm. In Fig. 8(a), we observe that 656
the performance of UMM beats SMM with some storage fractions 657
(e.g., 40% to 60%). However, when SMM sets the storage 658
fraction to 80%, it reaches the best performance, which achieves 659
27% shorter latency compared to UMM. More importantly, the 660
latency of our ATuMm is close to the lowest among all, and our 661
ATuMm beats UMM as well. Moreover, as shown in Fig. 8(b), 662
our ATuMm can achieve the best performance (i.e., the lowest 663
latency), compared with both UMM and SMM. We conclude that 664
ATuMm outperforms the other two existing memory managers 665
with the same computation resources allocated.

¹https://github.com/DanlinJia/spark_core_ATMM

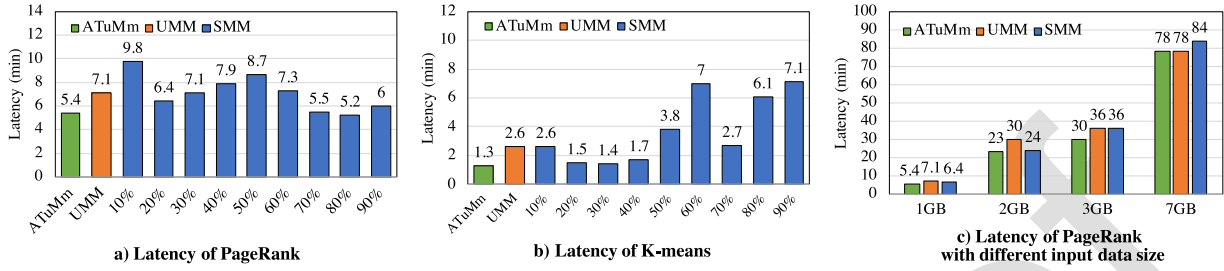


Fig. 8. Execution time of applications under SMM, UMM and ATuMm.

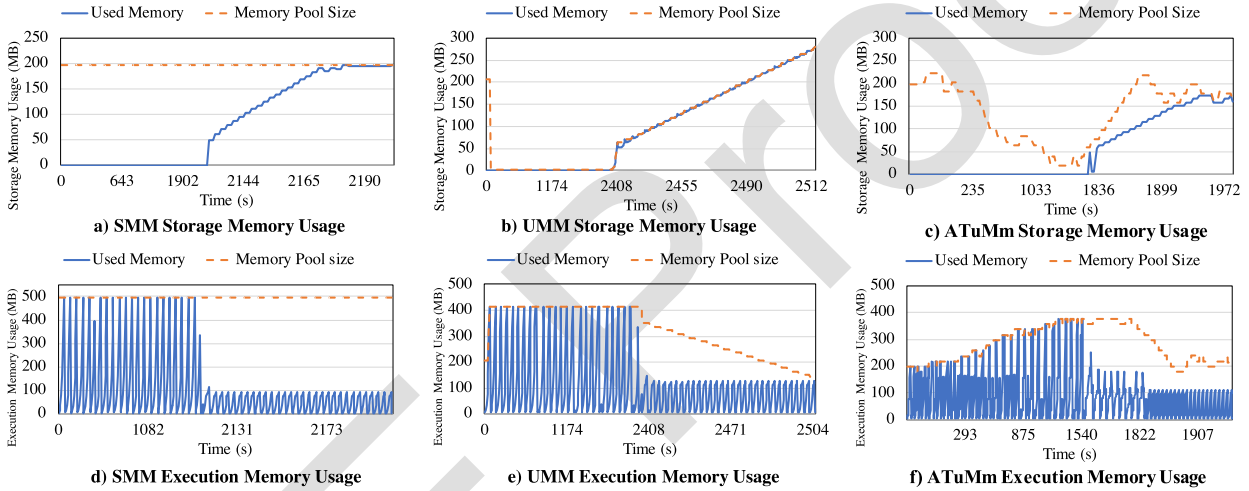


Fig. 9. Memory usage analysis of SMM, UMM and ATuMm.

2) *Sensitivity Analysis*: We also conduct a set of experiments to investigate the sensitivity of input data size, where we compare the performance of PageRank under three memory managers in the default mode with different input data sizes, such as 1 GB, 2 GB, 3 GB and 7 GB. As shown in Fig. 8(c), ATuMm achieves the best performance when the input data sizes are 1 GB, 2 GB, and 3 GB. Compared to UMM, ATuMm improves the latency by 25%. We interpret this improvement by observing that ATuMm leverages the GC time to repeatedly adjust the boundary between storage and execution memory, which prevents the Spark applications from a long GC duration as UMM introduced. When input data grows up to 7 GB, the overwhelming workload takes full usage of execution memory to process input data. Both UMM and ATuMm expand the execution memory pool aggressively to satisfy the massive execution memory requirements. As a result, UMM and ATuMm obtain similar performance (e.g., 78 minutes for 7 GB input data), which is better than that of SMM.

3) *Memory Usage and Garbage Collection Analysis*: We further look closely at the execution details of three Spark memory managers by plotting their memory usages in Fig. 9, where PageRank is running with 3 GB input data. Fig. 9(a)~(c) present the storage memory usage across time under the three memory managers, while Fig. 9(d)~(f) depict the corresponding execution memory usage. In each plot, the dashed line is the maximum memory size accessible for the corresponding

memory (such as storage or execution), and the solid line is the actual usage of the memory pool.

From Fig. 9(a)~(c), we observe that the storage memory utilization is similar for all three memory managers, which increases up to the maximum allowed storage pool size as time goes by. This is because RDDs are cached periodically in PageRank. Whereas, the storage memory pool sizes are different under three memory managers at different times. That is, both UMM and ATuMm dynamically change the storage memory pool sizes instead of the fixed one as SMM does. As shown in Fig. 9(a), the static storage memory pool starts to evict RDDs when the utilization of the storage memory pool is full. However, in Fig. 9(b), UMM drops the size of its storage memory pool to almost zero and then increase its storage pool when RDDs are cached. The storage memory pool changes more dynamically under ATuMm, as shown in Fig. 9(c). ATuMm first drops the storage fraction gradually as the execution memory pool expands, and then increases it as RDDs are cached. It is noticeable that ATuMm not only increases the storage memory pool based on storage memory requirements to cache RDDs, but also adjusts the pool size more rapidly than UMM to limit the execution memory pool size.

We further show our analysis of the execution memory usage under three memory managers in Fig. 9(d)~(f). SMM fixes the execution memory pool size regardless of workload diversity,

666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690

691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715

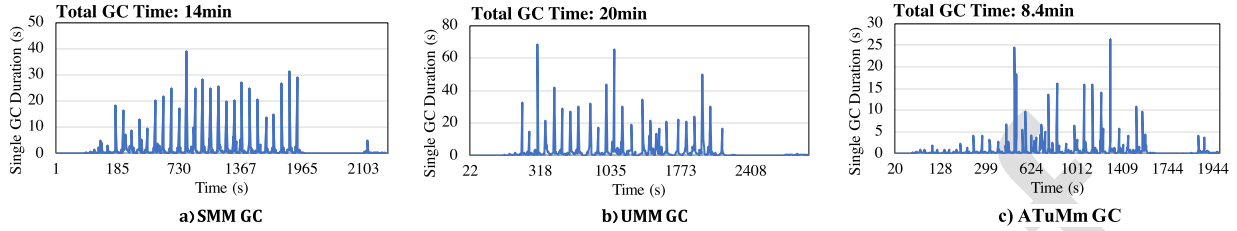


Fig. 10. GC analysis of SMM, UMM and ATuMm.

716 while UMM and ATuMm alter the execution memory pool size
 717 based on demands. Fig. 9(e) shows that the execution mem-
 718 ory pool of UMM expands aggressively and occupies almost
 719 all accessible memory when the first execution requirement
 720 comes. Contrarily, in Fig. 9(f), ATuMm increases gradually
 721 across time until it satisfies all execution requirements. This
 722 is because UMM expands the execution memory pool only
 723 based on execution memory requirements, while ATuMm fur-
 724 ther considers the impact of GC on Spark performance to con-
 725 trol the expansion of the execution memory pool. In addition,
 726 as the execution memory usage drops, UMM still gives the
 727 execution memory pool as much memory space as possible
 728 (i.e., all memory except that for caching RDDs). Conversely,
 729 ATuMm decreases the execution memory pool size more rapidly
 730 to limit the memory allocated to the execution memory pool. By
 731 this way, ATuMm can effectively prevent Spark applications
 732 from long GC durations introduced by overloaded execution
 733 memory. We can observe that the execution memory pool size
 734 converges to around 200 MB, which guarantees enough mem-
 735 ory for task execution and further offers a relatively low GC
 736 time.

737 We next present our observation regarding GC time. To show
 738 our observations, we use the PageRank application with 3 GB
 739 input data as representative and compare GC time using three
 740 memory managers. Fig. 10 shows the duration of garbage col-
 741 lection during the runtime of the application, where each spike
 742 represents an occurrence of a full GC (i.e., JVM stops all tasks
 743 and scans the whole heap to remove unreferred objects) that
 744 majorly contributes to GC time [15]. Fig. 10(a) shows that the
 745 maximum full GC time of SMM is around 40 seconds. While,
 746 under UMM, a full GC can take more than 70 seconds, see
 747 Fig. 10(b). More importantly, we can observe that the full GCs
 748 under ATuMm are all below 30 seconds in Fig. 10(c), which
 749 is smaller than both SMM and UMM. Besides, We observe
 750 that fewer spikes occurred under ATuMm than under UMM
 751 and SMM, which means that the frequency of full GCs under
 752 ATuMm is also lower than SMM and UMM. We also record the
 753 total GC time of SMM, UMM, and ATuMm, which is 14~min,
 754 20~min and 8.4~min, respectively. Since we use 4 executors in
 755 the experiment, the GC time of each executor should be divided
 756 by 4, which is considered as the contribution of GC to the overall
 757 execution time. Thus, we can conclude that ATuMm is able to
 758 significantly reduce the maximum and the total time of GCs
 759 when compared to SMM and UMM and thus accelerates the
 760 execution of Spark applications with minimum makespan (i.e.,
 761 total execution length).

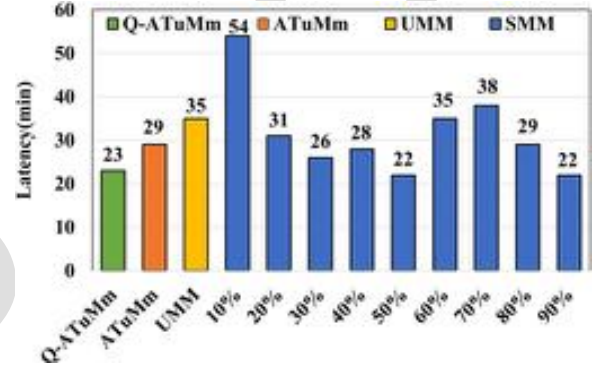


Fig. 11. Latency of PageRank under SMM, UMM, ATuMm, and Q-ATuMm.

C. Q-ATuMm Evaluation

762 We further implement and evaluate our Q-learning based
 763 version Q-ATuMm. We construct experiments on different
 764 categories of workloads (i.e., data-intensive applications and
 765 business queries) to evaluate the performance of Q-ATuMm,
 766 compared with that of SMM, UMM, and ATuMm. We tune
 767 the three hyper-parameters (i.e., learning rate, exploration
 768 ratio, and discount factor as shown in Section III-B2) in
 769 Q-ATuMm to achieve the best performance. The discus-
 770 sion on these hyper-parameters will be shown later in this
 771 section.

772 1) *PageRank Analysis:* We first construct the same exper-
 773 iments with PageRank on Q-ATuMm as shown in Section IV-B1.
 774 In order to trigger intensive data loading and processing, we
 775 increase the input data size to 5 GB. We observed that the
 776 application has fewer iterations to execute when the input size is
 777 small. Therefore, the Q-learning algorithm has fewer samples to
 778 learn. The performance of Q-ATuMm is worse with small data
 779 size. We also fix the number of iterations in PageRank as 20 in
 780 all experiments.

781 Fig. 11 illustrates the latency of PageRank under the four
 782 different memory managers. We manually set SMM storage
 783 fractions from 0.1 to 0.9 to observe the optimal latency exper-
 784 imentally. We observe that the best performance under SMM is
 785 achieved when the storage fraction is 50% and 90%, while UMM
 786 cannot reach that, which is consistent with our observations
 787 in Section IV-B1. On the other hand, we observe that both
 788 ATuMm and Q-ATuMm outperform UMM. More importantly,
 789 Q-ATuMm further reduces the latency by 28% compared to
 790 ATuMm.
 791

TABLE III
QUERY CLASSIFICATION

No.	Resource Intensity	Queries
1	I/O Intensive	Q1, Q3, Q4, Q10, Q21
2	CPU Intensive	Q1, Q3, Q6, Q12, Q13, Q21

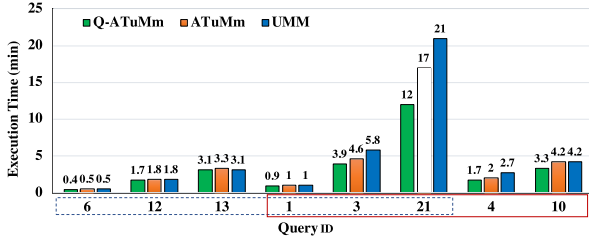


Fig. 12. Latency of TPC-H queries. Queries within the blue dashed box are CPU intensive. Queries within the red solid box are I/O intensive.

2) *Workload Intensity Analysis*: Q-ATuMm is further evaluated on a decision support benchmark named TPC-H [16] in the context of Apache Spark. TPC-H consists of twenty-two business-oriented queries and concurrent data modifications. TPC-H evaluates the performance of decision support systems by executing ad-hoc queries on a generated synthetic data set. In our experiment, we select representative queries running on a 10 GB data set. Work [17] investigates characteristics of TPC-H queries and classifies them based on resource intensity. We select two types of queries in TPC-H to evaluate Q-ATuMm, as shown in Table III. CPU Intensive queries contain operations like *order* and *select*, while I/O intensive queries either need to load large data set into memory or perform operations on multiple data sets, e.g., *join*. It is worth noticing that some queries can be both CPU and I/O intensive (e.g., Q1, Q3, and Q21).

We compare the performance of selected queries under Q-ATuMm with that under ATuMm and UMM. The first six queries in Fig. 12 illustrates the latency of CPU intensive queries with different memory managers. We observe that the latency of Q1, Q6, Q12, and Q13 does not have a visible variance among three memory managers, while Q-ATuMm outperforms the other two in Q3 and Q21. Our experimental results indicate that CPU-intensive queries hardly benefit from both ATuMm and Q-ATuMm, as their performance heavily relies on CPU resources. The last five queries in Fig. 12 are I/O intensive queries that need to load data into memory and trigger more RDD caching, which can significantly benefit from our new design. Thus, we observe a decent latency reduction above 20% in Q-ATuMm, compared with that in UMM. For Q1, we find that although Q1 needs to join two tables, each table is small. Therefore, even though Q1 is also classified as an I/O extensive query, its execution time is not reduced significantly by Q-ATuMm.

3) *Memory Usage and Garbage Collection Analysis*: To closely analyze the performance improvement under Q-ATuMm, we further collect the aggregated GC time of all executors under ATuMm, Q-ATuMm, UMM, and SMM with 0.9 storage fraction and show both total execution time (i.e., latency) and GC time for PageRank in Table IV. We first notice that GC time plays a dominant role in the total execution time.

TABLE IV
EXECUTION TIME AND GC TIME COMPARISON

Manager	Execution Time (min)	GC Time (min)
Q-ATuMm	23	21.48
ATuMm	29	26
UMM	35	31.72
SMM 0.9	22	20.24

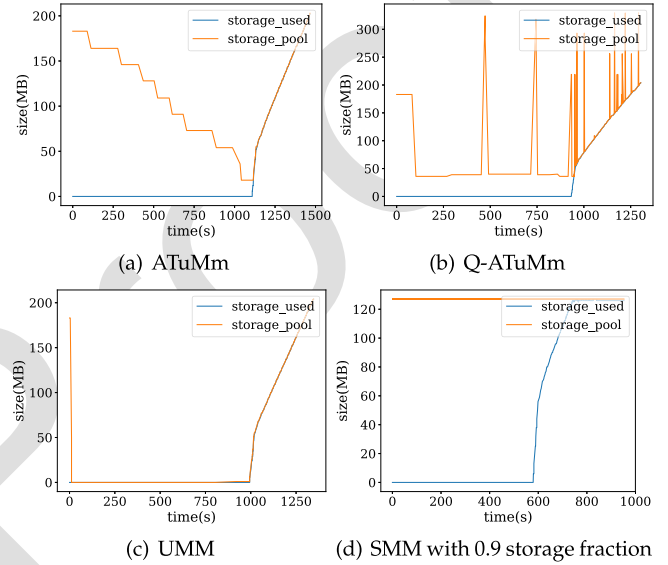


Fig. 13. Storage memory usage among all four memory managers.

By gradually reducing the storage fraction when the execution memory pool expands, our memory managers (i.e., ATuMm and Q-ATuMm) can significantly reduce the GC time by 17% and 32%, compared to UMM. Q-ATuMm further reduces the GC time (close to the optimal one as shown in the row of SMM 0.9 in Table IV) by using the Q-learning reinforcement technique to set the adjustment step for changing the memory boundaries automatically.

We further show storage memory usage among all four memory managers in Fig. 13. First, SMM has a fixed storage pool size (e.g., 0.9 storage fraction), and its storage memory usage increases up to the maximum allowed storage pool size as time goes by, which is caused by caching RDDs in each iteration. On the other hand, UMM, ATuMm, and Q-ATuMm dynamically change the storage memory pool size as time progresses based on the run-time memory resource demands. For example, as shown in Fig. 13, all of them start to increase the storage pool size at around 1000 seconds when RDDs start being cached.

However, we can observe that UMM immediately decreases the storage memory pool size to around zero to give more space to the execution memory pool, which unfortunately can cause a long GC time, as we discussed in Section II-C. To address this issue, ATuMm decreases the storage memory pool size gradually until it converges with the storage memory used size. It is visible that ATuMm gradually adjusts storage memory size based on the caching of RDDs, but it is less aggressive than UMM. For Q-ATuMm, we observe that the randomness that comes from exploration causes the spikes as the storage

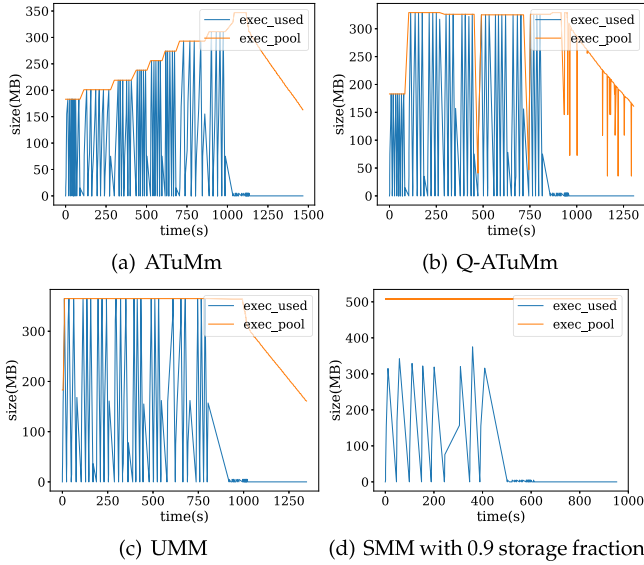


Fig. 14. Execution memory usage among all four memory managers.

memory pool size is dynamically adjusted. We also notice that the memory storage pool size decreases to below 50 almost from the starting point and stays there for about 900 seconds before the demand for storage memory increases because of RDD caching. In conclusion, we see that Q-ATuMm converges faster than ATuMm but less aggressive than UMM.

We also show execution memory usage among all four memory managers in Fig. 14. SMM's execution memory pool size remains fixed even though the actual execution memory usage is always lower than the allocated one, which indicates that SMM cannot fully utilize the execution memory, and meanwhile, it avoids triggering larger GC time. Based on the workload demands, UMM, ATuMm, and Q-ATuMm dynamically alter the execution memory pool size, which again proves to be more beneficial for execution memory utilization. ATuMm gradually increases execution storage as time passes, which helps reduce the long GC time. Q-ATuMm's execution memory pool size, on the other hand, is adjusted considerably to execution memory usage and converges at around 150 seconds, which is faster than ATuMm. The observation shows that our design of Q-ATuMm can converge fast to the run-time execution memory demands, but not as aggressive as that in UMM, which shortens GC time and saves execution time.

4) *Hyper-Parameter Tuning*: We finally discuss the impacts of three hyper-parameters, i.e., learning rate (α), exploration ratio (ϵ), and discount factor (γ), on Q-ATuMm's performance. We conduct a set of sensitivity analysis tests by setting different values of these hyper-parameters to run PageRank applications. Instead of extensively exploring all possible combinations, we selectively fix any two hyper-parameters and change the third one. Table V summarizes the top 5 combinations that obtain the best latency.

We find that three out of five appropriate values for the learning rate α are 0.3. Although a higher learning rate may guarantee Q-ATuMm converges quickly, it is possible to be trapped in

TABLE V
LATENCY OF TOP 5 HYPER-PARAMETER COMBINATIONS

Learning Rate	0.3	0.3	0.2	0.3	0.7
Exploration Ratio	0.1	0.5	0.2	0.9	0.1
Discount Factor	0.9	0.9	0.9	0.9	0.9
Latency (min)	23	24	24	24	24

a locally optimal solution. A small learning rate ensures that Q-ATuMm can achieve the optimal global solution, even with a slower speed. We also set the exploration ratio ϵ to 0.1 because a lower exploration ratio can allow more exploitation than exploring different states and identify the best values for achieving the optimal performance. As Q-ATuMm has a relatively simple state space, we expect Q-ATuMm to learn on the known states instead of exploring around randomly. Finally, considering that the discount factor determines the importance of future rewards, and PageRank is an iterative application with periodic patterns across time, we find that a significant discount factor (i.e., 0.9) can speed up the convergence.

We also tune the three hyper-parameters of Q-ATuMm to investigate their impacts on the performance of TPC-H applications. Similarly, we extensively change the values from 0.1 to 0.9 for each hyper-parameter and receive the following observations. First, we find that the discount factor is not sensitive for both CPU intensive and I/O intensive queries because most of the queries are completed within a short period before the discount factor takes effect. Second, the exploration ratio is less sensitive for CPU-intensive queries than for I/O intensive queries because CPU-intensive queries hardly benefit from Q-ATuMm. Finally, more than one combination of the three hyper-parameters can lead to the same best performance, which indicates that TPC-H queries are not sensitive to hyper-parameters of Q-ATuMm as they are not iterative applications.

V. CONCLUSION

Apache Spark speeds up large-scale data processing by leveraging in-memory computation. However, the existing Spark memory manager (UMM) incurs long garbage collections, which degrades Spark performance significantly. In this work, we first present a new Spark memory manager (ATuMm) that leverages the feedback of GC time and memory demands to partition the memory pool dynamically. We further adopt a reinforcement learning algorithm to develop an intelligent agent (Q-ATuMm) to manage memory partition for complicated workloads. We implement ATuMm and Q-ATuMm in Spark 2.4.0 and construct experiments in a real Spark cluster. We find that ATuMm obtains around 25% improvement of Spark performance, compared with existing memory managers in the best case. By applying learning-based memory management, Q-ATuMm can further improve Spark's performance to 34%. We contribute the latency improvement to successfully reducing the GC time for both ATuMm and Q-ATuMm. In the future, we plan to evaluate our design on a larger volume of applications with different types of resource intensity. By constructing experiments extensively, we are able to find a hyper-parameter combination that provides optimal performance for general data-processing

942 applications. We also plan to integrate other ML algorithms, e.g.,
943 LSTM, to compare cost and performance with Q-learning.

944

REFERENCES

- 945 [1] M. Armbrust et al., "A view of cloud computing," *Commun. ACM*, vol. 53,
946 no. 4, pp. 50–58, Apr. 2010.
- 947 [2] T. White, *Hadoop: The Definitive Guide*. Sunnyvale, CA, USA: Yahoo
948 Press, May 2012.
- 949 [3] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica,
950 "Spark: Cluster computing with working sets," in *Proc. 2nd USENIX Conf.*
951 *Hot Topics Cloud Comput.*, 2010, pp. 10–10.
- 952 [4] M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction
953 for in-memory cluster computing," in *Proc. 9th USENIX Conf. Netw.*
954 *Syst. Des. Implementation*, 2012, pp. 2–2.
- 955 [5] B. Jang, M. Kim, G. Harerimana, and J. W. Kim, "Q-learning algorithms:
956 A comprehensive classification and applications," *IEEE Access*, vol. 7,
957 pp. 133653–133667, 2019.
- 958 [6] R. S. Xin, J. E. Gonzalez, M. J. Franklin, and I. Stoica, "GraphX: A resilient
959 distributed graph system on spark," in *Proc. 1st Int. Workshop Graph*
960 *Data Manage. Experiences Syst.*, AMPLab, EECS, UC Berkeley, 2013,
961 Art. no. 2.
- 962 [7] L. Xu, M. Li, L. Zhang, A. R. Butt, Y. Wang, and Z. Z. Hu, "MEMTUNE:
963 Dynamic memory management for in-memory data analytic platforms,"
964 in *Proc. Int. Parallel Distrib. Process. Symp.*, 2016, pp. 383–392.
- 965 [8] S.-J. Chae and T.-S. Chung, "DSMM: A dynamic setting for memory
966 management in Apache Spark," in *Proc. IEEE Int. Symp. Perform. Anal.*
967 *Syst. Softw.*, 2019, pp. 143–144.
- 968 [9] M. A. Rahman, J. Hossen, and C. Venkateshaiah, "SMBSP: A self-tuning
969 approach using machine learning to improve performance of spark in Big
970 Data processing," in *Proc. IEEE 7th Int. Conf. Comput. Commun. Eng.*,
971 2018, pp. 274–279.
- 972 [10] D. Nikitopoulou, D. Masouros, S. Xydis, and D. Soudris, "Performance
973 analysis and auto-tuning for spark in-memory analytics," in *Proc. Des.*
974 *Automat. Test Europe Conf. Exhib.*, 2021, pp. 76–81.
- 975 [11] M. Kweun, G. Kim, B. Oh, S. Jung, T. Um, and W.-Y. Lee, "PokéMem:
976 Taming wild memory consumers in Apache Spark," in *Proc. IEEE Int.*
977 *Parallel Distrib. Process. Symp.*, 2022, pp. 59–69.
- 978 [12] Z. Zhu, Q. Shen, Y. Yang, and Z. Wu, "MCS: Memory constraint strategy
979 for unified memory manager in Spark," in *Proc. IEEE 23 rd Int. Conf.*
980 *Parallel Distrib. Syst.*, 2017, pp. 437–444.
- 981 [13] C. Sammut and G. I. Webb, Eds., *Bellman Equation*. Boston, MA, USA:
982 Springer, 2010, pp. 97–97. [Online]. Available: [https://doi.org/10.1007/](https://doi.org/10.1007/978-0-387-30164-8_71)
983 [978-0-387-30164-8_71](https://doi.org/10.1007/978-0-387-30164-8_71)
- 984 [14] U. R. Raval and C. Jani, "Implementing & improvisation of K-means
985 clustering algorithm," 2016.
- 986 [15] R. Xin and J. Rosen, "Tuning Java garbage collection for Apache Spark
987 applications," [Online]. Available: [https://databricks.com/blog/2015/05/](https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html)
988 [28/tuning-java-garbage-collection-for-spark-applications.html](https://databricks.com/blog/2015/05/28/tuning-java-garbage-collection-for-spark-applications.html)
- 989 [16] L. Yue-Peng, "TPC-H analysis and test tool design," *Comput. Eng. Appl.*,
990 2007.
- 991 [17] M. Bayati, J. Bhimani, R. Lee, and N. Mi, "Exploring benefits of NVMe
992 SSDs for BigData processing in enterprise data centers," in *Proc. 5th Int.*
993 *Conf. Big Data Comput. Commun.*, 2019, pp. 98–106.



Danlin Jia is a senior storage architecture engineer
with Memory Solutions Lab, Samsung Semiconductor
Inc.

994
995
996
997

Li Wang is currently working toward the PhD degree with Northeastern Uni-
versity in Boston, Massachusetts.

998
999
1000

Natalia Valencia received the master's degree in cybersecurity from Florida
International University.

1001
1002
1003

Janki Bhimani is an assistant professor with Florida
International University, Miami.

1004
1005
1006






Bo Sheng is an associate professor with Computer Science Department, Uni-
versity of Massachusetts Boston.

1007
1008
1009

Ningfang Mi is an assistant professor with the Department of Electrical and
Computer Engineering, Northeastern University, Boston, Massachusetts.

1010
1011
1012

Thermal Aware System-Wide Reliability Optimization for Automotive Distributed Computing Applications

Ajinkya S. Bankar , Shi Sha , *Senior Member, IEEE*, Janki Bhimani , *Member, IEEE*, Vivek Chaturvedi , *Member, IEEE*, and Gang Quan , *Senior Member, IEEE*

Abstract—As the automotive industry is shifting the paradigm towards autonomous driving, safety guarantee has become a paramount consideration. Temperature plays a key role in the system-wide reliability of the electronic control systems (ECS) used in the automotive. A vehicle is usually subjected to harsh temperature conditions from its operating environment. The increasing power density of IC chips in the ECS further exacerbates the operating temperature and thermal gradient condition on the chip, thereby significantly impacting the vehicle's reliability. In this paper, we study how to map a periodic distributed automotive application on a heterogeneous multiple-core processing architecture with temperature and system-level reliability issues in check. We first present a mathematical programming model to bound the peak operating temperature for the ECS. Then we propose a more sophisticated approach based on the genetic algorithm to effectively bound the peak temperature and optimize the system-wide reliability of the ECS by maximizing its mean-time-to-failure (MTTF). To this end, we present an algorithm to guarantee the peak temperature for periodic applications with variable execution times to ensure our approach's effectiveness. We also present several computationally efficient techniques for system-wide MTTF computation, which show several-order-of-magnitude speed-up over the state-of-the-art method when tested using synthetic cases and practical benchmarks.

Index Terms—Automotive Reliability, System-Level MTTF, Thermal Aware, ECU, ISO 26262.

I. INTRODUCTION

THE mainstream innovation in the automotive industry is driven by electronic systems that have transformed automobiles from a mechanical-only system to a sophisticated

network of embedded systems. Utilizing innovative electronics technologies provide safer integration systems, which reduce human failure in driving, e.g., from traditional GPS positioning to inter-vehicle communication, from the traditional CAN bus to Automotive Ethernet and Media Oriented Serial Transport [1]. Meanwhile, consolidating auto-electronics and enhancing ECS reliability becomes more important as endorsed by ISO-26262 standard [2] because more and more life-critical control decisions are made electronically on the roads.

Temperature plays a key role in terms of reliability for automotive systems. Among various failure mechanisms in an automotive ECS, the temperature-induced fault can reach as high as 55% [3]. The aggressive scaling of transistor sizes increases the chip power density and runtime temperature, and the ever-increasing computing complexity dramatically exaggerates the chip thermal gradient, which could harm the ECS reliability [4]. Moreover, the vehicles undergo stringent environmental conditions with the high temperature of ECUs reaching from 90 °C to 150 °C [5], which accelerates the aging or wear-out due to thermal-induced failure phenomena such as Electromigration (EM), Time-Dependent Dielectric Breakdown (TDDB), Negative Bias Temperature Instability (NBTI), Hot Carrier Injection (HCI), and Thermal Cycling (TC) [6]. Also, exceeding the temperature threshold may cause the ECU to enter the thermal emergency situation and be shut down automatically during runtime [7], which may cause catastrophic consequences. Therefore, the runtime temperature should be carefully managed for reliability concerns to meet the safety requirements for the automotive, e.g., the international standard ISO 26262.

Designers may use various active and passive cooling methods to restrict the ECU temperature, such as convection heat sinks and spreaders in low-end ECUs [8] or passive cooling for high-end ECUs, e.g., Tesla Model 3 [9]. However, relying solely on a more powerful mechanical cooling system cannot solve the thermal crisis in vehicle ECS as they are ineffective in dealing with the localized thermal hotspots due to non-uniform power densities on these ICs. In particular, for the emerging 3D ICs that are a promising alternative for addressing the rapidly growing computation/storage needs of Artificial Intelligence applications [10]–[12], the thermal problems become even more substantial, and the existing active and passive cooling techniques utilized in modern automobiles are powerless [13], [14]. Moreover, the design of the mechanical cooling system requires

Manuscript received 19 February 2021; revised 15 October 2021 and 7 March 2022; accepted 2 June 2022. Date of publication 24 June 2022; date of current version 17 October 2022. This work was supported by the Dissertation Year Fellowship of the University Graduate School, Florida International University. The review of this article was coordinated by Prof. Sohail Anwar. (*Corresponding author: Ajinkya S. Bankar.*)

Ajinkya S. Bankar and Gang Quan are with the Department of Electrical and Computer Engineering, Florida International University, Miami, FL 33172 USA (e-mail: abank013@fiu.edu; gaquan@fiu.edu).

Shi Sha is with the College of Business and Engineering, Wilkes University, Wilkes-Barre, PA 18766 USA (e-mail: shi.sha@wilkes.edu).

Janki Bhimani is with the Knight Foundation School of Computing and Information Sciences, Florida International University, Miami, FL 33199 USA (e-mail: janki.bhimani@fiu.edu).

Vivek Chaturvedi is with the Department of Computer Science And Engineering, Indian Institute of Technology, Palakkad, Kerala 678557, India (e-mail: vivek@iitpkd.ac.in).

Digital Object Identifier 10.1109/TVT.2022.3185978

thermal characteristics of the ECS under different workloads and environments to be incorporated in the design process for better design trade-offs. Hence, thermal-aware resource management is a critical part of the solution to minimize the temperature impact for ECS.

In this paper, we study the problem of how to map periodic distributed automotive applications on a heterogeneous ECU architecture to mitigate thermal-induced system-level reliability degradations. In particular, we aim at electromigration-aware MTTF maximization and latency minimization without violating the temperature threshold on distributed heterogeneous ECUs. Our proposed optimization framework can also be readily utilized to improve other failure types. Overall, we have made the following contributions:

- 1) We develop a simple thermal-aware mathematical programming-based method to bound the peak temperature when mapping periodic vehicle applications on a heterogeneous ECS architecture.
- 2) We find that existing approaches cannot safely bound the peak temperature when tasks' actual execution times vary from their Worst-Case Execution Times (WCETs). To this end, we propose a computation-efficient thermal bounding method that can be safely adopted in practical cases with execution time variance. Further, we formally prove a series of supportive lemmas and theorems in this work to ensure its effectiveness.
- 3) We improve the system-level MTTF computing efficiency. Our proposed formula can be used for problems with higher design complexity and more optimization dimensions than [15], [16]. Further, we extend our MTTF calculation approach to the more general case (when different ECUs may have different aging rates).
- 4) We incorporate our thermal bounding method and system-level MTTF estimation into a genetic algorithm-based approach to map the periodic vehicle applications on ECS. Based on both synthesized test cases and real-life benchmarks, the experimental results show that the proposed approach can achieve $54\times$ to $110\times$ speed-up over the state-of-the-art approach [16].

II. RELATED WORK

An automotive ECS carries safety-critical applications, which demands both stringent real-time responsiveness and high-reliability requirements. In this paper, we improve the optimization framework that can simultaneously deal with ECS temperature limitations, real-time constraints, and reliability requirements.

The thermal safeness ensured in this work is essential for achieving a sustainable running ECS, which prevents automatically triggered power gating, clock throttling, or shutting down ECUs when thermal emergency occurs. Some reliability optimization approaches have been proposed, with no issues of temperature on computing hardware taken into consideration, e.g. [17]–[22]. So, they are not always feasible in thermal-constrained platforms. For example, Huang *et al.* [23] optimized the energy, reliability, and makespan jointly on directed acyclic

graphs (DAGs); however, they did not consider the thermal effects on frequency-dependent transient faults in the reliability framework. Other existing approaches employ simplified power/thermal models, which can either cause thermal threshold violation during runtime or pessimistically predict the system performance. For example, Lee *et al.* [24] utilized constant maximum powers for peak temperature identification on consecutive execution phases, so its predicted peak temperature can overly constrain its actual resource utilization. Moreover, the solution in [24] does not optimize the reliability.

To facilitate more rigorous analytical thermal analyses, some more sophisticated algorithms have been developed to identify the peak temperature [25], [26]. However, their computational cost can be prohibitively high when scaling the application complexity and processing platforms [25]–[27]. To reduce the peak temperature identification complexity, thermal bounding approaches [28], [29] were developed to estimate the highest possible peak temperature for given task sets with different mappings. However, as shown later in this paper, these approaches [28], [29] cannot effectively bound the peak temperature when the task execution time varies. Other works, e.g. [30], [31], use the regression-based model and practical set-up, respectively, to estimate the peak chip temperature. These solutions cannot guarantee that the actual peak temperature stays below the estimated results and are also difficult to be incorporated into an optimization framework. Hence, there is a need to develop more effective and efficient temperature-constrained methodologies that can be safely applied on ECS with additional optimization factors, such as reliability enhancement and makespan minimization.

Many works consider temperature when establishing reliability optimization frameworks [32]. Zhou *et al.* [32] studied the task scheduling problem on a heterogeneous computing platform for latency minimization problem under the peak temperature and reliability constraint. However, their reliability model, i.e., the transient fault model, is independent of temperature. They also ignored the dependency of task executions and assumed that the temperature can reach stable status instantaneously. Ergun *et al.* [33] studied the system reliability maximization problem on IoT systems without considering the timing constraints; thus, their solutions become infeasible for the latency-critical automotive applications. These methodologies fall short of effectively dealing with the thermal challenges and their impacts on system reliability and fulfill critical ECU requirements in automobile system design.

To incorporate system reliability optimization into our work, we adopt the widely used Mean-Time-To-Failure (MTTF) to determine a system's lifetime reliability and develop a fault-tolerant mechanism based on processor/system state [34]. A recent study extended the validation scope of MTTF from Electromigration to Thermomigration and stress-migration induced failure [35]. However, one primary concern is that the computational cost of MTTF calculation can be prohibitively high [15], [16], [36], [37] to obtain high accuracy.

The rest of the paper is organized as follows. We introduce the system models and formally define the problem in Section III. In Section IV, a linear mathematical programming model for

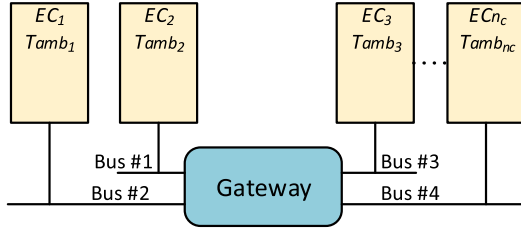


Fig. 1. The heterogeneous ECS Architecture.

a simple thermal approach is presented. In Section V, a more accurate peak temperature bounding method is proposed. In Section VII, our genetic algorithm details, experiment set-up, and results are described. At last, we conclude in Section VIII.

III. PRELIMINARY

In this section, we first discuss the architecture and system models used in this paper. Later, we define the system-level thermal and reliability models.

A. Architecture and Application Models

In this study, we consider integrating multiple ECUs as an ECS targeting safety-critical applications. Assume an ECS consists n_c heterogeneous ECUs interconnected by a bus network through a central gateway [18], [19], [38] as $\mathbf{EC} = \{EC_1, EC_2, \dots, EC_{n_c}\}$ in Fig. 1. Each ECU can be interfaced with various sensors and actuators. As the ECUs have different ambient temperatures as per their mounting locations in the vehicle [5], we assume different ambient temperatures for each ECU, $\mathbf{T}_{amb} = \{T_{amb_1}, T_{amb_2}, \dots, T_{amb_{n_c}}\}$. This ambient temperature accounts for the cumulative effect of the environment temperature and the surrounding electronics/mechanical device heat transfer.

We consider periodic automotive applications as in the existing works [39]–[41], and use the Directed Acyclic Graph (DAG), $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ to model their behavior, which will be mapped on the heterogeneous architecture described above. We assume that the node-set \mathcal{V} consists of total n_v nodes for sensing/control tasks in an ECS as $\mathcal{V} = \{\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_{n_v}\}$. Each task may require different computation times due to the heterogeneity of the ECUs. Therefore, we construct a 2-D matrix for worst-case execution time (WCET) of the task set on different ECUs as,

$$\mathbf{W}_{n_v \times n_c} = \{w_{ik}, i = 1, \dots, n_v; k = 1, \dots, n_c\}, \quad (1)$$

where w_{ik} represents the worst-case execution time of the i -th task (\mathcal{V}_i) executed on the k -th ECU (EC_k). We assume that the overhead for administering task executions can be reckoned by calibrating each task's execution times, and the WCET of the tasks are not affected by the thermal profile. The edge set $\mathcal{E} = \{e_{ij}, \text{ with } \mathcal{V}_i, \mathcal{V}_j \in \mathcal{V}\}$ represents the worst-case communication cost between different task nodes in a system.

B. Power and Thermal Models

The total power consumption of an ECU is a combination of dynamic power (P_d) and leakage power (P_s) [42]. The dynamic

power does not depend on the transient temperature [43], but each task in an automotive system has different dynamic power based on the switching activity [24]. Therefore, to exploit this phenomenon, we consider the dynamic power of task \mathcal{V}_i as, $P_d(i) = \alpha_i \cdot V_{dd}^3$. Where α_i is the activity factor with $\alpha_i = 0$ for the idle task, and V_{dd} is the supply voltage of the ECU.

We assume that the leakage power of an ECU is linearly dependent on the thermal state [43], i.e., $P_s = (\phi_1 \cdot T + \phi_0)$, where T is the temperature of the ECU, ϕ_0 , and ϕ_1 are ECU-dependent constants. Therefore, the all-inclusive total power consumption (P_{total}) of an ECU while running the task \mathcal{V}_i can be expressed as

$$P_{total} = P_d(i) + P_s = V_{dd}^3 \cdot \alpha_i + (\phi_1 \cdot T + \phi_0). \quad (2)$$

We adopt the widely used RC-thermal model [24], [29], [43]–[45] to capture the ECU temperature dynamics. $T(t)$ and $P(t)$ are the transient temperature ($^{\circ}\text{C}$) and its corresponding power consumption ($Watt$) at time instant t , respectively. Then, the transient temperature $T(t)$ follows

$$R_{th} \cdot C_{th} \cdot \frac{dT(t)}{dt} + T(t) - R_{th} \cdot P_{total}(t) = T_{amb_k}, \quad (3)$$

where R_{th}, C_{th} represent thermal-resistance ($^{\circ}\text{C}/W$) and thermal-capacitance ($J/^{\circ}\text{C}$). Given the equation (3), for task \mathcal{V}_i , we can easily identify its ending temperature $T(t_2)$ of the interval $[t_1, t_2]$ with $T(t_1)$ being the initial temperature as

$$T(t_2) = \frac{A(i)}{B} + \left(T(t_1) - T_{amb_k} - \frac{A(i)}{B} \right) e^{-B(t_2-t_1)} + T_{amb_k}, \quad (4)$$

where $A(i) = (\phi_0 + V_{dd}^3 \cdot \alpha_i) / C_{th}$ and $B = 1 / (R_{th} \cdot C_{th}) - \phi_1 / C_{th}$. If the ECU executes the periodic task profile, then the stable status temperature of the ECU can be given by the following theorem (Similar proof is described in Quan *et al.* [43] and hence omitted.)

Theorem 1: Let an ECU, i.e., EC_k runs a periodic schedule with the period of t_p , starting from the ambient temperature (T_{amb_k}). Let T_L be the ending temperature of the first period and let T_{ss} be the temperature when it reaches a stable status. Then,

$$T_{ss} = T_{amb_k} + \frac{T_L - T_{amb_k}}{1 - \mathbb{K}}, \quad (5)$$

where $\mathbb{K} = \exp(-B \cdot t_p)$.

C. Lifetime Reliability Model

In this paper, we focus on temperature sensitive *Electromigration* (EM) wear-out failure mechanism and estimate the system-level reliability of the ECUs as a Mean Time to Failure (MTTF). At the higher temperature and current density, the Electromigration causes displacement of the mass in the conductors of inadequate cross-section, thereby leading to the vacancies in the chip geometry, which can not be recovered. This can be correlated with the mean time to failure as [46],

$$MTTF(T) \propto A_c \cdot J^{-n} \cdot e^{\frac{E_a}{K \cdot T}}, \quad (6)$$

where A_c is a cross-section area of the conductor related constant; J is the current density in Amp/cm^2 ; E_a is the activation

energy in *electron-volts*; K is the Boltzmann's constant; T is the *Kelvin* temperature, and $n = 2$ unless otherwise specified. From equation (6), the lifetime reliability of the device is inversely dependent on the peak temperature, and it would be improved if the peak temperature is reduced.

D. Problem Formulation

For the given automotive DAG application set $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, we seek mapping strategies on the heterogeneous architecture \mathbf{EC} such that the latency is reduced (by satisfying the deadline) and system-level lifetime reliability is maximized (by optimizing the operating temperatures for ECUs judiciously). Simultaneously, we intend to bound the peak temperature to avoid the thermal emergency situation as stated before.

Problem 1: Given \mathbf{EC} and DAG $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, allocate all task nodes in \mathcal{G} to \mathbf{EC} such that (1) the peak temperature of the design (T_{peak}) is lower than the given temperature threshold (T_{max}); (2) the makespan (C_{max}), peak temperature, and system-level lifetime reliability ($MTTF$) of the design are optimized.

IV. MATHEMATICAL PROGRAMMING APPROACH

Clearly, Problem 1 in Section III-D is NP-hard, and different approaches can be applied to solve it, such as mathematical programming [47], simulated annealing [16], genetic algorithm [48], etc. In what follows, we first deal with Problem 1 using the mathematical program approach, as it is a common approach that can produce the optimal solution for an optimization problem.

To satisfy the peak temperature constraint of the \mathbf{EC} after partitioning task set \mathcal{V} , we define the decision variables x_{ik} as,

$$x_{ik} = \begin{cases} 1, & \text{if } \mathcal{V}_i \text{ is assigned to } EC_k; \\ 0, & \text{otherwise.} \end{cases} \quad (7)$$

Each task \mathcal{V}_i must be allocated to only one processor as constrained by,

$$\sum_{k=1}^{n_c} x_{ik} = 1, \quad \forall \mathcal{V}_i \in \mathcal{V}, \forall EC_k \in \mathbf{EC}. \quad (8)$$

Let us define another decision variable σ_i as the starting time of task \mathcal{V}_i , then the makespan of the system is defined to be,

$$\sigma_i + \sum_{k=1}^{n_c} w_{ik} \cdot x_{ik} \leq C_{max}, \quad \forall \mathcal{V}_i \in \mathcal{V}, \forall EC_k \in \mathbf{EC}. \quad (9)$$

In the meantime, if predecessor-successor task pairs are allocated to different ECUs, then the data dependencies among them are managed by the following equation,

$$\sigma_i + w_{ik} \cdot x_{ik} + e_{ij} \cdot (x_{jl} + x_{ik} - 1) \leq \sigma_j, \quad (10)$$

where EC_k and $EC_l \in \mathbf{EC}$, \mathcal{V}_i and $\mathcal{V}_j \in \mathcal{V}$, $\forall e_{ij} \in \mathcal{E}$. On the other hand, the following constraints ensure the executions of two tasks allocated on the same ECU will never overlap,

$$\begin{aligned} \sigma_i + w_{ik} - \sigma_j &\leq \mathcal{M} \cdot (2 - x_{ik} - x_{jk}) \\ \text{OR} \\ \sigma_j + w_{jk} - \sigma_i &\leq \mathcal{M} \cdot (2 - x_{ik} - x_{jk}), \end{aligned} \quad (11)$$

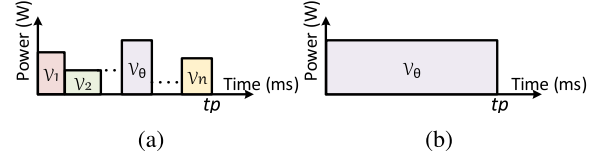


Fig. 2. (a) The WCET schedule with period t_p . (b) A hypothetical schedule with only task \mathcal{V}_θ for the entire period t_p .

where $\forall \mathcal{V}_i \neq \mathcal{V}_j \in \mathcal{V}$, $\forall EC_k \in \mathbf{EC}$, and \mathcal{M} is a large positive constant.

Note that the constraints imposed by the equations (8–11) ensure the minimal makespan and task precedence, but they do not contemplate the thermal behavior in the mathematical model. From equations (3) and (4), we can observe that the temperature is a non-linear function of t and hence unsolvable by linear solvers. Therefore, we use a simple thermal approach in which each task has a constant stable status temperature irrespective of the starting temperature and thermal capacitance of the ECU [29], [49]. In particular, for any task $\mathcal{V}_i \in \mathcal{V}$, we can set $\frac{dT(t)}{dt} = 0$ in equation (3) and obtain the constant stable status temperature as $T = T_{amb_k} + R_{th} \cdot P_{total}$. With this knowledge, we can find the stable status temperature for any $\mathcal{V}_i \in \mathcal{V}$ on EC_k as

$$T_k^*(i) = \frac{A(i)}{B} + T_{amb_k}, \quad \forall T_{amb_k} \in \mathbf{T}_{amb}. \quad (12)$$

Therefore, the highest system-wide temperature (denoted as T^*) can be formulated as

$$T^* = \max_{i,k} (T_k^*(i)), \quad \forall \mathcal{V}_i \text{ allocated on } EC_k. \quad (13)$$

Further, we have the following theorem to ensure that T^* bounds the peak temperature for any resultant task allocation.

Theorem 2: For any mapping of \mathcal{V} to \mathbf{EC} , with periodic execution, the resultant highest system temperature never exceeds T^* .

Proof: Suppose EC_k periodically executes n tasks $\{\mathcal{V}_1, \dots, \mathcal{V}_n\}$ at a period t_p , whose dynamic powers are $\{P_d(1), \dots, P_d(n)\}$ as shown in Fig. 2(a). Let task \mathcal{V}_θ consume the highest dynamic power among all tasks allocated on EC_k , i.e., $P_d(\theta) = \max\{P_d(1), \dots, P_d(n)\}$. Then, for all \mathcal{V}_i allocated on EC_k , since $P_d(\theta) \geq P_d(i)$, we have

$$T_k^*(\theta) \geq T_k^*(i). \quad (14)$$

Using a hypothetical schedule that keeps at a constant power consumption $P_d(\theta)$ in one period as Fig. 2(b) can bound the peak temperature of the schedule shown in Fig. 2(a) because both the execution time and power consumption for all \mathcal{V}_i are not larger than \mathcal{V}_θ on EC_k in Fig. 2(b). When applying equation (14) on all ECUs, $T^* = \max_{i,k} (T_k^*(i))$ can effectively bound the system-wide peak temperature. \square

With Theorem 2, we can readily add a peak temperature-related constraint as follows,

$$T^* \leq T_{max}. \quad (15)$$

Also, we need to incorporate the minimization of peak temperature into the design objective. Note that to reduce the makespan and to reduce the peak temperature are two conflicting design objectives. How to deal with multi-criteria optimization in a more sophisticated manner is not the focus of this paper. Instead, we use a weighted sum as our optimization objective as follows,

$$\text{Max} : \mathbb{W}_1 \cdot \frac{C_s - C_{\max}}{C_s} + \mathbb{W}_2 \cdot \frac{T_{\max} - T^*}{T_{\max}}, \quad (16)$$

where $\mathbb{W}_1, \mathbb{W}_2$ are the weights and $\mathbb{W}_1 + \mathbb{W}_2 = 1$. C_s is the minimal makespan that a task set can complete on a given ECS without considering its temperature constraint, reliability optimization, etc. The mathematical program minimizes the makespan of the application, but it can be readily adopted to use makespan as a strict constraint according to the application's real-time requirement.

The mathematical programming approach presented in this section has three challenges. First, it is well known that the proposed method is NP-hard in nature. Second, to identify the peak temperature, this approach adopts a strategy that assumes an ECU can immediately reach its stable status [49], which is pessimistic especially when the task execution time is very short (from tens of microseconds to several milliseconds [29]). Third, we can't incorporate system-level lifetime reliability in the mathematical programming framework due to the non-linearity. Therefore, we resort to the meta-heuristic genetic algorithm [48] to solve the Problem 1. In the following sections, we first study how to capture the peak temperature under a task mapping configuration, and then, we develop efficient MTTF calculations accordingly.

V. BOUND THE WORST-CASE PEAK TEMPERATURE

A key to solve Problem 1 in Section III-D is to estimate an accurate peak temperature for a given task/ECU mapping configuration. Several approaches have been proposed to estimate the peak temperature of a processor. For example, the numerical method splits each execution interval into short fragments and attempts to find the peak temperature by checking each small stretch (e.g. [50]–[52]). Also, an epoch-based peak temperature detection methodology has been proposed using a mix of analytical and numerical methods in [44] or by solving the first-order derivative on each processing core via the Newton-Raphson method in [45]. Another approach greedily searches the worst-case task arrival cases to bound the runtime peak temperature [26]. However, these approaches have very high computation complexity (e.g. [26], [50]–[52]), which makes them ineffective during the design space exploration.

For more computationally efficient approaches, besides the *simple thermal approach* stated in Section IV, Chaturvedi *et al.* [28] proposed a so-called “hypothetical step-up schedule” to bound the peak temperature for a periodic schedule. However, as shown in what follows, this approach works under the assumptions of each task instance always takes its worst-case execution times. It may fail when the task's execution time vary in runtime, which is quite normal in practical automotive scenarios [53].

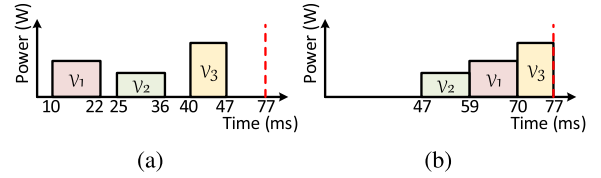


Fig. 3. An example of the step-up schedule.

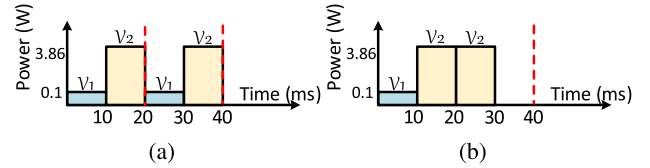


Fig. 4. A motivational example.

A. Motivation Examples

In real-time computing, it is a common practice to use the worst-case execution times to bound the longest completion time of a task set under a scheduling policy. However, we find that using the worst-case execution time-based real-time schedules (e.g. [50]–[52]) can not always identify the highest possible peak temperature during runtime. To put our discussions into perspective, assume a task set contains three periodic tasks with the same period, and one period of its schedule (based on the worst-case execution times) is depicted in Fig. 3(a). Fig. 3(b) shows its corresponding *step-up* schedule, with the same interval lengths but organized so that the dynamic power consumptions are monotonically increasing from the first interval to the last. It has been formally proved that the peak temperature of a schedule is bounded by its corresponding “step-up” schedule [28], [29]. For example, if $w_{1k} = 12$ ms, $w_{2k} = 11$ ms, $w_{3k} = 7$ ms and $P_d(1) = 2.204$ W, $P_d(2) = 0.962$ W, $P_d(3) = 2.924$ W with a period 77 ms, using the experimental set-up in Section VII-A, the peak temperature of the step-up schedule (Fig. 3(b)) is 51.97°C , which bounds the actual peak temperature of 51.69°C for the schedule in Fig. 3(a). It is much smaller than the peak temperature of 125.45°C using the simple peak temperature prediction approach of Section IV.

The example shown in Fig. 3 demonstrates that the step-up schedule can guarantee peak temperature only when all the tasks run with their worst-case execution times (WCETs). However, the temperature bound given by a step-up schedule may be violated when the tasks run with execution times lower than their WCETs, as shown in the following motivational example. Fig. 4(a) shows a step-up schedule with two tasks \mathcal{V}_1 and \mathcal{V}_2 . Assume the dynamic power consumptions and the WCETs for \mathcal{V}_1 (\mathcal{V}_2 resp.) are 100 mW (3.86 W resp.) and 10 ms (10 ms resp.), with a period of 20 ms. When tasks \mathcal{V}_1 and \mathcal{V}_2 take their WCETs as Fig. 4(a) and execute long enough to reach their thermal stable status, the peak temperature is 93.61°C . However, the actual task execution time may vary with the concurrent workload on one chip [54]. Without losing the generality, assume in a period of the stable status, task \mathcal{V}_1 changes its execution time between 0 and 10 ms as shown in Fig. 4(b). Then, the highest peak temperature can reach 94.02°C in Fig. 4(b) and violate the peak

temperature predicted by Fig. 4(a). This motivation example clearly shows that the step-up schedule can not effectively bound the peak temperature for a periodic schedule when task execution times are variable. Although it is possible to greedily search all possibilities of execution time combinations (e.g., in [26]) for the highest possible temperature, the computational cost could be prohibitively high when applying to multiple-ECU cases with tens to hundreds of applications. Therefore how to *efficiently* bound the worst-case peak temperature remains a problem.

In what follows, we propose a time-efficient algorithm to bound the peak temperature for variable execution time scenarios.

B. Accurate Peak Temperature Identification Algorithm

We begin with several lemmas to support the algorithm that tightly bounds the peak temperature for variable execution time scenarios and later prove a theorem to validate the proposed algorithm's effectiveness.

Lemma 1: Let EC_k run n tasks $\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_n$ consecutively with a t_p period. Let $T_m(q \cdot t_p)$ be the temperature at $t = q \cdot t_p$ when all tasks are executed with their WCETs. Then, we have $T_m(q \cdot t_p) \geq T(q \cdot t_p)$, if $T(q \cdot t_p)$ is the temperature at $t = q \cdot t_p$ when at least one task instance from $t \in [0, q \cdot t_p]$ runs with execution time less than its WCET.

Lemma 2: Let EC_k run \mathcal{V}_i during an interval $t \in [t_1, t_2]$ and let EC_k 's temperature at t_1 be T_1 . Then EC_k 's temperature monotonically increases (decreases *resp.*) within interval t , if $T_k^*(i) \geq T_1$ ($T_k^*(i) \leq T_1$ *resp.*).

As per Lemma 1, if at least one of the tasks on ECU executes shorter than its WCET, then the temperature at the end of q -th period is no larger than its WCET counterpart. From Lemma 2, we deduce that if the individual task stable status temperature is higher (lower *resp.*) than the starting point temperature in an interval, the ECU temperature monotonically increases (decreases *resp.*) when running in a constant execution mode during the interval. The detailed proofs of the lemmas are described in Appendices A and B, respectively.

With Lemma 1 and 2, an algorithm is developed to identify the peak temperature when an ECU runs a set of tasks that considers task completion time variation during the runtime.

Algorithm 1 first calculates the stable status temperature at the end of the application period, assuming that each task takes its WCET (line 1). Then, lines 2-8 iteratively identify the highest temperature of each interval in the schedule. Specifically, if a task's stable state temperature is higher than the current temperature bound (T_m), the task is taken into account to bound the possible higher peak temperature. Otherwise, the current interval is not considered in determining the peak temperature bound (T_m). Finally, the ending temperature for the last interval is output as the peak temperature (line 9). The complexity of Algorithm 1 is $O(n)$, where n is the number of tasks allocated to an ECU. Also, we formulate Theorem 3 as follows to ensure the guarantee of the peak temperature identified through Algorithm 1.

Theorem 3: Let tasks $\{\mathcal{V}_1, \dots, \mathcal{V}_n\}$ be executed on ECU EC_k periodically with variable execution times. Then, the highest

Algorithm 1: Bounding the Peak Temperature with Execution Time Variance.

Inputs: Task mapping based on WCET schedule of EC_k ; Power/thermal parameters; Timing parameter matrix $\mathbf{W}_{n_v \times n_c}$;

Output: Temperature bound T_m of EC_k .

- 1: Let $T_m = T_{ss}$ in Theorem 1 for the WCET schedule;
 - 2: **for** each task \mathcal{V}_i allocated on EC_k **do**
 - 3: Calculate $T_k^*(i)$ based on equation (12);
 - 4: **if** $T_k^*(i) > T_m$ **then**
 - 5: $T_m =$ the ending temperature of running \mathcal{V}_i with initial temperature T_m ;
 - 6: $T_m = T'_m$;
 - 7: **end if**
 - 8: **end for**
 - 9: **return** T_m
-

possible temperature during the execution is no more than T_m output from Algorithm 1.

This theorem is proved with the help of Lemma 1 and Lemma 2 in Appendix C. It substantiates the effectiveness of the peak temperature output from the proposed Algorithm 1. Next, we discuss our new methods to compute the system-level MTTF.

VI. COMPUTATIONALLY EFFICIENT SYSTEM-LEVEL MTTF FORMULATION

In this section, we first briefly introduce the relevant background for the state-of-the-art method that accurately calculates the system-level MTTF [16]. We then discuss the limitations of the method in [16] and propose our computationally efficient approaches to estimate the system-level MTTF.

Without losing the generality, we adopt the Weibull distribution to model the wear-out effects at the system-level [16]. The reliability of a single ECU at time instant t with temperature T is

$$\mathcal{R}(t, T) = e^{-\left(\frac{t}{\eta(T)}\right)^\beta}, \quad (17)$$

where $\eta(T)$ and β represent scale and slope parameters of the Weibull distribution, respectively. Then, we have

$$\eta(T) = \frac{MTTF(T)}{\Gamma\left(1 + \frac{1}{\beta}\right)}. \quad (18)$$

When executing periodic tasks, the reliability of an ECU in one period (t_p) is a function of the temperature and duration for each sub-interval ($a = 0, \dots, s-1$) [16], which can be formulated as

$$\mathcal{R}(t_p) = e^{-\left(\sum_{a=0}^{s-1} \frac{\Delta t_i}{\eta(T_i)}\right)^\beta}. \quad (19)$$

Let the aging factor of the ECU [16] be

$$\mathcal{A} = \sum_{a=0}^{s-1} \frac{\Delta t_i}{\eta(T_i)}. \quad (20)$$

Then, the reliability of the ECU for q consecutive periods is

$$\mathcal{R}(q \cdot t_p) = e^{-\left(\sum_{a=0}^{q(s-1)} \frac{\Delta t_a}{\eta(T_a)}\right)^\beta} = e^{-\left(q \cdot \sum_{a=0}^{s-1} \frac{\Delta t_a}{\eta(T_a)}\right)^\beta} = e^{-(q \cdot A)^\beta}.$$

Therefore, the reliability of the system with n_c ECUs can be expressed as

$$\mathcal{R}_{system}(q \cdot t_p) = e^{-\sum_{b=1}^{n_c} (q \cdot A_b)^\beta}. \quad (21)$$

As $MTTF \gg t_p$, the system-level $MTTF$ approximation can be calculated at the end of each period as

$$MTTF_{system} = \sum_{a=0}^{\infty} e^{-\sum_{b=1}^{n_c} (a \cdot A_b)^\beta} \cdot t_p. \quad (22)$$

To reduce the computational cost of (22), a more efficient method, Speedup Technique-I is proposed in [16] by assuming the reliability of an ECU keeps the same for every v periods. The estimated system-level $MTTF$ is

$$MTTF_{system}^{speed_up-I} = \sum_{a=0}^{\infty} e^{-\sum_{b=1}^{n_c} (a \cdot A_b \cdot v)^\beta} \cdot t_p \cdot v. \quad (23)$$

The accuracy of the state-of-the-art system-level $MTTF$ in (23) depends on three parameters, i.e., the highest value of a , period t_p , and the number of periods v used to expedite the approximation. It is advisable to reach a high value of a until the $MTTF$ is saturated, which drains a significant amount of computation time, as later observed in Section VII-B. To achieve higher computational efficiency, it is necessary to find an alternative strategy to estimate the system-level $MTTF$ efficiently, reducing the computation time without compromising the $MTTF$ estimation accuracy significantly.

A. A Computing Efficient Approach for $MTTF$ Calculation

When optimizing the system design goal such as $MTTF$ using meta-heuristics like a genetic algorithm or simulated annealing, high computation efficiency of the design objective function can enable a more effective design space exploration process. As mentioned before, the approach based on (23) is very time-consuming, which severely limits its searching scope and efficiency. Therefore, it is highly desirable that a more computing efficient approach for the $MTTF$ calculation can be utilized in the search engine during the design space exploration process.

As the automotive applications have task interval lengths in ms to μ s [18], [55], an ECU's temperature dynamics can be safely treated as a constant value in a period. This fact helps to simplify the system-level formulation of $MTTF$. Let $\eta_1, \eta_2, \dots, \eta_{n_c}$ be the scale parameters of the EC given by (18). We assume that they depend on a constant temperature over the period, which can be calculated using the average temperature

of the period in the stable status

$$T_{avg,b} = \frac{\sum_{a=0}^{s-1} \int_{l_a}^{l_{a+1}} \frac{A(a)}{B} + (T(t_a) - T_{amb_k} - \frac{A(a)}{B})e^{-B \cdot \Delta_a} + T_{amb_k}}{t_p}, \quad (24)$$

where s represents the number of intervals in a period, $\Delta_a = (l_{a+1} - l_a)$ is the length of each interval, t_p is the period, and b is the ECU index from 1 to n_c .

To further improve the computational efficiency and safely bound the $MTTF$ in an ECS, we take advantage of using a uniform slope parameter of the Weibull distribution on all the ECUs to expedite $MTTF$ computations by adopting the worst-case wear-out rate on each unit. The ECS system reliability can then be formulated as

$$\mathcal{R}_{system} = e^{-\sum_{b=1}^{n_c} \left(\frac{t}{\eta_b}\right)^\beta}. \quad (25)$$

Based on equation (25), we can formulate the system-level $MTTF$ as

$$MTTF_{system}^{Fast-same} = \int_0^{\infty} e^{-\sum_{b=1}^{n_c} \left(\frac{t}{\eta_b}\right)^\beta} dt. \quad (26)$$

By substituting $x = \sum_{b=1}^{n_c} \left(\frac{t}{\eta_b}\right)^\beta$ in (26), we have

$$x = t^\beta \cdot \sum_{b=1}^{n_c} \left(\frac{1}{\eta_b^\beta}\right). \quad (27)$$

$$x^{\frac{1}{\beta}} = t \cdot \left(\sum_{b=1}^{n_c} \left(\frac{1}{\eta_b^\beta}\right)\right)^{\frac{1}{\beta}}. \quad (28)$$

Further, we have

$$t = \frac{x^{\frac{1}{\beta}}}{\left(\sum_{b=1}^{n_c} \left(\frac{1}{\eta_b^\beta}\right)\right)^{\frac{1}{\beta}}}, \quad (29)$$

$$dt = \frac{x^{\frac{1}{\beta}-1} \cdot dx}{\beta \cdot \left(\sum_{b=1}^{n_c} \left(\frac{1}{\eta_b^\beta}\right)\right)^{\frac{1}{\beta}}}. \quad (30)$$

Note that $t \rightarrow 0$ leads to $x \rightarrow 0$ and $t \rightarrow \infty$ leads to $x \rightarrow \infty$. Now, we can factor out (26) and obtain our fast $MTTF$ formula as

$$MTTF_{system}^{Fast-same} = \frac{1}{\beta \cdot \left(\sum_{b=1}^{n_c} \left(\frac{1}{\eta_b^\beta}\right)\right)^{\frac{1}{\beta}}} \int_0^{\infty} e^{-x} \cdot x^{\frac{1}{\beta}-1} \cdot dx, \quad (31)$$

which converges to the following simple form [56],

$$MTTF_{system}^{Fast-same} = \frac{\Gamma\left(\frac{1}{\beta}\right)}{\beta \cdot \left(\sum_{b=1}^{n_c} \left(\frac{1}{\eta_b^\beta}\right)\right)^{\frac{1}{\beta}}}. \quad (32)$$

Equation (32) is independent of compute-expensive integration operations, and $\Gamma(\frac{1}{\beta})$ can be readily calculated. So, our proposed MTTF estimation method in (32) is highly computation efficient and can tightly bound the MTTF for a heterogeneous ECS. As shown in Section VII-A(B), the proposed MTTF formula in (32) can speed-up the computing efficiency of (23) by $12\times$ for synthetic test cases and $164\times$ for the *Real-Life* benchmark. Moreover, the proposed approximation approach can achieve an average error below 0.1% for the estimation of system-level MTTF.

B. Fast System-Level MTTF Calculation for ECUs With Different Degradation Rates

The MTTF formula proposed in Section VI-A can accurately estimate a system-level MTTF, when (i) the temperature fluctuation in one period is small enough to be considered as a constant value, e.g., when applications' periods on an ECS are as short as several μs or ms ; (ii) the ECU's wear-out rate has a negligible variance compared to the worst-case wear-out rate in an ECS, e.g., when all ECUs in an ECS have similar operational status and ambient temperatures. However, according to [5], the ambient temperature of different ECUs in one vehicle could vary as much as $90^\circ\text{C} - 150^\circ\text{C}$ due to different mounting locations, which may potentially cause a large wear-out rate variance within an ECS. Therefore, equation (32) could result in an inaccurate system-level MTTF estimation without an adequately justified wear-out rate.

When adopting the uniform wear-out rate MTTF formula in equation (32), results using the same slope parameter (β) of the Weibull distribution can help to save several magnitudes of the computational cost, but they may also deviate from the practical heterogeneous ECU settings significantly. Although the state-of-the-art method for system-wide MTTF calculation in equation (23) can be used to address the heterogeneity of the ECU wear-out rates, its computational cost could be prohibitively high. To this end, we use heterogeneous thermal wear-out rates of all the ECUs to construct system-wide statistical wear-out rate value, which can be safely implanted into equation (32). Specifically, let $\mathfrak{B} = \{\beta_1, \dots, \beta_{n_c}\}$ be the slope parameters of the Weibull distribution for ECUs. Then, we implant four different statistical parameters in (32), including maximal, minimal, average, and geometric mean values, as follows:

- 1) Let $\beta = \beta_{\max}$, where

$$\beta_{\max} = \max\{\beta_1, \dots, \beta_{n_c}\}. \quad (33)$$

- 2) Let $\beta = \beta_{\min}$, where

$$\beta_{\min} = \min\{\beta_1, \dots, \beta_{n_c}\}. \quad (34)$$

- 3) Let $\beta = \beta_{\text{avg}}$, where

$$\beta_{\text{avg}} = \text{average}\{\beta_1, \dots, \beta_{n_c}\} = \frac{\beta_1 + \beta_2 + \dots + \beta_{n_c}}{n_c}. \quad (35)$$

- 4) Let $\beta = \beta_{\text{geo}}$, where

$$\beta_{\text{geo}} = \text{geo mean}\{\beta_1, \dots, \beta_{n_c}\} = \sqrt[n_c]{\beta_1 \cdot \beta_2 \cdot \dots \cdot \beta_{n_c}}. \quad (36)$$

With different statistical wear-out rate approximations given by (33)-(36), we can obtain an effective slope parameter for each ECU as,

$$\eta_b = \frac{MTTF(T_{\text{avg},b})}{\Gamma(1 + \frac{1}{B})}, \quad (37)$$

where $b = 1, \dots, n_c$ and $B \in \{\beta_{\max}, \beta_{\min}, \beta_{\text{avg}}, \beta_{\text{geo}}\}$. With the help of (32) we estimate the computationally efficient system-level MTTF for different wear-out rates of the EC as,

$$MTTF_{\text{system}}^{\text{Fast-diff}} = \frac{\Gamma(\frac{1}{B})}{B \cdot \left(\sum_{b=1}^{n_c} \left(\frac{1}{\eta_b} \right) \right)^{\frac{1}{B}}}. \quad (38)$$

Using the same parameter settings in Section VII-A(A), we compare the MTTF accuracy of (38) with state-of-the-art equation (23). We observe that the geometric mean ($\beta = \beta_{\text{geo}}$) generates the lowest error in average ($\approx 2\%$) among all the candidates for both synthetic test cases and practical benchmarks. Therefore, the rest of the paper uses $\beta = \beta_{\text{geo}}$ for fast system-level MTTF calculation for ECUs with different degradation rates with minimal accuracy degradation.

With a safer peak temperature bound and a more accurate and efficient MTTF formulation, we are ready to employ a genetic algorithm for Problem 1. Our genetic algorithm implementation is similar to that presented in [48], and its set-up is briefly described in the following section.

VII. EXPERIMENT RESULTS

In this section, we present our experiments and results to validate the performance of our proposed approaches with different parameter settings.

A. Experimental Set-Up

Our genetic algorithm is set-up as follows:

- *Chromosome design*: Each chromosome represents a feasible solution to the task scheduling problem. Chromosome comprises the first part as task mapping and later as the task schedules. We randomly initialized the task mappings on the given ECUs, and initial task schedules were generated with respect to the precedence constraints of the DAG $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$. The pool of the initial population was randomly generated with the chromosomes for the given size.
- *Fitness function*: The formulation of the fitness function is crucial to refine the ECU design alternatives, and it also impacts the solution's superiority using a metaheuristic approach. To solve Problem 1, we need to co-optimize the makespan, peak temperature, and MTTF. First, we calculated the makespan similar to that in [48]. Then, we adopted our proposed peak temperature bound in Algorithm 1 and fast system-level MTTF with the same wear-out rate together. In another experiment, we endorsed the proposed fast system-level MTTF formulation's effectiveness with different wear-out rates. Accordingly, our fitness functions are:

- 1) The peak temperature for both Genetic Algorithms (GAs) is calculated using our proposed Algorithm 1. The system-level MTTF values are estimated by (23) and (32) for both GAs to compare their performance. We normalized the makespan, temperature, system-level MTTF and defined the weighted fitness function as,

$$f_1 = \mathbb{P}_1 \frac{C_l - C_{\max}}{C_l} + \mathbb{P}_2 \frac{T_{\max} - T_m}{T_{\max}} + \mathbb{P}_3 \frac{MTTF_{system}}{MTTF_{\max}}, \quad (39)$$

where $\mathbb{P}_1, \mathbb{P}_2, \mathbb{P}_3$ are the weights having $\mathbb{P}_1 + \mathbb{P}_2 + \mathbb{P}_3 = 1$, and C_l is the smallest possible makespan derived from the solutions of the first generation. $MTTF_{\max}$ is a system-level maximum MTTF estimated by assuming the ambient temperature for each ECU.

- 2) Our experimental study also examines the efficacy of the proposed system-wide MTTF calculation for ECUs with different wear-out rates by implementing two separate GAs with the system-level MTTF calculation by (23) and (38), respectively. We used a normalized weighted sum of the makespan and system-level MTTF in the objective function as

$$f_2 = \mathbb{R}_1 \frac{C_l - C_{\max}}{C_l} + \mathbb{R}_2 \frac{MTTF_{system}}{MTTF_{\max}}, \quad (40)$$

where $\mathbb{R}_1, \mathbb{R}_2$ are the weights with $\mathbb{R}_1 + \mathbb{R}_2 = 1$ and other parameters have the same significance as in the above case-(1). The fitness functions given by equations (39) and (40) explore the design space to search the solution with minimum makespan of the application by satisfying other design constraints.

- *Parent/survivor selection:* Due to multi-objective fitness functions, we maintained the Pareto optimal front during the evolution process from generations-to-generations and used the tournament selection policy to choose the survivors for the next generation [48]. In the tournament selection, we randomly chose 10 chromosomes, sorted them based on their fitness, and picked the top 5 for crossover and mutation. The remaining 5 elements act as the non-evolved chromosomes for the next generation.
- *Crossover/mutation operators:* We used an adaptive crossover operator on the task mapping part of the survivor chromosomes. For randomly chosen two chromosomes, a crossover point is randomly selected between 1 to n_v , and the parts of the two chromosomes after it are exchanged to produce two descendants. Afterward, the crossover descendants are mutated, where any of the pre-mapped tasks in a chromosome is randomly selected, and its mapping is randomly altered to another ECU. (see [48] for more details)

We utilize both synthetic test cases and practical benchmarks to verify the performance of the proposed approaches in terms of temperature bound and system-level MTTF formulations. The parameters for the ECUs used in our experiments are described in Table I. For synthetic test cases, the task graphs were randomly

TABLE I
ECU PARAMETERS FOR THE EXPERIMENT [24], [57]

$\phi_0(\text{A})$	$\phi_1(\text{A}/^\circ\text{C})$	$C_{th}(\text{J}/^\circ\text{C})$	$R_{th}(^\circ\text{C}/\text{W})$	max. Power(W)
0.035	0.016	0.0454	22	3.86

TABLE II
BENCHMARK PARAMETERS

Benchmark	n_c	n_v	$\mathbf{P}_d(\text{task})$	$\mathbf{W}_{n_v \times n_c, \mathcal{E}}$	\mathbf{T}_{amb}
Real-Life [‡]	16	31	[0.1W, 3.86W] [¶]	[100 μs , 400 μs] [‡]	[35 $^\circ\text{C}$, 45 $^\circ\text{C}$]
ACC [§]	4	20	[0.1W, 3.86W] [¶]	Specified for each task/edge [§]	[35 $^\circ\text{C}$, 45 $^\circ\text{C}$]

[‡]described in [18], [§]described in [55], [¶]described in [24].

generated using a Task Graph Generator [18] with *average computation cost* = 15 ms, *communication to computation ratio* = 1, and the *shape* parameter varied in the range of [0.3, 0.9] to cover a wide gamut of the test cases. The dynamic power for all task nodes was chosen to be uniformly distributed in the range of [0.1W, 3.86W] adhering to practical automotive systems [24]. Apart from synthetic tests, we verified our proposed formulations on two practical benchmarks, *Real-Life* [18] and *Automotive Cruise Controller (ACC)* [55], with their parameters listed in Table II.

To achieve about 10 years of a lifetime by assuming a drive time of 2.5 hours per day for an ECU at the temperature of 80 $^\circ\text{C}$ [42], we used $A_c = 10^{-7} \text{cm}^2$, $J = 5.7 * 10^5 \text{Amp/cm}^2$, and $E_a = 0.48 \text{eV}$ to estimate the electromigration-induced MTTF [46]. For the existing model of system-level reliability, we used $v = 100$ in equation (23) [16]. The ambient temperatures were randomly chosen in the interval [35 $^\circ\text{C}$, 45 $^\circ\text{C}$] to account for the variation of the ECU mounting locations if not otherwise specified.

Our experiments were lined-up in the following two major categories:

- Verification of different approaches to our target research problem:* For this set of experiments, we compare the effectiveness and efficiency of different approaches to our research problem, i.e., Problem 1 in Section III-D. Our experiments were conducted based both on synthetic test cases as well as practical real-life benchmarks. Specifically, for synthetic test cases, we adopted an ECS containing 3 ECUs with 10, 15, 20, and 25 tasks to simplify tracing parameter trade-offs. A sufficiently high-temperature threshold T_{\max} of 150 $^\circ\text{C}$ was set because the *simple thermal approach* can lead to a very high peak temperature and the mathematical solver is unable to produce a feasible solution under a tightly constrained T_{\max} otherwise. In our genetic algorithm implementation, the initial solution pool was chosen to be 500 random samples, and the genetic algorithm was progressed for 400 generations, thereby terminating with Pareto optimal front solution in the end. The uniform worst-case slope parameter of the Weibull distribution was chosen to be $\beta = 2$ [16]. For both synthetic test cases and benchmarks, without losing

TABLE III
AVERAGE CPU TIME

Num. of tasks	10	20	30	40	50	Real-Life	ACC
Avg. CPU time (ms) for MTTF method in [16]	44.21	48.59	50.46	52.96	56.40	685.156	49.84
Avg. CPU time (ms) for proposed MTTF method	3.609	3.812	3.906	4.063	4.188	4.17	3.859

TABLE IV
AVERAGE MTTF ERROR PERCENTAGE

Num. of tasks	10	20	30	40	50	Real-Life	ACC
Avg. error (%)	0.023	0.032	0.050	0.067	0.084	0.00033	0.077

generality, we set the weight combinations as $\mathbb{W}_1 = \mathbb{W}_2 = 1/2$ and $\mathbb{P}_1 = \mathbb{P}_2 = \mathbb{P}_3 = 1/3$, which commits an equal influence to each parameter.

The four approaches tested in this category are stated below:

- *Temperature oblivious approach (M-TO)*: The mathematical programming model that minimizes the application makespan but does not consider thermal behavior as the state-of-the-art integer linear programming method in [47].
 - *Simple thermal aware approach (M-STA)*: The mathematical programming approach explained in Section-IV that assumes an ECU reaches its stable status temperature immediately [29], [49].
 - *Accurate peak temperature identification with fast MTTF calculation (G-APTI-FMC)*: The genetic algorithm-based approach bounds the peak temperature accurately with the proposed Algorithm 1 and proposed fast system-level MTTF formulation using a uniform system-wide wear-out rate as described in Section-VI-A.
 - *Accurate peak temperature identification with traditional MTTF calculation (G-APTI-TMC)*: The genetic algorithm-based approach that identifies the peak temperature accurately using the proposed Algorithm 1 and the state-of-the-art system-level MTTF formulation in equation (23) [16].
- b) *Verification of the proposed approaches for fast system-level MTTF calculation*: For this set of experiments, we focus our study on the efficacy of the proposed approaches for fast MTTF calculation. To study the performance of our fast MTTF calculation based on the same degradation rate, i.e., based on equation (32), we assumed that all tasks were assigned to a single ECU in our generated synthesized test cases and real-life benchmarks. We compared this approach with the traditional approach (i.e., [16]), and the results for the average CPU times and error percentages by these two approaches were collected and listed in Table III and Table IV, respectively.

We then extended our experiments to more sophisticated scenarios when different ECUs may have different degradation rates. To this end, we assumed the system contains

100 tasks and the number of ECUs to be 5, 10, 15, and 20. We randomly chose the slope parameter β for each ECU in the range [2, 3], which signifies the early lifetime wear-out [58]. We used the equal weights as, $\mathbb{R}_1 = \mathbb{R}_2 = 0.5$ in this set-up. The following approaches were implemented and tested:

- *Traditional MTTF calculation with different degradation rates (G-TMC-DDR)*: It is a genetic programming-based approach with the objective of makespan minimization and system-level MTTF maximization, which is estimated by the existing method in equation (23) [16]. Here we used the GA with the initial solution space of 100 samples and evolution over 200 generations.
- *Fast MTTF calculation with different degradation rates for small size of GA (G-FMC-DDR-S)*: It is a genetic algorithm-based approach that minimizes the makespan and maximizes the system-level MTTF estimated by the proposed formulation in Section-VI-B with an initial population pool of 100 samples, and we evolve it for 200 generations.
- *Fast MTTF calculation with different degradation rates for large size of GA (G-FMC-DDR-L)*: It is also a genetic algorithm-based approach that minimizes the makespan and maximizes the system-level MTTF estimated by the proposed formulation in Section-VI-B with an initial population pool of 500 samples and evolved for 1000 generations.

The industrial-grade linear solver CPLEX 12.10.0 was used to solve the formulated mathematical programming problems of M-TO and M-STA. Whereas G-APTI-FMC, G-APTI-TMC, G-FMC-DDR, G-TMC-DDR-S, and G-TMC-DDR-L were implemented using Matlab R2020a, running on an HP Z800 server with 24 cores and 32 GB memory.

B. Experimental Results and Discussions

This section presents the experimental results for the set-up described in the previous section and discusses our findings in detail.

- a) *Verification of different approaches to our target research problem*: Fig. 5 shows the results (average peak temperature, average system-level MTTF, average makespan, and average CPU time) by four different approaches, i.e., *M-TO*, *M-STA*, *G-APTI-FMC*, and *G-APTI-TMC*, as introduced above.

From Fig. 5(c), we can see that *M-TO* has the smallest makespan among all the approaches, but Fig. 5(a) and Fig. 5(b) show that *M-TO* has the highest peak temperature and the lowest system-level MTTF. This result clearly indicates that constraining the peak temperature in an automotive ECS is necessary for hardware protection and system-wide lifetime reliability enhancement. When thermal behavior is considered, the makespans for other approaches become longer since the temperature constraint limits the computational throughput on an ECU. Also, with increased thermal prediction accuracy, *G-APTI-FMC* and *G-APTI-TMC* allow to have a longer makespan but

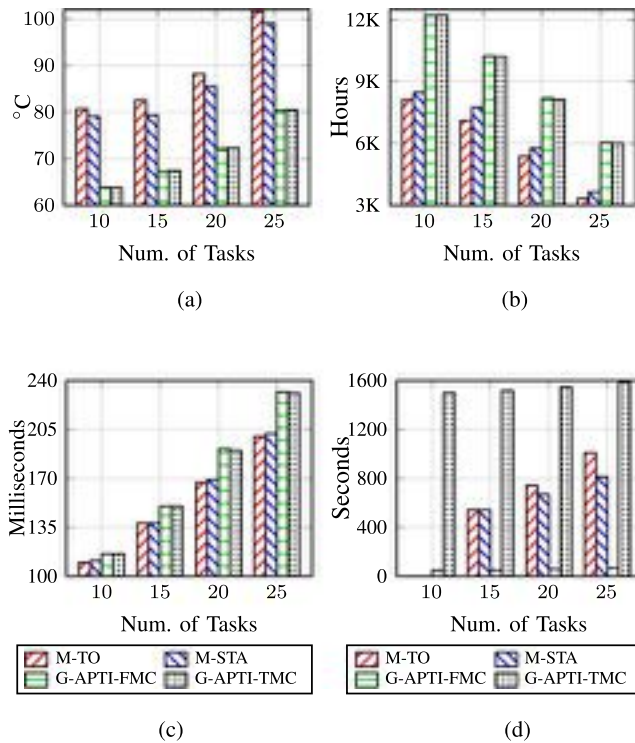


Fig. 5. Experimental results for 3-Processor system. (a) Avg. system peak temperature. (b) Estimated avg. system-level MTTF. (c) Avg. makespan. (d) Avg. CPU time.

a much lower peak temperature and higher system-level MTTF, as clearly shown in Fig. 5(a) and 5(b).

Meanwhile, from Fig. 5(a), we observe that the average peak temperatures for *G-APTI-FMC* and *G-APTI-TMC* differ approximately in the range of 0.1°C to 0.3°C . Also, in Fig. 5(c), their makespans differ by 0 ms to 1 ms. Similarly, the estimated average system-level MTTF differ by 0.1%, 0.26%, 0.77%, and 0.44% respectively from 10 to 25 tasks as depicted in Fig. 5(b). But, as observed in Fig. 5(d), there is a significant difference in CPU times between the *G-APTI-TMC* and *G-APTI-FMC*. Thanks to our fast MTTF calculation methods, our approach (*G-APTI-FMC*) can achieve similar performance results (average peak temperature, average makespan, and overall average system-level MTTF) with the CPU times about 23 to 31 times lower than *G-APTI-TMC*.

In terms of peak temperature identification accuracy, if we compare the results by *G-APTI-TMC* with *M-STA* in Fig. 5(a), we can see that *M-STA* is quite pessimistic, i.e., by 15.3°C , 11.9°C , 13.2°C , and 18.5°C as the task numbers increase from 10 to 25. These results demonstrate that our proposed temperature identification methods, as shown in Algorithm 1, can greatly help *G-APTI-TMC* to obtain the solution with significantly reduced peak temperature and much improved system-level MTTF over *M-STA*, as much as 44.38%, 31.49%, 40.78%, and 64.61% as observed in Fig. 5(b).

From Fig. 5(d), we can also see that the computational cost for *M-TO* and *M-STA* increases rapidly due to the NP-hard nature of the mathematical programming technique.

TABLE V
RESULTS OF REAL-LIFE BENCHMARK

	M-TO	M-STA	G-APTI-FMC	G-APTI-TMC
Avg. Makespan (μs)	518	532	1167	1169
Avg. Sys. T_{peak} ($^{\circ}\text{C}$)	116.1	114.6	74.5	74.4
Avg. Sys. MTTF (<i>Hr</i>)	1336	1389	5005	5006
Avg. CPU time (s)	237.8	683.3	135.5	26001.2

TABLE VI
RESULTS OF ACC BENCHMARK

	M-TO	M-STA	G-APTI-FMC	G-APTI-TMC
Avg. Makespan (<i>ms</i>)	147	150	184	188
Avg. Sys. T_{peak} ($^{\circ}\text{C}$)	107.4	102.7	85.3	84.9
Avg. Sys. MTTF (<i>Hr</i>)	2219	2209	4020	4095
CPU time (s)	259.51	123.93	79.34	1370.5

Even so, their CPU times are still lower than that by *G-APTI-TMC*. Besides their ineffectiveness in dealing with temperature issues, the results clearly show that *M-TO* and *M-STA* cannot be used for large-scale systems when the task and ECU numbers continue to grow. In the meantime, the results also highlight the need to speed-up the system-wide MTTF calculation in design space exploration.

Similar findings are observed for the two automotive benchmarks. For a practical *Real-Life* benchmark, as shown in Table V, there is an average makespan difference of $2\mu\text{s}$, an average peak temperature difference of 0.1°C , and an MTTF difference of 1 Hr between *G-APTI-FMC* and *G-APTI-TMC*, but *G-APTI-FMC* can speed-up the system-level MTTF calculations by 191 times than *G-APTI-TMC*. On the other hand, *M-STA* is 40.1°C higher in peak temperature and 3616 Hr lower in system-level MTTF than *G-APTI-FMC*, which underlines the important contribution of the proposed *G-APTI-FMC* for the practical systems.

In the meantime, for the *ACC* benchmark, in Table VI, we can see that our proposed approach *G-APTI-FMC* can improve the computation efficiency of *G-APTI-TMC* by over 17 times with degradation in the peak temperature and system-level MTTF no more than 0.4°C and 75 Hr , respectively. Compared with *M-TO* (*M-STA*, resp.), *G-APTI-FMC* can improve the peak temperature and system-level MTTF with 22.1°C (17.4°C , resp.) and 1801 Hr (1811 Hr , resp.), respectively.

- b) *Verification of the proposed approaches for fast system-level MTTF calculation:* For single ECU or ECUs with similar degradation rates, as shown in Tables III and IV, our proposed fast MTTF calculation can be extremely efficient and effective. As shown in Table III, the proposed MTTF method can speed-up the computations about $12\times$ for synthetic test cases and $164\times$ for the Real-Life benchmark compared with the state-of-the-art method (23). In the meantime, from Table IV, we can see that the average error is less than 0.1% for both synthetic task cases and practical automotive benchmarks.

To consider the case when different ECUs may have different degradation rates, we propose four candidate

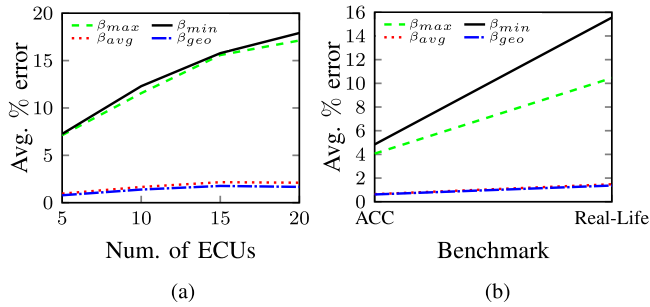


Fig. 6. Average error percentage. (a) Synthetic Test Cases. (b) Practical Benchmarks.

approaches, i.e., (33)-(36), to be used in (38). We tested these four approaches by mapping 100 tasks on 5, 10, 15, 20 ECUs in the synthesized test cases and our two practical benchmarks. We compare their performance in MTTF calculations by collecting the estimation errors normalized by the traditional one [16] (equation (23)). These results are denoted as β_{max} , β_{min} , β_{avg} , and β_{geo} , respectively, and shown in Fig. 6.

As shown in Fig. 6(a) and 6(b), we can see that the approach with β_{geo} has less than a 2% average error compared to the existing system-level MTTF equation (23) for both synthetic test cases and practical benchmarks. Further, we analyze the proposed methodology to compute the system-level MTTF with different degradation rates. We randomly generated the synthetic test cases for the configurations of 100-Tasks and 5, 10, 15, 20 ECUs and collected the 100 feasible test results, together with their peak temperatures, system-level MTTFs, makespans, and CPU times of the solutions, shown in Fig. 7.

In design space exploration, e.g., in a genetic algorithm or simulated annealing, evaluating the design objective function accurately and quickly is vital for the algorithm's effectiveness. To accurately evaluate the fitness of a design alternative helps to direct the search in the right direction, while the computation efficiency in fitness evaluation enables an extensive design space exploration. With the similar size of design space by *G-TMC-DDR* and *G-FMC-DDR-S*, as shown in Fig. 7(a), we observe that the average peak temperatures differ by 0.1 °C, 0.2 °C, 0.9 °C, 0.3 °C when ECU numbers increase from 5 to 20. Accordingly, their average system-level MTTFs differ by 0.8%, 0.6%, 1.9%, and 2.1%, as shown in Fig. 7(b) and their average makespans differ by 0 ms, 5 ms, 1 ms, 2 ms in Fig. 7(c). However, as observed in Fig. 7(d) (left Y-axis for the *G-TMC-DDR*, right Y-axis for the *G-FMC-DDR-S* and *G-FMC-DDR-L*), the CPU time of the *G-TMC-DDR* is 54, 95, 110, 108 times higher than that of *G-FMC-DDR-S*, respectively, as the ECU number increase from 5 to 20. Hence, with our fast MTTF calculation as formulated in (38), we can achieve a similar (a little inferior) performance in terms of the peak temperature, system-level MTTF, and makespan, but significant CPU time improvement.

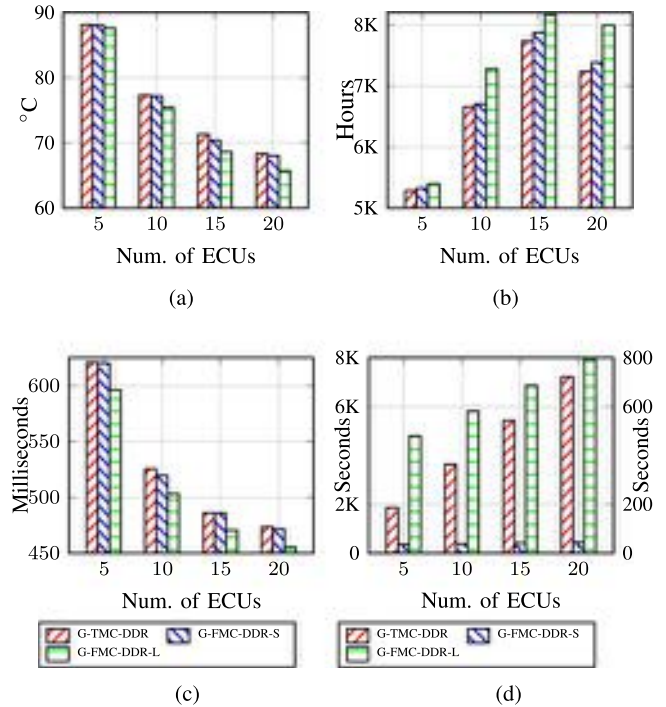


Fig. 7. Experimental results for 100-Tasks system. (a) Avg. system peak temperature. (b) Est. avg. system-level MTTF. (c) Avg. makespan. (d) Avg. CPU time.

The highly efficient MTTF calculation enables us to search a much larger solution space. In *G-FMC-DDR-L*, we increase both the population size and the reproduction generations in the genetic algorithm, with the total size of design space increased by 25 times. As observed in Fig. 7(d), with increased search space, the *G-FMC-DDR-L* approach requires more CPU time than *G-FMC-DDR-S*, but it helps to yield better results. For *G-FMC-DDR-L*, in Fig. 7(a), we see that the average peak temperature is reduced by 0.5 °C, 1.8 °C, 2.7 °C, 2.6 °C when ECU numbers increase from 5 to 20. Accordingly, their average system-level MTTFs improve by 113 Hr, 622 Hr, 434 Hr, 764 Hr as shown in Fig. 7(b), and Fig. 7(c), the average makespans reduced by 25 ms, 23 ms, 16 ms, 18 ms compared with *G-TMC-DDR*. In the meantime, it is worth mentioning that, by increasing the size of the initial population and number of generations for evolution in *G-FMC-DDR-L*, we can obtain solutions with better performance metrics (i.e., average peak temperature, average system-level MTTF, and average makespans). Simultaneously consuming much less CPU times, i.e., 3, 6, 7, 8 times less than the *G-TMC-DDR* according to Fig. 7(d). We can see similar results from the two practical automotive benchmarks. As shown in Table VII, comparing *G-TMC-DDR* and *G-FMC-DDR-S*, there is not much change in the average makespan. Their peak temperatures differ slightly by 0.33 °C, and average MTTF also differ slightly by 15 Hr. However, *G-FMC-DDR-S* is faster than *G-TMC-DDR* by 100 times. In the meantime, as

TABLE VII
RESULTS OF ACC BENCHMARK FOR DIFF DEGRADATION RATE

	G-TMC- DDR	G-FMC- DDR-S	G-FMC- DDR-L
Avg. Makespan ($m.s$)	188	188	180
Avg. Sys. T_{peak} ($^{\circ}C$)	89.69	90.02	88.34
Avg. Sys. MTTF (Hr)	4149	4164	4264
Avg. CPU time (s)	1567.9	15.58	178.92

TABLE VIII
RESULTS OF REAL-LIFE BENCHMARK FOR DIFF DEGRADATION RATE

	G-TMC- DDR	G-FMC- DDR-S	G-FMC- DDR-L
Avg. Makespan (μs)	1489	1490	1463
Avg. Sys. T_{peak} ($^{\circ}C$)	69.70	68.94	66.06
Avg. Sys. MTTF (Hr)	8173	8195	8867
Avg. CPU time (s)	32245.2	12.42	260.1

shown in Table VIII, the makespan, peak temperature, and MTTF estimation for *G-FMC-DDR-L* outperform *G-TMC-DDR* significantly, i.e., by 26 μs , 3.64 $^{\circ}C$, and 694 Hr , respectively, with CPU time 124 times lower. The prominence of our fast system-level MTTF calculation helps to achieve better design space exploration due to low timing complexity in the practical systems and offers better results.

VIII. CONCLUSION AND FUTURE WORK

In this paper, we study how to map a periodic automotive application on ECU with temperature issues taken into consideration. We first propose a mathematical programming approach to meet the peak temperature constraint while reducing the application latency. We further propose a more sophisticated genetic algorithm-based approach to deal with the peak temperature constraint and system-wide reliability optimization. To this end, we develop two key algorithms, i.e., guaranteeing the peak temperature for periodic tasks with variable execution times and the analytical approach for system-wide MTTF calculation, which can speed-up the process by several orders of magnitudes. Experimental results for two practical automotive benchmarks show that the proposed peak temperature algorithm is accurate by 17 $^{\circ}C$ and 40 $^{\circ}C$, which results in 81% and 260% more accurate MTTF estimation. At the same time, the proposed MTTF formulation is faster by 17 \times and 191 \times with an accuracy loss of less than 0.1%, which validates the efficiency of our proposed approach.

As the automotive industry is set to be transformed into autonomous cars, the temperature will play an increasingly critical role in the safety and reliability of automotive applications, especially as more advanced Artificial Intelligence (AI)/Deep Learning technologies are incorporated into autonomous driving systems. The future work of this research includes extending the thermal awareness to more advanced AI/Deep Learning applications and more advanced hardware architecture, such as those that incorporate GPU and dedicated NPU hardware units that can accommodate such applications.

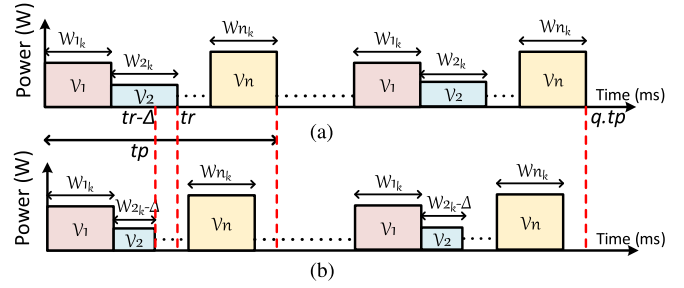


Fig. 8. (a) The WCET schedule with an ending temperature of $T_m(q \cdot t_p)$ in the stable status. (b) The schedule with a varied execution time with an ending temperature of $T(q \cdot t_p)$ in the stable status.

APPENDIX A PROOF OF LEMMA 1

Lemma 1: Let an ECU EC_k run n tasks $\mathcal{V}_1, \mathcal{V}_2, \dots, \mathcal{V}_n$ consecutively with a period of t_p . Let $T_m(q \cdot t_p)$ be the temperature at $t = q \cdot t_p$ when all tasks are executed with their WCETs. Then, $T_m(q \cdot t_p) \geq T(q \cdot t_p)$, if $T(q \cdot t_p)$ is the temperature at $t = q \cdot t_p$ when at least one task instance from $t \in [0, q \cdot t_p]$ runs with execution time less than its WCET.

Proof: Let's consider an arbitrary schedule with n tasks and period t_p as shown in Fig. 8. Let $T_m(q \cdot t_p)$ and $T(q \cdot t_p)$ be the temperature at $t = q \cdot t_p$ in Fig. 8 a and Fig. 8 b, respectively. In Fig. 8 a, all the tasks take their respective WCETs. In Fig. 8 b, the execution time of task \mathcal{V}_2 is reduced by Δ compared to its WCET. Therefore, the starting time of the remaining tasks in the scheduling sequence remains the same or is shifted to the left by Δ amount of time. From these two figures, we indeed find that both schedules are identical up-to-time instant $(t_r - \Delta)$, so as their temperature, $T(t_r - \Delta) = T'(t_r - \Delta)$. Let $T(t_r)$ and $T'(t_r)$ be the starting temperatures for the remaining part of the schedules, respectively. The schedule in Fig. 8 a has a higher accumulated computing time at $(q \cdot t_p)$ than the schedule in Fig. 8 b, and the workload in Fig. 8 b is moved away from $t = q \cdot t_p$. Based on *Lemma 3* of Schor *et al.* [59], we conclude that $T_m(q \cdot t_p) \geq T(q \cdot t_p)$. \square

APPENDIX B PROOF OF LEMMA 2

Lemma 2: Let an ECU EC_k run \mathcal{V}_i during an interval $t \in [t_1, t_2]$ and let EC_k 's temperature at t_1 be T_1 . Then, EC_k 's temperature monotonically increases (decreases *resp.*) within interval t , if $T_k^*(i) \geq T_1$ ($T_k^*(i) \leq T_1$ *resp.*).

Proof: From equation (3), we can infer that when EC_k reaches its stable state temperature, equation $T_k^*(i) = R_{th} \cdot P_{total}(i) + T_{ambk}$ holds, since $\frac{dT(t)}{dt} = 0$. From equation (3), we can also infer that

$$\begin{aligned} \frac{dT_1}{dt} &= \frac{-T_1 + R_{th} \cdot P_{total}(i) + T_{ambk}}{R_{th} \cdot C_{th}} \\ &= \frac{-T_1 + T_k^*(i)}{R_{th} \cdot C_{th}}. \end{aligned} \quad (41)$$

Since $T_k^*(i) \geq T_1$, we have $\frac{dT_1}{dt} \geq 0$. Then, EC_k 's temperature monotonically increases and vice-versa. \square

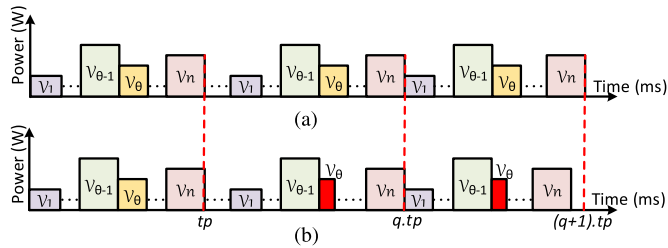


Fig. 9. (a) The WCET schedule. (b) The schedule with a varied execution time of tasks \mathcal{V}_θ in a stable state.

APPENDIX C PROOF OF THEOREM 3

Theorem 3: Let tasks $\{\mathcal{V}_1, \dots, \mathcal{V}_n\}$ be executed on ECU EC_k periodically with variable execution times. Then, the highest possible temperature during the execution is no more than T_m output from Algorithm 1.

Proof: Let a baseline schedule contain n tasks $\{\mathcal{V}_1, \dots, \mathcal{V}_n\}$ with period t_p and each task takes its WCET, as shown in Fig. 9 a. Without losing generality, we assume Fig. 9 b contains all the same intervals as Fig. 9 a, except task \mathcal{V}_θ whose execution time is reduced by Δ in the q -th period.

Let $T_W(t)$ and $T_A(t)$ be the temperatures at the instance of t for Fig. 9 a and Fig. 9 b, respectively. Based on Lemma 1, at $t = (q \cdot t_p)$, we can infer that the temperature of Fig. 9 b is no larger than Fig. 9 a, so we have

$$T_W(q \cdot t_p) \geq T_A(q \cdot t_p). \quad (42)$$

Now, consider a period from $t = (q \cdot t_p)$ to $t = ((q + 1) \cdot t_p)$. Starting from $t = (q \cdot t_p)$ let the temperature output based on Algorithm 1 be $\tilde{T}_W(i)$ after going through task \mathcal{V}_i in the loop (line 2-7) in Algorithm 1. Then, based on Lemma 2, $\tilde{T}_W(i)$ must be equal or higher than the temperature when completing \mathcal{V}_i with the schedule in Fig. 9 b. Meanwhile, the starting temperature at each interval considered in Algorithm 1 is no lower than that of $T_A(q \cdot t_p)$. Therefore, $\tilde{T}_W(i)$ monotonically increases to T_m in each valid interval as per Algorithm 1.

On the other hand, for the schedule in Fig. 9 b, if task \mathcal{V}_i has the stable state temperature lower than $T_A(q \cdot t_p)$, the temperature will be lowered. Therefore, we certainly find that at any time instant, the temperature predicted by Algorithm 1 (i.e., T_m) is higher than its periodic counterpart with varied execution times of the tasks. \square

ACKNOWLEDGMENT

The authors would like to thank anonymous reviewers for their comments and suggestions which contributed significantly to improving quality of this paper.

REFERENCES

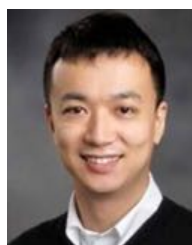
- [1] J. Scheuring and U. Bernhard, "Time-sensitive networking in the automotive industry," *ATZelectronics Worldwide*, vol. 15, no. 9, pp. 54–58, 2020.
- [2] *Road Vehicles-Functional Safety*, ISO Standard 26262, 2018.
- [3] ZVEI-German Electrical and Electronic Manufacturers' Association, "Handbook for robustness validation of automotive electrical/electronic modules", Frankfurt, Germany, Tech. Rep., 2013.
- [4] S. Hamdioui, D. Gizopoulos, G. Guido, M. Nicolaidis, A. Grasset, and P. Bonnot, "Reliability challenges of real-time systems in forthcoming technology nodes," in *Proc. IEEE Des., Automat. Test Europe Conf. Exhib., DATE*, 2013, pp. 129–134.
- [5] M. Krüger, S. Straube, A. Middendorf, D. Hahn, T. Dobs, and K. D. Lang, "Requirements for the application of ECUs in e-mobility originally qualified for gasoline cars," *Microelectronics Rel.*, vol. 64, pp. 140–144, 2016.
- [6] P. Mercati, F. Paterna, A. Bartolini, L. Benini, and T. Rosing, "WARM: Workload-aware reliability management in linux/android," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 36, no. 9, pp. 1557–1570, Sep. 2017.
- [7] S. Pagani, P. D. Manoj, A. Jantsch, and J. Henkel, "Machine learning for power, energy, and thermal management on multicore processors: A survey," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 39, no. 1, pp. 101–116, Jan. 2020.
- [8] H. Martin, G. Przemyslaw, W. Bernhard, and R. Sven, "Affordable and safe high performance vehicle computers with ultra-fast on-board ethernet for automated driving," *Adv. Microsystems Automot. Appl.*, pp. 56–68, 2019.
- [9] J. Korta, P. Skruch, and K. Holon, "Reliability of automotive multidomain controllers: Advancements in electronics cooling technologies," *IEEE Veh. Technol. Mag.*, vol. 16, no. 2, pp. 86–94, Jun. 2021.
- [10] W. Lu, P.-T. Huang, H.-M. Chen, and W. Huang, "An energy-efficient 3D cross-ring accelerator with 3D-SRAM cubes for hybrid deep neural networks," *IEEE J. Emerg. Sel. Topics Circuits Syst.*, vol. 11, no. 4, pp. 776–788, Dec. 2021.
- [11] K. Ueyoshi et al., "QUEST: Multi-purpose log-quantized DNN inference engine stacked on 96-MB 3-D SRAM using inductive coupling technology in 40-nm CMOS," *IEEE J. Solid-State Circuits*, vol. 54, no. 1, pp. 186–196, Jan. 2019.
- [12] D. Kim, J. Kung, S. Chai, S. Yalamanchili, and S. Mukhopadhyay, "Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory," *ACM SIGARCH Comput. Archit. News*, vol. 44, no. 3, pp. 380–392, Jun. 2016.
- [13] J.-H. Han, K. Torres-Castro, R. E. West, N. Swami, and M. Stan, "Thermal analysis of microfluidic cooling in processing-in-3D-stacked memory," in *Proc. IEEE 22nd Int. Conf. Thermal, Mech. Multi-Physics Simul. Experiments Microelectronics Microsystems*, EuroSimE, 2021, pp. 1–6.
- [14] S. Wang, Y. Yin, C. Hu, and P. Rezai, "3D integrated circuit cooling with microfluidics," *Micromachines*, vol. 9, no. 6, 2018, Art no. 287.
- [15] Y. Ma, T. Chantem, R. P. Dick, and X. S. Hu, "Improving system-level lifetime reliability of multicore soft real-time systems," *IEEE Trans. Very Large Scale Integration (VLSI) Syst.*, vol. 25, no. 6, pp. 1895–1905, Jun. 2017.
- [16] H. Lin, Y. Feng, and X. Qiang, "Lifetime reliability-aware task allocation and scheduling for MPSoC platforms," in *Proc. IEEE Des., Automat. Test Europe Conf. Exhib.*, 2009, pp. 51–56.
- [17] L. Niu, "Reliability-aware energy-efficient scheduling for (m,k)-Constrained real-time systems through shared time slots," *Microprocessors Microsystems*, vol. 77, 2020, Art. no. 103110.
- [18] G. Xie, G. Zeng, Y. Liu, J. Zhou, R. Li, and K. Li, "Fast functional safety verification for distributed automotive applications during early design phase," *IEEE Trans. Ind. Electron.*, vol. 65, no. 5, pp. 4378–4391, May 2018.
- [19] G. Xie et al., "Reliability enhancement toward functional safety goal assurance in energy-aware automotive cyber-physical systems," *IEEE Trans. Ind. Inform.*, vol. 14, no. 12, pp. 5447–5462, Dec. 2018.
- [20] A. Taherin, M. Salehi, and A. Ejlali, "Reliability-aware energy management in mixed-criticality systems," *IEEE Trans. Sustain. Comput.*, vol. 3, no. 3, pp. 195–208, Jul.–Sep. 2018.
- [21] M. A. Haque, H. Aydin, and D. Zhu, "On reliability management of energy-aware real-time systems through task replication," *IEEE Trans. Parallel Distrib. Syst.*, vol. 28, no. 3, pp. 813–825, Mar. 2017.
- [22] D. Zhu, "Reliability-aware dynamic energy management in dependable embedded real-time systems," *ACM Trans. Embedded Comput. Syst.*, vol. 10, no. 2, pp. 1–27, 2010.
- [23] J. Huang, R. Li, X. Jiao, Y. Jiang, and W. Chang, "Dynamic DAG scheduling on multiprocessor systems: Reliability, energy, and makespan," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 39, no. 11, pp. 3336–3347, Nov. 2020.
- [24] Y. Lee, H. S. Chwa, K. G. Shin, and S. Wang, "Thermal-aware resource management for embedded real-time systems," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 37, no. 11, pp. 2857–2868, Nov. 2018.

- [25] I. Ukhov, P. Eles, and Z. Peng, "Temperature-centric reliability analysis and optimization of electronic systems under process variation," *IEEE Trans. Very Large Scale Integration (VLSI) Syst.*, vol. 23, no. 11, pp. 2417–2430, Nov. 2015.
- [26] L. Schor, I. Bacivarov, H. Yang, and L. Thiele, "Worst-case temperature guarantees for real-time applications on multi-core systems," in *Proc. IEEE 18th Real-Time Embedded Technol. Appl. Symp.*, 2012, pp. 87–96.
- [27] A. L. Da Silva, A. L. Del Mestre Martins, and F. G. Moraes, "Fine-grain temperature monitoring for many-core systems," in *Proc. 32nd Symp. Integr. Circuits Syst. Des.*, 2019, pp. 1–6.
- [28] V. Chaturvedi, H. Huang, S. Ren, and G. Quan, "On the fundamentals of leakage aware real-time DVS scheduling for peak temperature minimization," *J. Syst. Architecture*, vol. 58, no. 10, pp. 387–397, 2012.
- [29] S. Sha, A. S. Bankar, X. Yang, W. Wen, and G. Quan, "On fundamental principles for thermal-aware design on periodic real-time multi-core systems," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 25, no. 2, pp. 1–23, Feb. 2020.
- [30] K. C. Chen, H. W. Tang, Y. H. Liao, and Y. C. Yang, "Temperature tracking and management with number-limited thermal sensors for thermal-aware NoC systems," *IEEE Sensors J.*, vol. 20, no. 21, pp. 13018–13028, Nov. 2020.
- [31] M. Sojka, O. Benedikt, Z. Hanzalek, and P. Zaykov, "Testbed for thermal and performance analysis in MPSoC systems," in *Proc. IEEE 15th Conf. Comput. Sci. Inf. Syst.*, FedCSIS, 2020, pp. 683–692.
- [32] J. Zhou et al., "Reliability and temperature constrained task scheduling for makespan minimization on heterogeneous multi-core platforms," *J. Syst. Softw.*, vol. 133, pp. 1–16, 2017.
- [33] K. Ergun et al., *RelloT: Reliability Simulator for IoT Networks*. W. Song, K. Lee, Z. Yan, L.-J. Zhang, and H. Chen, Eds., Cham, Switzerland: Springer International Publishing, 2020.
- [34] M. I. Bin Bandan, S. Bhattacharjee, S. K. Jali, and D. K. Pradhan, "Instantaneous mean-time-to-failure (MTTF) estimation for checkpoint interval computation at run time," *Microelectronics Rel.*, vol. 98, pp. 69–77, 2019.
- [35] K. N. Tu and A. M. Gusak, "A unified model of mean-time-to-failure for electromigration, thermomigration, and stress-migration based on entropy production," *J. Appl. Phys.*, vol. 126, no. 7, 2019, Art. no. 075109.
- [36] S. S. Sahoo, B. Veeravalli, and A. Kumar, "CL(R) early: An early-stage DSE methodology for cross-layer reliability-aware heterogeneous embedded systems," in *Proc. 57th ACM/IEEE Des. Automat. Conf.*, 2020, pp. 1–6.
- [37] H. Salamy, "Energy-aware schedules under chip reliability constraint for multi-processor systems-on-a-chip," *J. Circuits, Syst. Comput.*, vol. 29, no. 9, 2020, Art. no. 2050135.
- [38] G. Xie, K. Yang, H. Luo, R. Li, and S. Hu, "Reliability and confidentiality co-verification for parallel applications in distributed systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 6, pp. 1353–1368, Jun. 2021.
- [39] K.-J. Lee, J.-W. Kim, H.-J. Chang, and H.-S. Ahn, "Mixed harmonic runnable scheduling for automotive software on multi-core processors," *Int. J. Automot. Technol.*, vol. 19, pp. 323–330, 2018.
- [40] S. Bateni and C. Liu, "Predictable data-driven resource management: An implementation using autoware on autonomous platforms," in *Proc. IEEE Real-Time Syst. Symp.*, RTSS, 2019, pp. 339–352.
- [41] M. Li, J. Gao, L. Zhao, and X. Shen, "Adaptive computing scheduling for edge-assisted autonomous driving," *IEEE Trans. Veh. Technol.*, vol. 70, no. 6, pp. 5318–5331, Jun. 2021.
- [42] A. S. Bankar, S. Sha, V. Chaturvedi, and G. Quan, "Thermal aware lifetime reliability optimization for automotive distributed computing applications," in *Proc. IEEE 38th Int. Conf. Comput. Des.*, ICCD, Hartford, CT, USA, 2020, pp. 498–505.
- [43] G. Quan and V. Chaturvedi, "Feasibility analysis for temperature-constraint hard real-time periodic tasks," *IEEE Trans. Ind. Inform.*, vol. 6, no. 3, pp. 329–339, Aug. 2010.
- [44] Q. Han, M. Fan, O. Bai, S. Ren, and G. Quan, "Temperature-constrained feasibility analysis for multicore scheduling," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 35, no. 12, pp. 2082–2092, Mar. 2016.
- [45] S. Pagani, J. J. Chen, M. Shafique, and J. Henkel, "MatEx: Efficient transient and peak temperature computation for compact thermal models," in *Proc. IEEE Des., Automat. Test Europe Conf. Exhib.*, DATE, 2015, pp. 1515–1520.
- [46] J. R. Black, "Electromigration—A brief survey and some recent results," *IEEE Trans. Electron Devices*, vol. 16, no. 4, pp. 338–347, Apr. 1969.
- [47] A. Ait El Cadi, O. Souissi, R. Ben Atitallah, N. Belanger, and A. Artiba, "Mathematical programming models for scheduling in a CPU/FPGA architecture with heterogeneous communication delays," *J. Intell. Manuf.*, vol. 29, no. 3, pp. 629–640, 2018.
- [48] F. A. Omara and M. M. Arafa, "Genetic algorithms for task scheduling problem," *J. Parallel Distrib. Comput.*, vol. 70, no. 1, pp. 13–22, 2010.
- [49] A. Mutapcic, S. Boyd, S. Murali, D. Atienza, G. De Micheli, and R. Gupta, "Processor speed control with thermal constraints," *IEEE Trans. Circuits Syst. I: Regular Papers*, vol. 56, no. 9, pp. 1994–2008, Sep. 2009.
- [50] K. Skadron, K. Sankaranarayanan, S. Velusamy, D. Tarjan, M. R. Stan, and W. Huang, "Temperature-aware microarchitecture: Modeling and implementation," *ACM Trans. Architecture Code Optim.*, vol. 1, no. 1, pp. 94–125, 2004.
- [51] S. Sharifi, D. Krishnaswamy, and T. S. Rosing, "PROMETHEUS: A proactive method for thermal management of heterogeneous MPSoCs," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 32, no. 7, pp. 1110–1123, Jul. 2013.
- [52] R. Rao and S. Vrudhula, "Fast and accurate prediction of the steady-state throughput of multicore processors under thermal constraints," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 28, no. 10, pp. 1559–1572, Oct. 2009.
- [53] M. Faizan and A. S. Pillai, "Dynamic task allocation and scheduling for multicore electronics control unit (ECU)," in *Proc. IEEE 3rd Int. Conf. Electron., Commun. Aersp. Technol.*, ICECA, 2019, pp. 821–826.
- [54] B. Andersson, H. Kim, D. D. Niz, M. Klein, R. R. Rajkumar, and J. Lehoczy, "Schedulability analysis of tasks with corunner-dependent execution times," *ACM Trans. Embedded Comput. Syst.*, vol. 17, no. 3, May 2018, Art. no. 2050135.
- [55] S. K. Roy, R. Devaraj, A. Sarkar, S. Sinha, and K. Maji, "Optimal scheduling of precedence-constrained task graphs on heterogeneous distributed systems with shared buses," in *Proc. IEEE 22nd Int. Symp. Real-Time Distrib. Comput.*, ISORC, 2019, pp. 185–192.
- [56] F. Qi and C. P. Chen, "A complete monotonicity property of the gamma function," *J. Math. Anal. Appl.*, vol. 296, no. 2, pp. 603–607, 2004.
- [57] Y. Lee, E. Kim, and K. G. Shin, "Efficient thermoelectric cooling for mobile devices," in *Proc. IEEE/ACM Int. Symp. Low Power Electron. Des.*, ISLPED, 2017, pp. 1–6.
- [58] S. Speaks, "Reliability and MTBF overview" Vicor Reliability Engineering, Tech. Rep., 2014.
- [59] L. Schor, H. Yang, I. Bacivarov, and L. Thiele, "Worst-case temperature analysis for different resource models," *IET Circuits, Devices Syst.*, vol. 6, no. 5, pp. 297–307, 2012.



Ajinkya S. Bankar received the B.E. and M.E. degrees from Savitribai Phule Pune University, Baramati, India, in 2010 and 2013, respectively, and the Ph.D. degree from the Department of Electrical and Computer Engineering, Florida International University, Miami, FL, USA, in 2022. From 2013 to 2018, he was an Assistant Professor with the Vidya Pratishthan's Institute of Engineering and Technology, Savitribai Phule Pune University. He is currently a Data Scientist with Crowley, Jacksonville, FL, USA. His research interests include advanced

real-time computing system design, power/thermal aware design, sensitivity analysis of deep neural networks, and machine learning.



Shi Sha (Senior Member, IEEE) received the B.S. degree in electrical engineering from the Beijing University of Aeronautics and Astronautics, Beijing, China, the M.S. degree in telecommunications systems management from Murray State University, Murray, KY, USA, and the Ph.D. degree from the Department of Electrical and Computer Engineering, Florida International University, Miami, FL, USA. He is currently an Assistant Professor with the College of Business and Engineering, Wilkes University, Wilkes-Barre, PA, USA. His research inter-

ests include embedded machine learning, real-time scheduling, and high-performance/low-power computing.



Janki Bhimani (Member, IEEE) is an Assistant Professor at Florida International University, Miami. She received her Ph.D. and M.S. degrees in Computer Engineering from the Northeastern University, Boston. Her B.Tech. is from Gitam University, India in Electrical and Electronics Engineering. Her current research interests are performance modeling, resource management, and capacity planning for various emerging inter-disciplinary memory and storage research domains.



Gang Quan (Senior Member, IEEE) received and B.S. degree from the Department of Electronic Engineering, Tsinghua University, Beijing, China, the M.S. degree from the Chinese Academy of Sciences, Beijing, China, and the Ph.D. degree from the University of Notre Dame, Notre Dame, IN, USA. He is currently a Professor with the Department of Electrical and Computer Engineering, Florida International University. His research interests and expertise include real-time computing, smart IoT design, power/thermal-aware computing, advanced computer architecture, and cloud and reconfigurable computing.



Vivek Chaturvedi (Member, IEEE) received the M.S. degree from Syracuse University, Syracuse, NY, USA, and the Ph.D. from Florida International University, Miami, FL, USA. He is currently an Assistant Professor with the Department of Computer Science and Engineering, Indian Institute of Technology, Palakkad, Palakkad, India. He was a Research Scientist with the School of Computer Science and Engineering, Nanyang Technological University, Singapore. His current research interests include power and thermal optimization in multi/many core processors, including both 2D and 3D architectures. He is also actively working in the areas of security and reliability of embedded systems.

Automatic Stream Identification to Improve Flash Endurance in Data Centers

JANKI BHIMANI, Florida International University

ZHENGYU YANG and JINGPEI YANG, Samsung Semiconductor Inc.

ADNAN MARUF, Florida International University

NINGFANG MI, Northeastern University

RAJINIKANTH PANDURANGAN, CHANGHO CHOI, and VIJAY BALAKRISHNAN,
Samsung Semiconductor Inc.

The demand for high performance I/O in Storage-as-a-Service (SaaS) is increasing day by day. To address this demand, NAND Flash-based Solid-state Drives (SSDs) are commonly used in data centers as cache- or top-tiers in the storage rack ascribe to their superior performance compared to traditional hard disk drives (HDDs). Meanwhile, with the capital expenditure of SSDs declining and the storage capacity of SSDs increasing, all-flash data centers are evolving to serve cloud services better than SSD-HDD hybrid data centers. During this transition, the biggest challenge is how to reduce the Write Amplification Factor (WAF) as well as to improve the endurance of SSD since this device has a limited program/erase cycles. A specified case is that storing data with different lifetimes (i.e., I/O streams with similar temporal fetching patterns such as reaccess frequency) in one single SSD can cause high WAF, reduce the endurance, and downgrade the performance of SSDs. Motivated by this, *multi-stream* SSDs have been developed to enable data with a different lifetime to be stored in different SSD regions. The logic behind this is to reduce the internal movement of data—when garbage collection is triggered, there are high chances of having data blocks with either all the pages being invalid or valid. However, the limitation of this technology is that the system needs to manually assign the same *streamID* to data with a similar lifetime. Unfortunately, when data arrives, it is not known how important this data is and how long this data will stay unmodified. Moreover, according to our observation, with different definitions of a lifetime (i.e., different calculation formulas based on selected features previously exhibited by data, such as sequentiality, and frequency), *streamID* identification may have varying impacts on the final WAF of multi-stream SSDs. Thus, in this article, we first develop a portable and adaptable framework to study the impacts of different workload features and their combinations on write amplification. We then propose a feature-based stream identification approach, which automatically co-relates the measurable workload attributes (such as I/O size, I/O rate, and so on.) with high-level workload features (such as frequency, sequentiality, and so on.) and determines a right combination of workload features for assigning *streamIDs*. Finally, we develop an adaptable stream assignment technique to assign *streamID* for changing

This work was partially supported by National Science Foundation Awards CNS-2008324 and CNS-2008072, the National Science Foundation Career Award CNS-1452751, and the Samsung Semiconductor Inc. Research Grant.

Authors' addresses: J. Bhimani and A. Maruf, Florida International University, 11200 SW 8th St, CASE 238B, Miami, FL 33199; emails: jbhimani@fiu.edu, amaruf@fiu.edu; Z. Yang, 10100 Venice Boulevard Culver City, CA 90232; email: Yangzy1988@gmail.com; J. Yang, 1600 Amphitheatre Parkway, Mountain View, CA 94043; email: jingpei@google.com; N. Mi, 409 Dana Research Building, 360 Huntington Avenue, Boston, MA 02115; email: ningfang@ece.neu.edu; R. Pandurangan, Rajinikanth Pandurangan, 110 Holger Way, San Jose, CA 95134; email: rajinikanth.p@gmail.com; C. Choi, 3655 N First St, San Jose, CA 95134; email: changho.c@samsung.com.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2022 Association for Computing Machinery.

1553-3077/2022/04-ART17 \$15.00

<https://doi.org/10.1145/3470007>

workloads dynamically. Our evaluation results show that our automation approach of stream detection and separation can effectively reduce the WAF by using appropriate features for stream assignment with minimal implementation overhead.

CCS Concepts: • **Computer systems organization** → **Multiple instruction, multiple data**;

Additional Key Words and Phrases: Solid state drives, multi-streaming, write amplification factor, I/O stream detection, coherency, I/O workload characterization, NAND flash endurance

ACM Reference format:

Janki Bhimani, Zhengyu Yang, Jingpei Yang, Adnan Maruf, Ningfang Mi, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2022. Automatic Stream Identification to Improve Flash Endurance in Data Centers. *ACM Trans. Storage* 18, 2, Article 17 (April 2022), 29 pages.

<https://doi.org/10.1145/3470007>

1 INTRODUCTION

The emergence in I/O intensive applications requires more fast and more reliable storage pools. Flash-based storage devices such as **Solid-state Drives (SSDs)** thus becoming the mainstay due to their better performance compared to traditional **hard disk drives (HDDs)**, and as a result, today's computing devices widely adopt SSDs. Moreover, the decrease in NAND flash cost-per-gigabyte further accelerates the adoption of SSDs to support enterprise datacenters and virtualized cloud environments.

Although with many advantages of NAND flash technology, one major drawback lies in its limited number of *Write Cycles*. In SSD, there are three main operations, i.e., *Read*, *Program* (write) and *Erase* (delete). Among these operations, Read and Program can be performed at the page level. However, Erase can only be performed at a larger unit of erasure block level that consists of multiple pages. In order to reclaim free space to write new data, the SSD device needs to erase the whole block. The valid pages in the selected block are copied to a new block before erasing. This process of selecting the erase block and moving its valid pages before erasing is called **Garbage Collection (GC)**. Consequently, actual physical writes performed on the SSD device internally are more than the total logical writes performed by user applications running on the host. The ratio of the amount of data physically written to the NAND flash, including additional internal writes, to the amount of data logically written by user applications running on the host, is known as **Write Amplification Factor (WAF)**. The lower WAF indicates better endurance of an SSD device. This is because the flash cells in SSD have a fixed number of write cycles (also called as the **program and erase (PE)** cycles), and once the limit is reached, the corresponding cell is worn out and cannot be used anymore (i.e., died). Therefore, it is essential to have an efficient data placement on SSDs for reducing the WAF and enhancing the endurance of SSDs. There are numerous of technologies proposed to address this WAF issue, and one of them is multi-stream SSD. The motivation is to enable data with a different lifetime to be stored in different SSD regions so that the internal movement of data can be reduced.

Multi-stream SSDs: In data centers, the period data resides is often highly variable. Upon storing such data in SSDs, it often inevitably causes a large degree of data fragmentation due to data invalidation and then dramatically increases the WAF when garbage collection is triggered. To address this issue, the storage industry recently developed a new multi-streaming technology [23] that allows a host system to explicitly open different streams in SSD devices and allocate write requests to these streams according to the expected lifetime of data. Traditional SSDs have only one active append point where new data is written. Now, multi-stream technology enables the device to maintain more than one open erase block to append data writes in different physical locations of an SSD. In a multi-stream SSD [1], *streamIDs* are assigned to data according to their

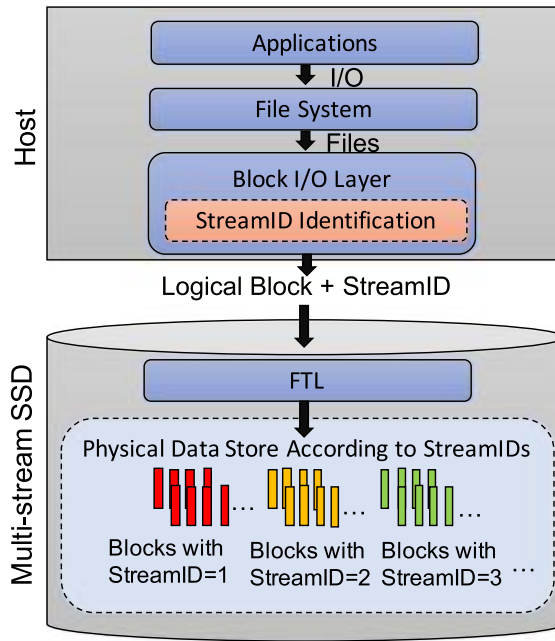


Fig. 1. Operation of Multi-stream SSDs with respect to I/O stack.

lifetime. The assignment of streamIDs can be done at any layer, such as the application layer, the file system layer, or the block layer. Figure 1 shows the I/O stack of a multi-stream SSD where streamID identification is made at the block layer. Once each logical block is assigned a streamID, a list of logical blocks with their corresponding streamIDs will be sent as an input to the **Flash Translation Layer (FTL)** in the multi-stream SSD device. The FTL then stores data blocks with the same streamID to the same physical blocks. This ensures that data with the same lifetime (i.e., the same streamID) can be invalidated together, which thus reduces the garbage collection overhead and results in low write amplification by avoiding extra internal data movements.

Challenges of Using Multi-Stream Technology: An efficient streamID assignment in multi-stream flash drives can reduce write amplification and improve the endurance of SSDs. To achieve the best possible benefit of multi-stream SSDs, it is critical to construct good streams, i.e., data with similar lifetime should be assigned the same streamIDs. However, it is challenging to predict the data lifetime. Although historical information of different features (such as frequency, sequentiality, and so on.) for past data accesses can be used to predict the expected lifetime of data, we found that stream identification using different features may have different impacts on the WAF of multi-stream SSDs. Moreover, feature sets that can accurately capture the expected lifetime of data may vary with different applications and workloads. Using a combination of not useful features cannot offer good performance improvement of multi-stream SSDs over no-streaming legacy SSDs. Therefore, a good streamID assignment technique to quantify the impact of different features and their combinations on the endurance of SSDs is essential.

Novel StreamID Assignment Approach: We extract various I/O workload features (e.g., frequency, adjacent access, and sequentiality) to study the impacts of these features and their combinations on write amplification of multi-stream SSDs. However, we observed that these features are not sufficient to capture the lifetime of data when using them individually. We thus propose a new feature, named *coherency*, to capture the friendship between logical blocks. Coherency can be more closely related to the lifetime of data. Unfortunately, by investigating all these different

features, we found that none of the features (e.g., frequency, adjacent access, sequentiality, and even coherency) can be claimed as the best for all I/O workloads, different features have varying impacts on WAF, and the benefit derived by using the combination of multiple features is not additive. Thus, another big challenge in developing this multi-stream framework is *how to determine a combination of workload features that are best for assigning appropriate streamIDs under a given I/O workload.*

To address this issue, we build an analytical *correlation model* to capture the co-relation between easily obtained workload attributes (such as I/O size, random write ratio, reuse ratio, and autocorrelation of write rates) with high-level workload features (such as frequency and coherency) and develop a novel **Feature-based I/O Stream identification (FIOS)** method to identify the best set (or combination) of features suitable to a workload for streamID assignment. FIOS can obtain a good feature combination automatically rather than experimenting with all possible combinations in a brute-force manner. We believe in the domain of multi-stream SSDs, this work is very important, as this is the first work that systematically measure, quantify, and understand the performance of multi-stream SSDs while running simultaneous instances of various containerized applications.

We modify the SSD module of DiskSim¹ [7] to simulate the multi-stream SSDs. First, we implement FIOS in modified DiskSim and then spend a lot of efforts and time towards designing Linux kernel Daemons for the real system implementation of OFIOS. We compare our performance with the state-of-the-art algorithms [39, 40, 42]. We consider the legacy SSDs that do not use multi-streaming technology as the baseline. Our experimental results show that our techniques can always identify a good combination of appropriate features to decide streamIDs and thus be able to improve the lifetime of SSD devices by reducing WAF.

Summarizing, this article provides the following major contributions

- Investigate the correlation between workload attributes (e.g., reuse ratio, write rate, and so on.) and high-level application features (e.g., coherency, sequentiality, and so on.), and analyze the impact of different features and their possible combinations on SSD **write amplification (WA)**.
- Propose a new feature, *coherency*, which represents the friendship between write operations with respect to their update time.
- Design a dynamic streamID detection scheme, which can determine a good combination of multiple features for different workloads.
- Develop and implement in Linux kernel, a parallel background learning mechanism that does not interfere with foreground stream identification.
- Design a novel auto-tuning module to dynamically identify and set the best feature combination while running simultaneous parallel instances of diverse containerized applications.

Section 2 presents the preliminary results to motivate this research. Section 3 presents the framework design of our proposed FIOS and OFIOS. Section 4 describes the architecture and data structures used and explains the system modifications required to deploy our framework. Section 5 describes the details of our platform setup to perform the evaluation. In Section 6, we evaluate FIOS and OFIOS. We discuss the implementation choices and overheads in Section 7. Section 8 presents related works. Lastly, we summarize our conclusions and future plan in Section 9.

2 MOTIVATION

The technology of multi-stream SSDs [1, 15] is newly invented to allow to explicitly write different data that are associated with each other or have a similar lifetime into a single stream. That

¹<https://github.com/benh/diskssim/tree/master/ssdmodel>.

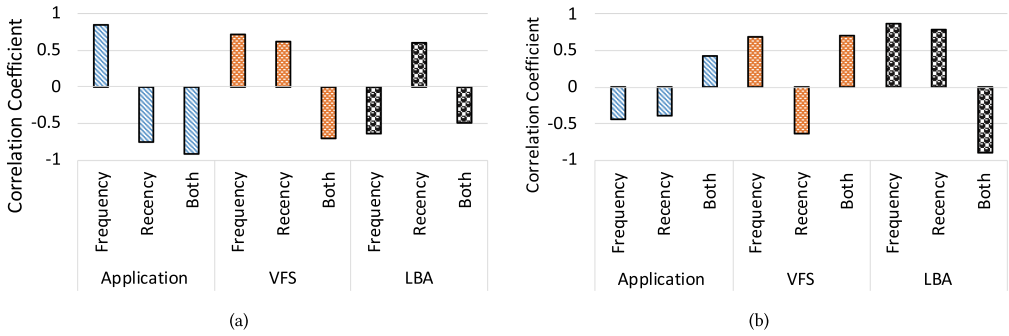


Fig. 2. Correlation Coefficient in RocksDB: (a) Standalone, (b) 16 Parallel Containers.

is, a group of data writes may be a part of a stream, and each stream is identified by a unique streamID that can be assigned either by the operating system or by the corresponding application. Identifying streamID that captures the lifetime of data is very important.

As discussed in the earlier works [39, 42], it is challenging to choose an appropriate layer of instrumentation to accurately identify streamID that captures the lifetime of pages on SSDs. A straightforward approach is to study the types of operations in the write path of an application and assign streamID based on file types [2, 15, 38]. For example, an application like Cassandra [31] mainly consists of three different types of operations in its write path—logging data to the commit log, flushing data from memtable to stable, and performing background compaction. Thus, depending upon such information from each application, we can modify applications like Cassandra [31] to assign a different streamID to commit logs, compressed metadata, stable indices, and stable data. This approach results in reducing WAF up to 65% [2]. However, first, *it is not adaptable as this requires studying and individually modifying each application to make them capable of assigning streamIDs*. Second, in data centers and virtualized cloud infrastructures where multiple applications run simultaneously, the combined characteristics of the write path may vary significantly compared to that of each individual application.

On the distributed and parallel platforms, the details instrumented at higher levels of the OS stack such as application layer, file system, or container data volume may not correlate well with the actual data layout on storage devices, especially when we have multiple instances of different applications running simultaneously in parallel using multiple storage devices. To further analyze the above hypothesis, we instrument characteristics at three different layers of I/O stack, (1) application—by grouping data using the application-level information of SSTable levels following the approach discussed in previous works such as VStream [37, 42], (2) **virtual file system (VFS)**—by grouping data according to the **program context (PC)** at VFS layer following PCStream [25], and (3) block layer—by grouping data according to the characteristics observed by analyzing logical block addresses as done in AutoStream [39]. We use the RocksDB database with a YCSB workload of 10,000 records. We operated standalone instances and 16 simultaneous instances, each running in a separate docker container. We choose two features that are most commonly used in all the prior works [25, 39, 42] as frequency also known as temperature and recency. We rank the importance of the data according to frequency, recency, and both combined. Figure 2, shows the **Pearson correlation coefficient (PCC)** [9] of features with the data pages lifetime on SSD. The closer the value is to 1 or -1 , the more relevant it is to the data page lifetime. For this experiment, we compute the lifetime of the data pages by performing post hoc analysis on the data placements while replaying workload traces on the DiskSim simulator.

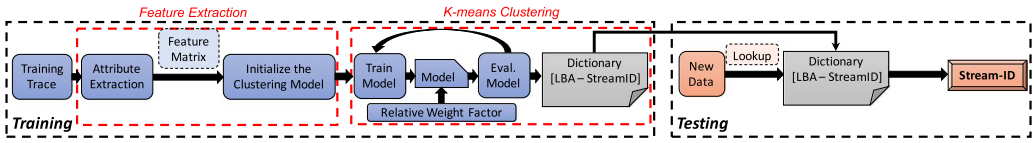


Fig. 3. Block diagram of our FIOS framework.

	Feature_1	Feature_2	Feature_j	Feature_m	
1	Sector_chunk_1	0	1	1	0
2	Sector_chunk_2	1	0	0	0
⋮					
i	Sector_chunk_i	0	1	0	1
⋮					
n	Sector_chunk_n	1	0	0	1

Fig. 4. Feature matrix data structure.

From Figure 2, we see that for standalone deployment, the application level data co-relates strongly with the data lifetime. However, while multiple instances of applications are running in parallel, then the data instrumented at lower levels such as block layer strongly co-relates with the data lifetime. This is because, while running parallel application containers, the farther layer of instrumentation and decision making means higher chances of dependencies on design choices used for the intermediate layer, which thus reduces the co-relation of the observations from instrumented data to the lifetime of data stored on SSDs. As a result, the prior techniques such as “VStream,” “Optimizing NoSQL DB on flash,” “PCStream,” and “AutoStream,” that are tightly coupled with higher layers of the host to identify streamIDs, are not much useful in heterogeneous parallel platforms. From Figure 2, we also see that the correlation to the data lifetime while using information from multiple features can be higher than just using individual features. Thus, motivated by the two major abovementioned observations, we propose a new feature-based approach that strives to determine and assign streamID at block layer, based on the best combination of multiple features.

3 FRAMEWORK DESIGN

In this section, we explain the design and the main components of our framework.

3.1 FIOS Design

The block diagram of our FIOS technique is shown in Figure 3, which consists of two main phases: *training* and *testing*. The training phase is the pre-processing step for streamID detection, which needs to be performed only once. The testing phase represents the actual runtime phase of applications, during which streamID assignment is performed. Training and testing phases are both performed in a cyclic pattern to capture the runtime workload changes. To better understand how FIOS operates, we give an example of single cycle of these two phases. FIOS first uses `blktrace` and `blkparse` commands to obtain a real I/O block trace from the application platform as a training trace. This trace is then used by the training phase to extract the required features such as frequency, adjacent access, sequentiality, and coherency. Later, we describe the details of how each feature is extracted from an I/O trace in Section 3.2.

The captured features are enclosed in the form of a feature matrix, such as the one shown in Figure 4. Accordingly, the feature matrix consists of $n \times m$ cells, where m is the number of features analyzed for streamID detection, and n is the total number of sector_chunks in the storage

volume. The storage volume may include a single SSD or multiple SSDs. Each sector_chunk comprises of several sectors. The sector_chunks in rows are arranged in an incremental numerical order such that the product of row number and each sector_chunk's size gives the sector_chunk address. Each cell in the feature matrix gives data quanta, which reflects the importance of the i th sector_chunk with respect to the j th feature. Notice that if a sector_chunk consists of only a single block address, then that sector_chunk would be the same as a **logical block address (LBA)**. Thus, we use sector_chunk and logical block address interchangeably in this article.

Through feature extraction, FIOS creates a feature matrix, which is then used for clustering write/update sector_chunks into different streams. To obtain high computational efficiency, we improved a multi-threaded K-means clustering algorithm [10] to cluster sector_chunks into K streams in parallel. In particular, each feature makes one dimension of clustering inputs, and a relative weight factor can be used as an optional input to emphasize the relative importance of each feature in deciding streamIDs. By default, all features are considered to be equally important with the same weights. The number of clusters (i.e., K) of the K-means algorithm maps to the number of streams supported by the SSD drive (e.g., 16 streams in the latest SSD drives). FIOS uses the K-means algorithm to group all data points in the feature matrix into the given number of streams. The clustering results are stored in the LBA-StreamID dictionary that consists of the pairs of sector_chunks and streamIDs.

In the testing phase, we have actual I/O operations (e.g., writes/updates) performed on storage devices. For a read, the operation of legacy and multi-stream SSDs remains the same. For a write or an update, a multi-stream SSD allows multiple append points. Thus, to decide data placement on a multi-stream SSD, FIOS assigns a streamID to each sector_chunk through a quick lookup in the LBA-StreamID dictionary. Thereafter, the assigned streamID is penetrated through the I/O stack until the data is actually written to the physical address space of that streamID. For an erase, both legacy and multi-stream SSDs work in the same way by searching all finalized blocks for a GC candidate and then copying out valid pages from the candidate.

3.2 Feature Extraction

Now, we turn to present how FIOS extracts workload features from the collected I/O traces and completes a feature matrix. As introduced above, the feature matrix comprises of the importance factor for each sector_chunk with respect to different features. In order to keep low instrumentation overhead, we decide to represent the importance factor for a sector_chunk as a binary datum. Each entry (i,j) in the feature matrix can then be represented by a single binary bit (e.g., 1 or 0) that indicates whether j th feature is considered to be important ("1") or not ("0"). We use a vector $\vec{\delta}$, to contain the thresholds for each feature, as criteria to determine the values (i.e., 0 or 1) for each entry.

For instance, the feature of frequency indicates how often a particular sector_chunk is accessed. If the number of accesses of that sector_chunk is greater than the predefined threshold (e.g., $\delta[\text{frequency}] = 4$ times), then the frequency entry of that particular address is set as 1, otherwise 0. We set $\delta[\text{frequency}]$ to the median value of the frequencies of sector_chunks, assuming that the frequencies of sector_chunks follow a Gaussian distribution. The feature of the adjacent access indicates that sector_chunks that are adjacently accessed during a time window are more likely to be accessed together again. We set $\delta[\text{adjacent_access}]$ to construct multiple groups according to the access time of sector_chunks. We can also have the feature of sequentiality to capture if an incoming I/O access is sequential to the previous one. We calibrate the threshold vector using an ensemble of piecewise linear regression models [33].

	Feature_1	Feature_2	Feature_j	Feature_m	
1	Sector_chunk_1	0.3	0.6	0	0.3
2	Sector_chunk_2	0.9	1	0.7	0.6
⋮					
i	Sector_chunk_i	1	0.1	0.4	0.2
⋮					
n	Sector_chunk_n	0.5	0.2	1	0.8

Fig. 5. Decimal fraction feature matrix.

3.3 Binary to Decimal Extension of Algorithm

As explained above, one of the limitations of FIOS is that it is a binary model. The binary version of FIOS only considers a single binary bit as an importance factor for each feature. This may easily become a limitation to capture complex workloads. For simple features like sequentiality, it is straightforward that any sector_chunks can either be sequential or not. However, for more advanced features like coherency, a particular sector_chunk may be more coherent with some groups than others. In order to capture such various levels of similarities, we extend FIOS to consider the decimal feature matrix instead of binary. The decimal extension gives a better resolution to the feature matrix. We choose decimal over hexadecimal just because the number with base ten is well understood and can be easily represented as percentages. It is important to note that the number of partitions to separate data into different streams for multi-stream SSD is not limited to 2 for binary and 10 for the decimal extension. But, being independent on the resolution of the feature matrix, FIOS as explained in Section 3.1 can be used to construct a dictionary with any number of streams supported by multi-stream SSDs.

Mainly, we modify our feature extraction technique to label all sector_chunks with a decimal between 0 and 1 that indicates internal grouping with respect to each feature. Figure 5 shows an example of the feature matrix with decimal fractions. Particularly, for features, such as frequency and adjacent_access, we set the threshold (δ) as a stepwise function. This stepwise function partitions the sector_chunks into ten different categories. The feature of sequentiality is straightforward and binary in nature, so it remains the same. Finally, while extracting coherency, we maintain a counter to obtain the number of friendly groups of each sector chunk. Then, we perform linear range shifting to obtain a value of coherency feature between 0 and 1 for all sector_chunks.

Once the decimal feature matrix is generated, the K-means algorithm is run to cluster features to build a dictionary. After that, during the runtime for the testing phase, the operation remains the same as in the case of a binary algorithm. It is interesting to analyze the endurance gain and additional overhead introduced by generating such a decimal feature matrix when compared to its binary counterpart. We next present our evaluation using the decimal feature matrix. We integrate the binary and decimal feature matrix as a plugin with our FIOS framework that allows user you change easily upon restart. Also, note that changing feature extraction from binary to decimal and vice-versa upon reboot will not impend any operation on existing data as the only data we persist from prior mode is Dictionary and its format remains the same for both binary and decimal feature extraction.

3.4 Coherency

Unfortunately, we observe that existing well-known features like frequency and sequentiality are not sufficient to capture the lifetime of data when using them individually. We thus propose a new feature, named *Coherency*, to capture the friendship between sector_chunks. Coherency can be more closely related to the lifetime of data. We refer to two addresses (i.e., sector_chunks) as friends if we observe that they are mostly updated together in multiple time windows. Intuitively,

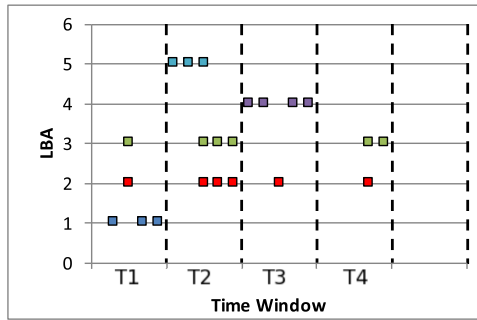


Fig. 6. A sampling graph for describing a novel feature of coherency.

grouping coherent addresses in the same stream can be beneficial because all sector_chunks in that stream will have a similar update pattern.

Figure 6 illustrates the general idea of the coherency feature, where the x -axis corresponds to time windows, and the y -axis corresponds to LBAs or sector_chunks. If a particular group of LBAs are mostly updated together in multiple time windows, then these LBAs may be referred to as being coherent with each other. For example, if we snoop at a specific time interval, which is shown by a vertical dashed line in Figure 6, we can say that LBA2 and LBA3 are coherent because they are concurrently updated in three of the four-time windows, i.e., T1, T2, and T4. This indicates that these two sector_chunks (i.e., LBA2 and LBA3) have a tendency to be updated at the same time, such that grouping them together into a single stream can help to reduce WAF.

In particular, we maintain lists of unique sector_chunks that have been accessed in each time window. We compare the list of each time window to identify the common sector_chunks that appear in at least two lists. We then mark these common sector_chunks as coherent (i.e., 1). The process is finished when all unique sector_chunks of every time window are allocated their coherency values. In our current model, because we consider the datum of our feature matrix as either 0 or 1, it is a limitation that we cannot differentiate different groups of friends while capturing coherency. As of now, our model only partitions sector_chunks into two groups, i.e., one consisting of LBAs which are “friendly to somebody” and the other consisting of LBAs, which are “friendly to nobody.”

Additionally, as many of our workloads and file systems are append-only, it is interesting to understand how coherency can be useful for them. The SSD address space is fixed depending upon the capacity of the SSD. So, eventually, after SSD reaches its steady-state, even if the workload is append-only, physical pages in that SSD that stores “friendly” LBAs according to our feature of coherency will be invalidated at nearly the same time. It is also important to clarify that even though an append-only application maintains a never-ending log, this log is eventually mapped to the finite LBA space. Updates to a set of “friendly” LBAs are not necessarily caused by the updates of the same application data. Although we do not perform in-place updates for append-only workloads, but rather perform compaction such as for levels in RockDB. We want to claim that because of data mappings done at the intermediate operating system layers (such as file system) and SSD-related management policies (e.g., FTL mapping, wear-leveling, garbage collection), a group of sector chunks can be recurrently accessed together. The coherency feature we consider and instrument at the block layer aims at capturing the relationship of these sector chunks.

Regarding the bursty and not heavy writes, there are some different scenarios. First, if there are very sparse and eventual writes that have arriving intervals considerably longer than the time interval we compute coherency, then we label these sparse writes as “coherent to nobody”. Second,

Table 1. Summary of the Relation between Features and Workload Characteristics

Feature	ACF	RW	R	S	λ
Adjacent Access (A)	High	-	-	-	High
Coherency (C)	High	High	-	-	-
Sequentiality (S)	-	-	-	High	-
Frequency (F)	-	-	High	-	-

(ACF - Auto-Correlation of Inter-arrival Time, RW -Random Write Ratio, R - Reuse Ratio, S - Estimated Size of each Writes in KB, λ - Write Rate in GB/Day).

if these writes are sparse but have intervals shorter than the time interval we compute coherency, then we mark these sector chunks as coherent because they are updated together even though their traffic is not heavy. On the other hand, if writes are bursty but these simultaneously accessed sector chunks are not re-accessed together later, then the proportion of coherent sector chunks will remain small.

3.5 Combination of Multiple Features

As we discussed, an I/O workload has different features, such as frequency, coherency, and so on. How to use one or multiple features to cluster data points into the desired number of streams is not trivial. We found that using multiple features might give better performance than using a single feature. However, we also found that using unsuitable features to assign streamIDs may not be able to help to improve the performance and even may cause performance degradation. Thus, a critical issue is how to find out an optimal combination of features that can enable FIOS to achieve the high quality of streamID packetization with the minimum overhead. Given n features, we can have $2^n - 1$ possible combinations. We investigate and evaluate the impacts of these feature combinations on the write amplification of multi-stream SSDs in Section 6.1.

In summary, our results indicate that (1) none of the features (such as frequency, adjacent access, sequentiality, and coherency) can be claimed as the best for all I/O workloads, (2) different features have varying impacts on WAF, and (3) the benefit derived by using the combination of multiple features is not additive. Therefore, a big challenge in developing this multi-stream framework is *how to determine a combination of workload features that are best for assigning appropriate streamIDs under a given I/O workload*. Investigating the results for many workloads, we further propose a new approach to determine a good feature combination.

3.6 Automatic Feature Selection

To address the above issue, we build an analytical *correlation model* to capture the co-relation between easily obtained workload characteristics, such as I/O **size (S)**, **random write ratio (RW)**, **reuse ratio (R)**, and **autocorrelation (ACF)** of write rates) with high-level workload features (such as frequency and coherency) and identify a good set (or combination) of features suitable to a workload for streamID identification. Our goal is to obtain such a good feature combination automatically, rather than experimenting with all possible combinations.

The initial step in our approach is to determine the workload characteristics that can mainly affect the selection of features. Table 1 summarizes the implications that we develop regarding the relationship between workload characteristics and features used for streamID identification. For example, as shown in the first row of the Table 1, if a workload has high ACF and high λ , then **adjacent access (A)** is one of the good features to select.

ACF	RW	R	S	λ	
1	0	0	0	1	→ A
1	1	0	0	0	→ C
0	0	0	1	0	→ S
1	0	0	1	0	
0	0	1	0	0	→ F
1	0	1	0	0	

Fig. 7. Bit mapping log of base matrix \mathbb{B} .

Based on the information in Table 1, we construct a base matrix \mathbb{B} as shown in Figure 7, where “1” corresponds to the “High” impact in Table 1 and “0” corresponds to the “Low” impact. In \mathbb{B} , some features (e.g., **sequentiality (S)** and **frequency (F)**) need to have 2 rows each. For example, no matter ACF of inter-arrival time is high or low, the feature of frequency is a good choice as long as the R is high. If we can map a workload’s characteristics to a row in the base matrix, then we can choose the corresponding feature. However, we notice that the characteristics of a workload may not exactly map to one of the 6 possible rows of base matrix \mathbb{B} , and thus it is not straightforward to determine the best combination of features. Given this, we have the following problem objective and solutions.

Objective: Assume a map function f_{map} which determines the best possible combination of features ($\vec{\psi} \rightarrow F, A, S, C$) based on the workload characteristics ($\vec{\theta} \rightarrow ACF, RW, R, S, \lambda$), i.e.,

$$\vec{\psi} = f_{map}(\vec{\theta}). \quad (1)$$

Thus, our objective is to determine the above defined function f_{map} .

Solution: The attribute vector $\vec{\theta}$ is constructed based on users input of workload characteristics, where an attribute is assigned 1 if the given workload exhibits such a characteristic. For instance, after analyzing a given workload’s characteristics (consider *MSR-prn0*), we can get its attribute vector as $\vec{\theta}$ as [1 1 0 0 1]. It represents that *MSR-prn0* has high *ACF*, *RW*, and λ , and low *R* and *S*. If the attribute vector $\vec{\theta}$ is given by the user has “0” for a particular attribute, then any feature that has a high impact of this attribute is definitely not a good candidate. For example, for the above considered $\vec{\theta}$, attributes such as *R* and *I/O S* are “0”. Thus, the features of sequentiality and frequency which have a high impact on *I/O size* and *reuse ratio* (as seen from Figure 7) are not good candidates. We construct the factorization vector $\vec{\alpha}$ that indicates the above information of whether or not each row in base matrix \mathbb{B} is a good candidate for given $\vec{\theta}$. Such a factorization vector $\vec{\alpha}$ can be represented as below,

$$\vec{\alpha} = [\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5, \alpha_6]^T, \quad (2)$$

where $\alpha_i \in \{0, 1\}$. If α_i is 0, then the i^{th} row of base matrix \mathbb{B} is not a good candidate and vice-verse. Moreover, as atleast one of the features must be selected, we have $\sum_{i=1}^6 \alpha_i > 0$. Thus, we obtain α by the following elimination on base (\mathbb{B}) with respect to careful examination of input attribute vector $\vec{\theta}$. This elimination is represented by a factorization function f_{fact} as,

$$\vec{\alpha} = f_{fact}(\vec{\theta}). \quad (3)$$

Finally, the function f_{map} can be expressed by some other function f_{deriv}^{-1} . Function f_{deriv}^{-1} is just the replacement of f_{map} such that $f_{map}(\vec{\theta}) = f_{deriv}^{-1}(\vec{\theta}) \quad \forall \vec{\theta}$. This is done to make the rest of the process easier. Further, $f_{deriv}^{-1}(\vec{\theta})$ gives all possible combinations of the rows of base

ALGORITHM 1: Workload_Run_Daemon (WRD)

Parameter: $T_{RunWindow}, T_{TrainWindow}$
Initialize : $TimeStamp = 0,$
 $Workload_Set[W_1, W_2, \dots, W_n],$
 $curr \rightarrow BD = NULL,$
 $curr \rightarrow SID = NULL,$
 $\vec{W}_{\theta} = [W_{1\theta}, W_{2\theta}, \dots, W_{n\theta}],$
 $\vec{W}_{\psi} = f_{mapFeat}(\vec{W}_{\theta})$

Input: Sector_Chunk
Result: StreamID

- 1: **while** $TimeStamp < Workload_Time$ **do**
- 2: **while** $curr \rightarrow time \bmod T_{RunWindow} == 0$ **do**
- 3: $start_time \leftarrow curr \rightarrow time$
- 4: $curr \rightarrow W \in \overrightarrow{Workload_Set}$
- 5: set $W_{\theta} \leftarrow \vec{W}_{\theta}[curr \rightarrow W]$
- 6: set $W_{\psi} \leftarrow f_{mapFeat}(W_{\theta})$
- 7: **while** $0 \leq \frac{(T_{TrainWindow} - start_time)}{T_{TrainWindow}} \leq 1$ **do**
- 8: $SID \leftarrow WTD(W_{\psi}, T_{TrainWindow})$
- 9: **if** $SID \neq \emptyset$ **then**
- 10: $StreamID \leftarrow hash_get(SID(Sector_Chunk))$
- 11: **else**
- 12: $curr \rightarrow time = TimeStamp + +$

B for which elements in $\vec{\alpha}$ is equal to 1, i.e., $(\vec{\theta} = f_{deriv}(\bigvee_{i=0}^6 (\vec{\alpha}_i \times B_i)))$. We solve equation $\vec{\theta} = f_{deriv}(\bigvee_{i=0}^6 (\vec{\alpha}_i \times B_i))$ for the only unknown “ f_{deriv} ” in it. Recall that the attribute vector $\vec{\theta}$ is constructed based on users input of workload characteristics, where an attribute is assigned 1 if the given workload exhibits such a characteristic. We construct the factorization vector $\vec{\alpha}$ that indicates the above information of whether or not each row in base matrix B (i.e., shown in Figure 7) is a good candidate for given $\vec{\theta}$. Finally, in the last step just by computing the inverse of this function f_{deriv} , we can achieve our objective of finding ψ as $f_{deriv}^{-1}(\vec{\theta}) = f_{map}(\vec{\theta}) = \vec{\psi}$.

3.7 OFIOS: Online FIOS

The workloads being executed in data centers change over time. Also, in a virtual machine and containerized environment, the number of active instances of the workloads varies. In order to predict the best feature combinations and maintain a dictionary that automatically adapts to the changing workloads, we design and develop the **Online FIOS (OFIOS)** technique. To dynamically adapt the dictionary constructed using FIOS, we run two parallel bunches of processes. One bunch of processes undertake the operations of FIOS as explained in Sections 3, 4, and 3.3. To manage these processes, we design a new **Workload_Run_Daemon (WRD)** and implement it using `systemd` [3]. Algorithm 1 describes the runtime process of OFIOS. Upon a write/update to any particular sector_chunk, WRD performs a lookup in the **Simplified Index Dictionary (SID)** as explained in Section 4 and returns the StreamID to place that data on the flash-based storage. To initialize, we capture the characteristics (θ) of different workloads and multiple instances of parallel workloads (i.e., W_1, W_2, \dots, W_n) into the feature matrix ψ as described in Sections 3 and 3.3. During the workload execution, within the current time window $T_{RunWindow}$, the latest patch of SID with respect

ALGORITHM 2: Workload_Training_Daemon (WTD)

Initialize: $TimeStamp = 0$
Input: $W_{\psi}^{\rightarrow}, T_{TraceWindow}, T_{TrainWindow}$
Result: SID

- 1: **while** $TimeStamp < T_{TraceWindow}$ **do**
- 2: $label_blk \leftarrow random_sampling[/dev/disk]$
- 3: $trace_exe \leftarrow blktrace[label_blk]$
- 4: $set\ TrainTrace \leftarrow blkparse[trace_exe]$
- 5: $TimeStamp ++$
- 6: **while** $TimeStamp < T_{TrainWindow}$ **do**
- 7: $set\ BD \leftarrow Run\ FIOS\ on\ TrainTrace\ with\ feature\ W_{\psi}^{\rightarrow}$
- 8: $hashset\ SID \leftarrow Reduction(BD)$
- 9: Update $curr \rightarrow BD = BD$
- 10: Update $curr \rightarrow SID = SID$

to that of the last time window is pulled from the **Workload_Training_Daemon (WTD)** (WTD is explained next). Finally, OFIOS looks up SID for the StreamID of any particular sector chunk.

The second set of processes operates simultaneously in the background during the operation of WRD by the first set of processes. The processes in this second set use non-blocking I/Os [13] to construct a new dictionary. This dictionary under construction ensures to incorporate the recent and ongoing changes in the workloads and their active instances. Our WTD manages the second set of processes. Algorithm 2 describes the operations of WTD. For each time window ($T_{TrainWindow}$), WTD takes the input of the feature matrix (W_{ψ}^{\rightarrow}) generated in the last window and produces a new SID. In the first $T_{TrainWindow}$ time period of each window which is $T_{TraceWindow}$ long, we do the random sampling of LBAs from the storage address space. Then, we run blktrace and blkparse to instrument the characteristics of the running workload as $TrainTrace$. Finally, we run FIOS with $TrainTrace$ and W_{ψ}^{\rightarrow} to update the dictionary, which is used to assign streamID in the WRD launched after that. We evaluate OFIOS and compare our performance with the state-of-the-art existing techniques in Section 6.4.

4 MAIN ARCHITECTURE

In this section, we present the I/O stack architecture overview of our FIOS implementation and introduce the basic data structures used in the implementation. Following that, we discuss the modifications on an existing SSD simulator, i.e., Disksim [7] to enable the multi-stream interface.

4.1 FIOS: Architecture Overview

Our prototype of FIOS can be implemented at any levels, such as the file system layer on the host side or the FTL layer inside the device. In our implementation, we choose to deploy our prototype at the block layer. Later in Section 7, we will discuss the benefits of this design choice. Figure 8 shows the I/O stack of FIOS architecture, which consists of two main components to assist in streamID assignment, i.e., (1) **Base Dictionary (BD)** and (2) SID. As shown in Figure 8, BD is persisted in flash memory and SID is stored within the memory management subsystem in DRAM. On a system failure like a power outage, SID can be rebuilt from BD on rebooting. At the software level, host applications may run directly in the system, or containers like Docker and LxC. The application layer performs read and write I/Os, which are passed to the underlying file system and flash memory through system calls. For every I/O write or update, a lookup operation is performed to SID by calling `hash_get` to get streamID for the corresponding block addresses. The

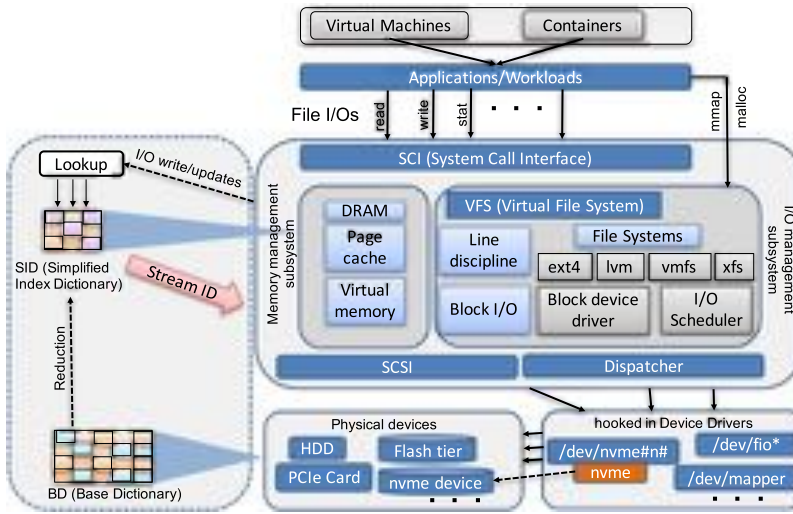


Fig. 8. The I/O stack of FIOS architecture.

obtained streamID is piggybacked on a reserved field of regular and queued write commands as specified in the ATA command set. Along with every write operation, the corresponding streamID is penetrated through all the below layers until that I/O is written or updated on the SSD.

4.2 Basic Data Structures

Base Dictionary: The BD is the result of the training phase (see Figure 3) of our FIOS, which contains the mapping between streamIDs and sector_chunks. Recall that there are 64 sectors in one sector_chunk. Each sector_chunk has a streamID, as metadata associated with it, which is stored in BD on flash memory. Such a metadata uses 0.5 bytes to keep the streamID. In our implementation, we consider a logical volume of flash of a 3 TB, which is stripped over the physical volume of three 1 TB SSDs. Thus, for 3 TB flash volume, we require less than 50 MB in total to store all streamID metadata, which is only 0.001% of total flash disk space.

Simplified Index Dictionary: The SID is used to perform a lookup for streamIDs at run time (i.e., the testing phase). SID is the compressed version of BD, where we reduce the size of base dictionary by combining all consecutive sector_chunks that have the same streamID and replacing multiple lines with one line. Following that, instead of sector_chunk number, we have the range of sector chunks for each row in SID. We can then store SID in DRAM memory for fast hash_get lookup. We observe that SID is 50% more efficient in terms of space when compared to BD. Once SID is created and stored, we can move BD to the back-end storage until we have a new dictionary from another round of the training phase. In our experiments, we have 128 GB DRAM in the server. The SID footprint for different workloads consumes less than 25 MB of main memory. Therefore, the SID footprint overhead is only about 0.02% of the size of the main memory.

4.3 Multi-stream SSD Architecture

In an SSD, a single flash internal consists of multiple connected dies through a serial I/O bus and common control signals. Each die has its own chip enable and ready/busy signals. Thus, one of the dies can accept commands and data while the others are carrying out other operations. Furthermore, a die consists of multiple planes, and each plane is framed by multiple blocks containing pages where data is actually stored. Additionally, each plane has some register space to

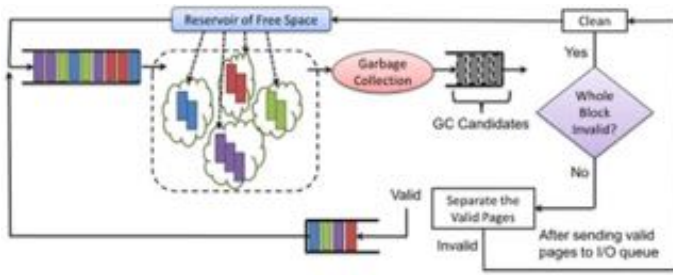


Fig. 9. Modification to garbage collection in order to enable multi-stream.

store data allocation and each block has one page reserved for metadata. We implement our FIOS in DiskSim’s SSDSim plugin [7], which has been widely used to simulate SSDs. However, DiskSim does not support the multi-streaming technology. Thus, we modified this existing SSD simulator to allow us to simulate multi-stream SSDs and evaluate our new FIOS method. The overhead of our implementation will be discussed in Section 7.

In particular, we mainly modify two modules, i.e., `ssd_init` and `ssd_clean`. The `ssd_init` module is modified to maintain a `streamID` attached to the hierarchy of `active_page`, `active_block`, and `active_plane`. We ensure that for all dies, the total number of `active_planes` is always equal to the number of streams. Each `active_plane` has only one `active_block` and each `active_block` has only one `active_page`. Thus, the main difference between a legacy SSD and a multi-stream SSD is that we have multiple simultaneous open erase blocks (one for each stream) in the multi-stream SSD, but only a single open erase block in the legacy SSD.

If all pages in `active_block` are filled, then `block_alloc_pos` will take care of assigning a new block, which then becomes `active_block` with the same `streamID` as the previous `active_block`. Blocks are not pre-allocated to each stream. Instead, they are allocated on-demand to each stream after the previous one of the same stream is finalized. In DiskSim, the `ssd_clean` module performs GC, which is a global component and not aware of the streams. Thus, when garbage collection starts, `ssd_clean` should search all the blocks to find one or multiple candidates to clean. On the other hand, there are still blocks with some valid pages whose `streamIDs` need to be preserved. The modified `ssd_clean` module of DiskSim thus assigns these pages to a new physical plane that is ensured to belong to the preserved `streamID`. Figure 9 illustrates how the modified DiskSim works for supporting multi-streaming when garbage collection happens.

5 PLATFORM SETUP

Our evaluation environment of DiskSim is calibrated based on the real testbed specs, summarized in Table 2. We adopt the similar flash volume structure developed in our previous work [11], which consists of a logical volume of 3 TB with full stripping of 128 KB. We configure the parameter file of DiskSim [7] to support the **On-Stack Replacement (OSR)** write policy [30] and the wear-aware garbage collection cleaning policy [27]. We modify the SSDSim module of DiskSim to enable simulating operations of multi-stream SSDs to validate our results. To calibrate DiskSim, we use the prototype of multi-stream SSDs made by using Samsung’s NVMe PM953 SSD with M.2 form factor. The Ubuntu kernel patches to use this multi-stream SSD prototype is available at <https://github.com/multi-stream/multi-stream>. Here, we set 64 sectors on a disk (i.e., 64×512 Bytes on SSDs) as one `sector_chunk` by default for most of our experiments considering the empirically observed sensitivity towards storage overhead of `streamID` computation.

Table 2. Testbed Specs

CPU Type	Intel(R) Xeon(R) CPU E5-2640 v3
CPU Speed	2.60 GHz
Number of CPU Cores	32 Hyper-Threaded
CPU Cache Size	20480 KB
CPU Memory	128 GB
OS Type	Linux
Kernel Version	4.2.0-37-Generic
Operating System	Ubuntu 16.04 LTS
File System	ext4
Flash Storage	960 GB
Page Size	8 KB
Pages Per Block	64
Blocks Per Plane	351,562
Planes Per Die	8
Dies Per Element	2
Elements Per Gang	1
Flash Over-Provisioning	11%
Metadata Storage Reservation	1 Page Per Block

We conduct a trace-replay simulation in modified DiskSim by using 100+ enterprise workloads from the **University of Massachusetts (UMass)** [4], **Microsoft Research—Cambridge (MSR)** [34], and **Florida International University (FIU)** [5] trace repositories. We use Linux default file system ext4, which is a journaling file system. We understand that using different file systems may yield different performance, but this is outside the scope of this work. We analyze the performance using a wide variety of workloads with different characteristics. Table 3 shows some workload characteristics of the selected workloads. Brief descriptions about workload characteristics shown in Table 3 are as follows,

- *ACF*—the auto-correlation factor of inter-arrival time between two write/update requests;
- *RW*—the percentage of random writes among the total write I/Os;
- *S*—the estimated average size of each write;
- *WV*—the working volume that represents the spatial capacity demand;
- *R*—the reuse ratio of the total amount of data (in bytes) accessed in the disk (i.e., *WV*) to the total address range (in bytes) of accessed data, which is the unique set of *WV*. A large *R* ensures that there is some unique data that is reused heavily;
- λ —the daily logical write rate (GB/day);
- *Peak rate*—the peak throughput demand with the 5 minutes statistical analyses window;
- *Throughput*—the overall operation rate.

The primary goal of multi-streaming technology is to provide better data placement which can reduce the extra writes during garbage collection. Thus, the major evaluation metric is WAF, which is the ratio of physical writes (in bytes) on a device to logical writes (in bytes) by applications running on a host. The lower the WAF is, the better the lifetime and performance of the flash device are.

To evaluate our OFIOS, we setup a total of eight parallel docker containers to simultaneously execute four different SQL and NoSQL database applications—MySQL, Cassandra, RocksDB, and MongoDB for 1 hour. We have two containers for each application. Each database application

Table 3. Statistics for Postmark, IOzone and Selected FIU, MSR-Cambridge and UMASS Workloads

Workload	ACF	RW (%)	R	S (KB)	WV (KB)	λ (GB/Day)	Peak rate (IOPS)	Throughput (IOPS)
Postmark	0.99	98.29	37.61	26	10244024	43.87	112.32	18.45
IOzone	0.98	49.45	1.21	197	5094328	28.08	109.53	14.08
FIU-1	0.78	75	157.35	9	1065000000	139.4	271.65	6.6
FIU-2	0.87	82	32.89	8	1084000000	114.72	156.79	1.02
FIU-3	0.97	57	182.04	8	1084000000	185.09	207.02	2
MSR-hm0	0.92	66	4.48	15	28000000	58.51	254.55	9.24
MSR-hm1	0.95	58	157.37	31	51000000	1.57	156.13	6.98
MSR-mds0	0.98	69	32.47	19	67000000	21.04	298.33	23.38
MSR-prn0	0.98	61	4.72	25	132000000	131.33	409.66	17.72
MSR-proj0	1	27	6.27	29	32000000	412.19	484.82	28.95
UMASS-1	0.21	64	1242.93	8	1289000000	575.94	218.59	122.05
UMASS-2	0.02	76	1.13	5	1156000	76.6	159.94	90.25

(ACF - Auto-Correlation of Inter-arrival Time, RW - Random Write, R - Reuse Ratio, S - Estimated Size of each Write, WV - Working Volume Size, λ - Write Rate, BFC - Best Feature Combination).

Table 4. Workload Configurations of Different Real Applications (KV - key/value, col. - columns, conn. - connections, R - reads, W - writes, U - updates, Bench. - benchmark, W1 to W6 - workloads)

Application	Bench.	Size	W1	W2	W3	W4	W5	W6
MySQL-1	TPCC	4200 warehouse	100 conn.	200 conn.	300 conn.	400 conn.	500 conn.	600 conn.
MySQL-2	TPCC	200 warehouse	1000 conn.	2000 conn.	3000 conn.	4000 conn.	5000 conn.	6000 conn.
Cassandra-1	Cassandra stress	10 million records	R/W 70%/30%	R/W 60%/40%	R/W 50%/50%	R/W 40%/60%	R/W 30%/70%	R/W 20%/80%
Cassandra-2	Cassandra stress	1 million records	R/W 80%/20%	R/W 70%/30%	R/W 60%/40%	R/W 50%/50%	R/W 40%/60%	R/W 30%/70%
RocksDB-1	DB_bench	560 million records	U/R 50%/50%	U/R 60%/40%	U/R 70%/30%	U/R 80%/20%	U/R 90%/10%	U/R 100%/0%
RocksDB-2	DB_bench	10 million records	U/R 100%/0%	U/R 90%/10%	U/R 80%/20%	U/R 70%/30%	U/R 60%/40%	U/R 50%/50%
MongoDB-1	YCSB	220 million records	U/R 50%/50%	U/R 60%/40%	U/R 70%/30%	U/R 80%/20%	U/R 90%/10%	U/R 100%/0%
MongoDB-2	YCSB	10 million records	U/R 100%/0%	U/R 90%/10%	U/R 80%/20%	U/R 70%/30%	U/R 60%/40%	U/R 50%/50%

runs 6 different streams of workloads. Each workload runs for 10 minutes. Table 4 shows the configuration of our 8 containers and 6 workloads of each container.

6 EVALUATION

In this section, we first study the impact of streamID identification using different features, such as frequency, sequentiality, on the WAF of multi-stream SSDs, and overall throughput of workloads. Then, we analyze characteristics of different workloads and derive some implications for

determining which feature or a combination of features can obtain the minimum WAF for a given workload. Then, we validate the FIOS automatic feature selection model to determine the best features for streamID identification. We also evaluate the performance impact of FIOS on the overall throughput of applications, and the sensitivity of FIOS with different sector chunk sizes. Finally, we discuss our evaluations of OFIOS, by comparing its performance with the state-of-the-art techniques. We also analyze the sensitivity of OFIOS to its variable parameters.

6.1 Impacts of Workload Features

We inspect the impact of streamID identification using different features on the WAF. The baseline of our comparison is the WAF of no streaming legacy SSDs with the same capacity and configurations. We also compare the results of our feature-based streamID identification with one straightforward way of partitioning data into different streams. In this straightforward approach, the total address space of SSDs is equally partitioned into the number of supported streams, and streamID is assigned according to I/O address.

As discussed in Section 3, our FIOS framework extracts various workload features and considers different feature combinations to cluster logical blocks into streams. Figure 10 shows the results of relative WAFs normalized by the WAF of no streaming legacy SSD for binary feature matrix. Later in Section 3.3, we discuss our advancement regarding the decimal feature matrix. The blue horizontal line at 1 represents the relative WAF of legacy. Thus, the smaller the relative WAF is, the better the endurance of multi-stream SSDs can be obtained. Specifically, 15 bars represent some possible feature combinations used by FIOS. For example, *FSAC* stands for a combination of *Frequency*, *Sequentiality*, *Adjacent access*, and *Coherency*.

We conduct our experiments with a totally of 100+ workloads and summarize our observations with 12 representative workloads in Figure 10. Overall, we observe that the multi-streaming technology offers a good opportunity to reduce WAF, e.g., at least a 20% reduction in WAF for all workloads. For some write-intensive workloads e.g., Postmark (see Figure 10(a)) that have the majority of random writes, the WAF can be reduced by more than 70%. On the other hand, we also notice that none of these features can be always the best for different types of applications. For example, *Coherency* gives the best WAF reduction for MSR-hm0 (see Figure 10(f)), while a combination of ***Frequency and Coherency (FC)*** is much better for MSR-hm1 (see Figure 10(g)). This is because different workloads have different characteristics. For example, by looking closely to the workloads we find that both MSR-hm0 and MSR-hm1 have more random writes and a high auto-correlation (i.e., ACF) of update time, see Table 3. As a result, the feature of coherency that groups randomly occurred friendly sector_chunks into the same stream becomes a good choice. Additionally, MSR-hm1 has a high reuse ratio of updates. Thus, combining features of frequency can capture the multi-touch reuse count. Moreover, from Figure 10(g), we also find that further adding other features (e.g., sequentiality), FSC does not further reduce the WAF, which implies that the combination of *more features does not guarantee a better reduction in WAF*.

Furthermore, Figure 10 shows the results of relative WAFs normalized by the WAF of no streaming legacy SSD using the decimal feature matrix when compared with the binary feature matrix. Overall, we observe that for different workloads, the WAF is reduced by using the decimal feature matrix when compared to the binary feature matrix (see Figure 10). This is because, with the decimal feature matrix, we can capture granular properties to cluster data more accurately into streams. Furthermore, we also notice that the best feature combination with which we obtain the maximum WAF reduction for a workload remains the same irrespective of the granularity of the feature matrix. For example, *Coherency* gives the best WAF reduction for MSR-hm0 with both the binary and decimal feature matrices. This validates that the best feature combination is

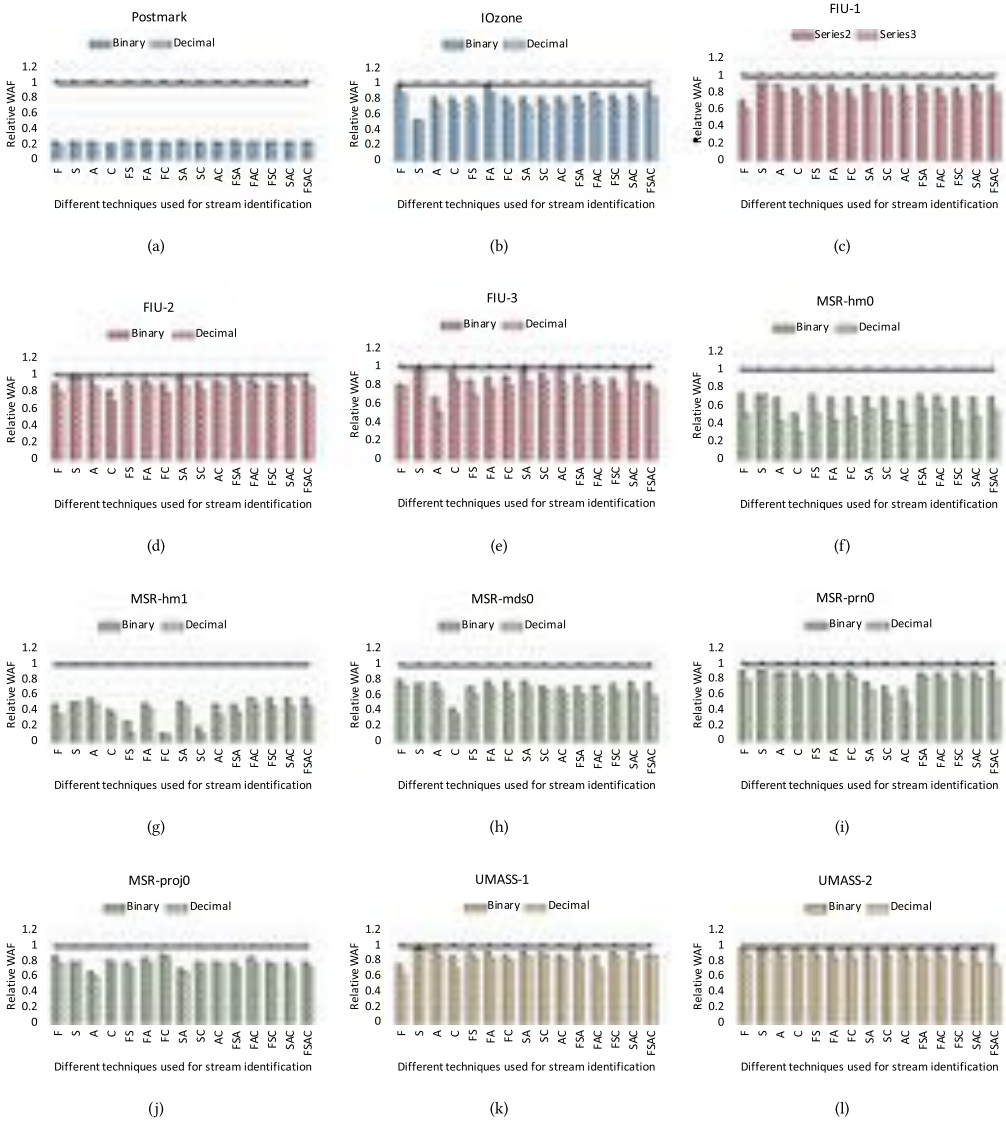


Fig. 10. Relative WAF w.r.t. a legacy device for Postmark, IOzone and selected FIU, MSR-Cambridge, and UMASS workloads by using our framework with dictionary framed from different features and their combinations for streamID identification. (F - Frequency, S - Sequentiality, A - Adjacent access).

closely related to workload characteristics, and we can accurately predict the best feature combination by understanding the relation between features and workload characteristics.

6.2 Validation of Feature Selection by FIOS

Now, we turn to validate our approach for a given workload choosing the best combination of features under various workloads. We use WAF as the metric to evaluate our FIOS under six different I/O workloads, as shown in Figure 11. The best feature combination derived by FIOS are also mentioned on the top of the FIOS bars in Figure 11. For comparison, we plot the results of

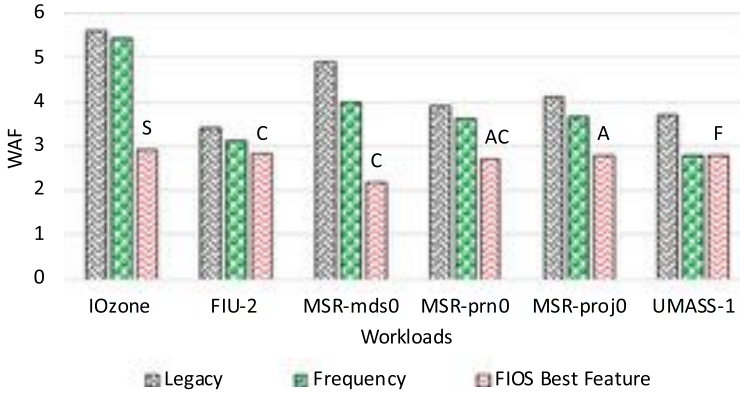


Fig. 11. Write amplification factor using the best feature selected using FIOS when compared to legacy no streaming, streamID identification with equal partition, and with a usually used feature of frequency. The best feature selected by FIOS is mentioned on top of the bar for each workload.

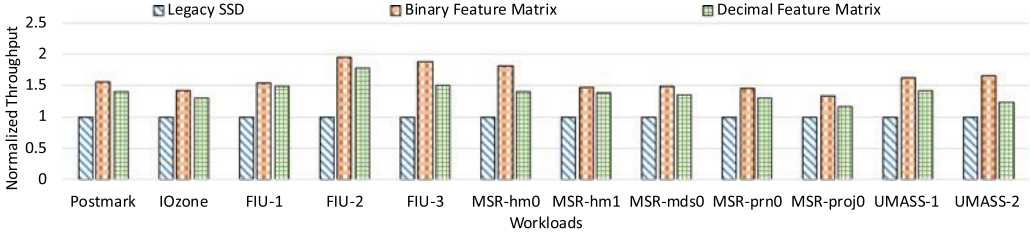


Fig. 12. Throughput of workloads using the binary feature matrix and the decimal feature matrix for normalized to throughput obtained using non-streaming legacy SSD.

legacy and multi-stream SSDs under the fixed feature (e.g., frequency) selection approaches. These results imply that the existing methods that use the fixed features (such as frequency) for streamID assignment cannot effectively utilize the benefits of multi-stream SSDs. In contrast, for all these workloads FIOS is able to obtain the lowest WAF by using appropriate features for streamID assignment.

6.3 Performance and Sensitivity of FIOS

Furthermore, for the best feature combination, we compare the workload throughput of multi-stream SSDs with non-streaming SSDs, while using the binary feature matrix and the decimal feature matrix in Figure 12. We observe that by using a multi-stream SSD with accurate data streams, the throughput of workloads is improved by at least 1.17 times when compared with a legacy SSD with no streaming capability. This is because by better organizing the data within SSDs, the overhead for internal operations such as GC is reduced, and hence more device bandwidth becomes available for the application data. We also see that the throughput of using the binary feature matrix is better when compared to the decimal feature matrix. This is because computing the decimal feature matrix for each feature requires more time, as it is more complicated than the binary feature matrix. Thus, using the decimal feature matrix instead gives better endurance, but consumes some additional performance overhead. However, the overall throughput using the decimal feature matrix still remains better when compared to the legacy SSD. *In summary, our results show*

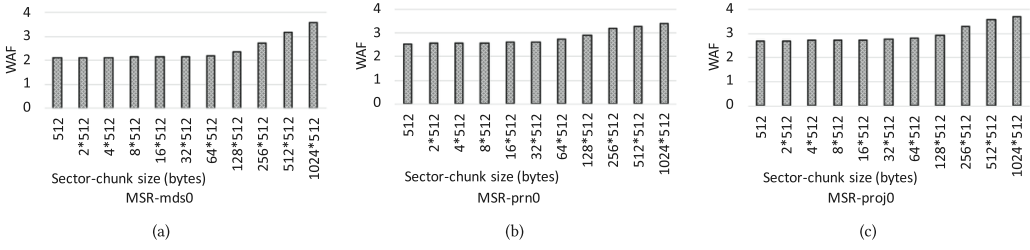


Fig. 13. Sensitivity analysis by changing sector chunk size (a) MSR-mds0, (b) MSR-prn0, and (c) MSR-proj0.

that by allowing us to better organize the data within multi-stream SSDs, we can reduce WAF as well as improve throughput.

Finally, in Figure 13, we observe the sensitivity of FIOS with different sector chunk sizes. We see that with a smaller sector chunk size, better (i.e., lower) WAF can be obtained. However, the rate of the increase in WAF with the increase in the size of sector chunk is initially very low, but WAF increases rapidly for sector chunks with larger sizes. Very small size of sector chunks (e.g., 512 bytes) will lead to high management overhead in terms of memory required to store dictionaries as well as CPU cycles required to compute for many chunks. While, very large sector chunk size (e.g., 1024×512 bytes) reduces the WAF reduction benefit of FIOS. Therefore, after conducting such empirical analysis on many different workloads, we chose to set chunk size as 64×512 bytes because for most of the workloads, we observed this is a knee point until which the rate of increase of WAF with the increase in the sector chunk size is low and after that, it increases rapidly (see Figure 13).

6.4 OFIOS Results

To evaluate our OFIOS, we setup a total of eight parallel docker containers to simultaneously execute four different SQL and NoSQL database applications as described earlier in Section 5. We empirically configure $T_{RunWindow}$ and $T_{TrainWindow}$ to have an equal length of 100 seconds for optimally capturing the dynamics of workloads. However, to reduce the overhead, depending upon the number of cores in the system, the length of the $T_{TrainWindow}$ can be reduced in OFIOS. The length of the $T_{TraceWindow}$ needs to be less than the length of $T_{TrainWindow}$. Empirically observing the time taken to construct new dictionaries by FIOS, we set $T_{TraceWindow}$ as 75% of $T_{TrainWindow}$. Later, in Section 7 we will further explore the time consumed by FIOS to construct new dictionaries.

Figure 14 shows the runtime WAF under both FIOS and OFIOS, where the green line with a circle marker shows the WAF observed using FIOS with Frequency as a feature and the red line with a triangle marker shows the WAF observed using OFIOS. We can see that the overall WAF can be further reduced by using OFIOS. As we know, FIOS cannot dynamically adapt the dictionary during the runtime. For example, after the initial training in the first 100 seconds, Frequency is identified as the best feature and then used for the streamID assignment throughout the entire execution time by FIOS. Whereas, OFIOS automatically adapts the dictionary during the runtime of the workload. Thus, according to the workload, the best feature combination to assign streamIDs is periodically adapted. In Figure 14, the feature combinations of OFIOS are shown along the x-axis for every 100 seconds. We see that the adaptive OFIOS can further improve the endurance of SSDs.

Further, in Figure 15, we compare the performance of three existing techniques with OFIOS. Figure 15(a) shows that the WAF of our scheme is reduced by 47.1% and 2.6% compared with baseline and LSTM+KM [40], respectively. OFIOS also results in lower average WAF compared to the other two techniques of multi-queue using AutoStream [39] and vStream [42]. All these three

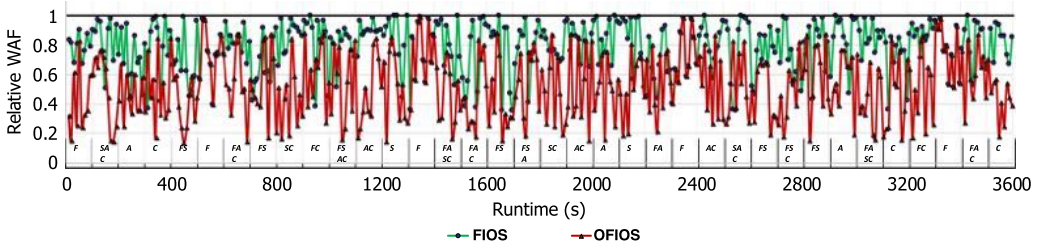


Fig. 14. Reduction in relative WAF w.r.t. legacy device using OFIOS compared to FIOS .

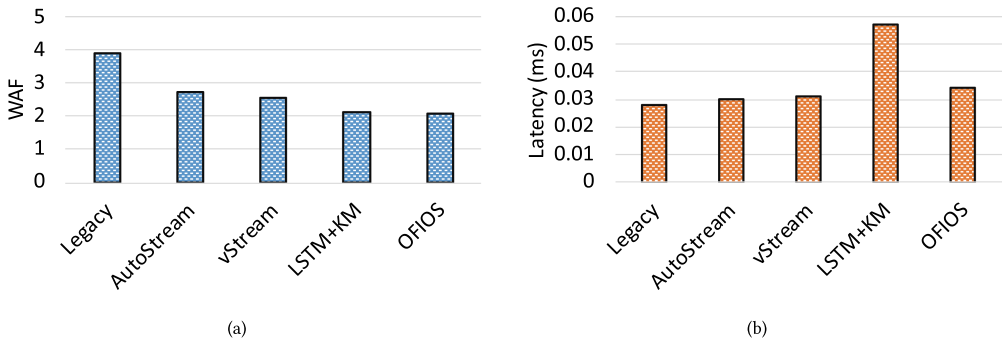


Fig. 15. Comparison with state-of-the-art techniques for: (a) Write Amplification Factor, and (b) Average I/O Latency.

previous techniques uses update frequency as its feature. We implement a multi-queue technique proposed in AutoStream [39] that uses multiple software queues to maintain data in sorted order w.r.t. its update frequency. The multi-queue technique implements promote and demote operations within these queues to track the changes in the workloads. vStream [42] identifies virtual streams by post processing the dead pages, and then uses k-means clustering to group any number of virtual streams to an available number of physical streams. LSTM+KM [40] predicts the future frequency using a neural network called **Long Short-Term Memory (LSTM)** and then uses k-means clustering to group writes into an available number of physical streams.

For an efficient streamID assignment, it is very important that it runs with minimum critical I/O path latency. Usually millions of I/Os are performed every second on high performance storage server, so even a small additional latency to each I/O can result in very drastic application throughput degradation. Thus, apart from just evaluating the reduction in WAF, we also evaluate and compare the additional I/O latency overhead in Figure 15(b). Figure 15(b) shows the average combined read/write latency while running the above eight simultaneous application containers. We see that OFIOS introduces very minimal additional latency compared to legacy (non-streaming) SSD. Other techniques of AutoStream [39] and vStream [42] also introduce small additional latencies due to their simplistic algorithms. However, their WAF benefit is not as good as that of OFIOS (see Figure 15(a)). LSTM+KM [40] introduces comparatively the highest additional latency due to its use of the neural network. LSTM neural network require a lot of resources and time to get trained and become ready to provide accurate prediction. Moreover, at runtime recurrent training is required to adopt to the changing workloads. Thus, LSTMs can yield high prediction accuracy, but become quite inefficient for latency sensitive applications. This is also acknowledged by the authors of LSTM+KM in their article. With reduced GC overhead in OFIOS, more device

bandwidth is released to serve both read and write I/Os. So, the additional performance overhead of our framework is mostly compensated with such performance improvement, resulting into a minimal overhead. Finally, the above results indicate that our scheme can obviously reduce the WAF by introducing very low additional read/write latency to capture the impact of dynamically changing parallel applications. Additionally, notice that OFIOS performs well even while using append-only applications such as RocksDB, because the application's append-only pages that stores the always increasing log is a notion of virtual addressing which is different from the physical pages within SSDs. Thus, although the application's append-only pages (e.g., SST files) keep increasing instead of overwriting, as a fixed size SSD can only have fixed number of LBAs, so the same LBAs needs to be reused. To perform good stream identification, OFIOS do not need to relate the lifetime of the application's append-only pages with the LBAs. Rather, the feature that we discuss in our work are directly captured at the block layer.

Finally, we analyze the sensitivity of OFIOS to the duration of $T_{RunWindow}$ and $T_{TrainWindow}$. Configuring $T_{TrainWindow}$ less than the duration of $T_{RunWindow}$ ensures that the additional resources consumed by background operations to train our model are only occupied during the fraction of the runtime window. Note that configuring $T_{RunWindow}$ smaller than $T_{TrainWindow}$ just leads to additional overhead without any benefits. Thus, we conducted experiments with the training for a different proportion of the run window. Particularly, we train for the last "n" portion of each run window. We set "n" as 20%, 40%, 60%, 80% and compare the results with $n = 100\%$ (i.e., $T_{TrainWindow} = T_{RunWindow}$). $n = 100\%$ means that we always train in the background. The results in Figure 14 show the WAF of OFIOS with $n = 100\%$. The length of the training window that is equal to that of the run window (i.e., $T_{TrainWindow}/T_{RunWindow} = 1$) signifies that we capture the activities of the workloads in the background to dynamically train OFIOS to adapt quickly to workload changes. Decreasing the ratio $T_{TrainWindow}/T_{RunWindow}$ means we train only for the portion of each run window. For example, $T_{TrainWindow}/T_{RunWindow} = 0.2$ means we train for last 20% of each run window. Then, based upon the observations in this last 20% of the run window, we decide the feature to use in the next time window.

Figure 16 shows the runtime WAF for four different settings of the training window time (i.e., $T_{TrainWindow}/T_{RunWindow} = 0.2, 0.4, 0.6, \text{ and } 0.8$). Figure 16 also shows the feature combinations used by OFIOS to identify streams under each of these settings while executing multiple parallel containerized workloads. Figure 17 shows the average WAF for the same experiments discussed in Figure 16. Figures 16 and 17 show that higher WAF reduction can be achieved with a larger training window as feature identification is done better according to the dynamics of the workloads. Upon in-depth analysis of feature combinations used by OFIOS during run time for different intermediate time intervals, we see that the larger the training window, more number of windows used same as that of features used with $T_{TrainWindow}/T_{RunWindow} = 1$. The blue shaded features represent that for those time windows, the same feature as that of $T_{TrainWindow}/T_{RunWindow} = 1$ is used. More number of blue-shaded time windows represent that OFIOS can perform well using more appropriate feature combinations resulting in lower WAF. OFIOS will not work well if the LBAs are managed in an append-only fashion. This is the limitation of our proposed technique. However, we would like to emphasize that as the file system organizes and maintains the mapping of application data into logical block addresses. Hence, OFIOS works fine with append-only applications if the underlying file system is not append-only. This can be observed from our results in this section in which we validate OFIOS using append-only applications such as RocksDB. We use Linux default file system ext4, which is a journaling file system. We understand that using append-only file systems such as **log-structured file system (LFS)** may raise new challenges. We will pay attention to design new methods for dealing with append-only file systems in our future study.

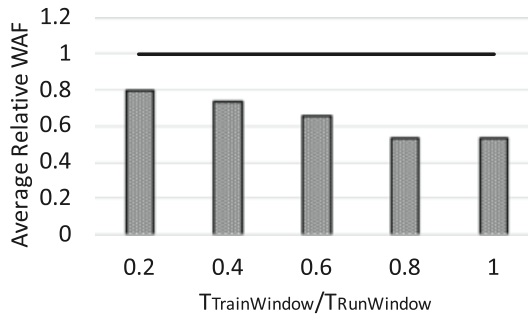


Fig. 17. Average relative WAF w.r.t. legacy device, training and running for different size of time windows (i.e., $T_{TrainWindow}/T_{RunWindow}$).

applications be responsible to identify different streams (for e.g., commit logs, metadata, indices, and so on.) in their workloads. However, when multiple applications are running simultaneously on the same host, stream management becomes more complex. Another option is to perform stream identification at the file system layer that appends each file with a corresponding streamID. This is portable to different applications without modifying them. But, this approach sometimes does not work when some applications bypass the file system to speed up data access. In addition, as the upper layer application changes, the file system may not be able to dynamically adjust the streamID assignment. On the other hand, streamID identification at the FTL layer is limited by the computing and buffering capability of an SSD that is comparatively slower and smaller when compared to the computing speed and buffering capability of the host (i.e., CPU and main memory). We find that the block layer implementation can avoid the above issues, i.e., being portable to different applications and multi-tenant environments and being able to utilize the host resources for stream management. Thus, we choose to implement our stream identification framework at the block layer in this article.

7.2 Implementation Overhead

We use BD and SID, as described in Section 4, to maintain additional data of our framework. For a 3 TB flash volume SSD storage, we require less than 50 MB in total to store all streamID metadata, which is only 0.001% of the total flash disk space. In our experiments, we have 128 GB DRAM in our server. The SID footprint for different workloads consumes less than 25 MB of main memory. Therefore, the SID footprint overhead is only about 0.02% of the size of the main memory. We mainly modified two modules, i.e., `ssd_init` and `ssd_clean` in `DiskSim`. In order to pertain the modification throughout, we had to make subsequent changes in the `ssd_timing`, `ssd_gang`, and `ssd_utils` modules. In total, we modified approximately 560 lines of codes to enable the operation of a general multi-stream SSD excluding streamID identification.

The streamID identification time under FIOS is considered as the total additional operation time, including the time required for feature extraction, K-means clustering, construction of aBD and SID. Figure 18 shows the average time for streamID identification by FIOS using four features (i.e., frequency, adjacent access, sequentiality, and coherency) with respect to a number of blocks in the workload. We can see that FIOS does add extra latency in order to identify the best feature combinations for improving SSD endurance. With an increase in dataset size (i.e., number of blocks), the time to identify and maintain streamIDs increases as well. In our experiment, a 1.5 TB SSD was required to run a workload with 3 billion LBAs (blocks). Thus, we believe it is acceptable for FIOS to take around 600 seconds for streamID identification. FIOS is greatly advantageous to be used for all the applications and workloads that are not very latency sensitive such as email-server, but even

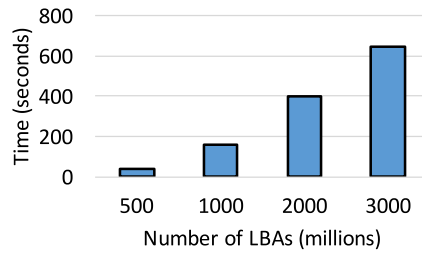


Fig. 18. Average time for streamID identification using four features (frequency, adjacent access, sequentiality, and coherency) and the binary feature matrix as a function of the number of blocks in the workload.

for latency-sensitive applications (such as online marketplace) the additional delay introduced is manageable. The OFIOS simultaneously queries the streamID using SID, while working to identify data streams in the background. As DRAM access latency is uniform and very low, so we notice that on an average each query takes around 20 ns. Thus, collectively querying for 3 billion LBAs only consumes $6\mu\text{s}$ additional to the time taken by background operations to identify streams. Finally, we note that there exists a trade-off between time efficiency and endurance improvement. To take advantage of OFIOS, configuring appropriate window size is necessary considering the abovementioned overheads and the estimated blocks accessed per second by the workloads. Since a multi-stream SSD is developed to enhance the lifetime of SSDs, we believe that the main goal of a streamID identification algorithm should be to reduce write amplifications with the minimal time overhead.

8 RELATED WORK

The high performance (compared to HDDs) and recently reduced cost (e.g., \$/GB) of SSDs make them perfect persistent storage devices. However, currently, the major issue with the flash technology is the limited lifetime of flash cells due to limited PE cycles. In order to balance the wear out of flash cells in the device and enable them to use it as plug and play, efficient FTL algorithms [8, 14, 24, 28, 29, 36, 43] have been developed. The FTL plays an important role in filtering out sequential streams from the mixed (sequential + random) streams. The work in [7] observed that SSD's performance and lifetime are highly sensitive to I/O workloads. The previous work in [32] designed new file systems, specifically for enhancing the lifetime of the SSDs. These file systems concentrated on transforming all random writes at the file system level to sequential ones at the SSD level and considered a new data grouping strategy on writing by putting data blocks with similar update likelihood into the same segment. It was noticed that the reduction of the internal write amplification of SSDs might increase the lifetime of SSDs. Thus the research work started exploring all features that are related to the write amplification factor. Reference [16] proved that the write amplification factor of SSDs depends on over-provisioning, cleaning policy, and workload pattern. Based on the measurement on NVMe disks, the study in [41] further revealed the relationship between write amplification factor and the write I/O sequential ratio.

The recent technology evolves a new product of multi-stream SSDs, which offers an intuitive storage interface to inform the host system about the expected lifetime of the data [1, 18, 20, 21, 23, 35]. Experimentation in [23] uses a real multi-stream SSD prototype to show that the worst-case update throughput of a Cassandra NoSQL database system can be improved by nearly 56%. The work in [15, 23, 38] did modification in certain applications (i.e., FIO [6], Cassandra [31], and RocksDB [19]) to enable application assigned streamID. The study shows that with multi-streaming, SSDs can be more efficiently used for achieving consistently better performance and endurance. These existing techniques to assign streamIDs for a multi-stream SSD are too

specific (such as application assigned streamIDs which requires to modify the application). Recently, LSTM+KM [40] proposes a scheme to place data according to its predicted future temperature using a neural network called LSTM in temporal and spatial dimensions. Ref. [42] proposes a new concept of **virtual streams (vStreams)** for the multi-streamed SSDs. vStreams provide application developers to a large number of virtual streams regardless of the number of physical streams supported by the device. However, the mapping of these virtual streams to the physical streams needs to be managed by applications. In [25, 26], stream management technique called PCStream the stream allocation decisions are also made at a higher abstraction level. Thus, PCStream does not support widely used applications like MySQL [17] that rely on a write buffer. Moreover, applications MonetDB [22] that heavily access files with mmap related functions such as mmap() and msync() are not supported. As a result, in our research, we present a more portable and adaptable method with minimalist overhead for feature-based streamID assignment in multi-stream SSDs at the LBA level. Additionally, the innovation of our work lies in the scalable nature of our method, which considers multiple I/O features simultaneously to group data into any number of streams required by the SSD firmware.

9 CONCLUSION

In this article, we first presented the difference between multi-stream storage and legacy storage and explored the benefits of enhancing SSDs with multi-stream, i.e., increasing the lifetime and performance of SSDs. We then investigated the impact of different workload features on write amplification to enable better utilization of multi-stream SSDs. We proposed a feature-based streamID assignment technique (FIOS), which is capable of extracting features and assigning streamIDs with low overhead. FIOS is also scalable to incorporate different numbers of features and construct multiple streams to support the framework. To the best of our knowledge, this is the first streamID assignment technique for multi-stream SSDs, which does not require any modification to applications for using multi-stream SSDs. We also proposed a new feature to better capture a lifetime, called coherency, which represents the friendship between write operations with respect to their update time. We further investigated the correlation between workload attributes and workload features to automatically determine a combination of features that can offer the most WAF reduction. Finally, we designed the OFIOS technique to learn along with the changing-workload in the background and automatically adapt the best feature combination to assign streamIDs. Our experimental results show that FIOS achieves at least a 20% decrease in write amplification across different workloads, which is a good qualitative improvement. We also found that different features have varying impacts on WAF, which indicates the importance of identifying a good combination of features. Our evaluation also shows that our automation approach OFIOS of stream detection can effectively further reduce the WAF by dynamically changing the feature combination w.r.t. changes in workloads. In our future work, we plan to improve our methods to deal with the append-only file systems and explore different machine learning techniques to learn the optimal value for the parameters such as window size of background training, trace collection time, and sampling size.

ACKNOWLEDGMENTS

This work was initiated during Janki Bhimani's internship at Samsung Semiconductor Inc. [12].

REFERENCES

- [1] Multi-Stream Technology. 2020. Retrieved 15 March, 2020 from <http://www.samsung.com/semiconductor/insights/article/25465/multistream>.
- [2] Performance and Endurance Enhancements with Multi-stream SSDs on Apache Cassandra. 2020. Retrieved 27 Jan., 2020 from https://www.samsung.com/semiconductor/global.semi.static/Multi-stream_Cassandra_Whitepaper_Final-0.pdf.

- [3] systemd. 2020. Retrieved 18 Dec., 2020 from <http://manpages.ubuntu.com/manpages/bionic/man1/systemd.1.html>.
- [4] UMass Trace Repository. 2020. Retrieved 18 Dec., 2020 from <http://traces.cs.umass.edu/index.php/Storage/Storage>.
- [5] (accessed January 13, 2017). *SNIA Iotta Repository*. Retrieved from http://iota.snia.org/historical_section.
- [6] (accessed September 7, 2016). *FIO - flexible I/O benchmark*. Retrieved from <http://linux.die.net/man/1/fio>.
- [7] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design tradeoffs for SSD performance. In *Proceedings of the USENIX Annual Technical Conference*. 57–70.
- [8] Jinwook Bae, Hanbyeol Kim, Junsu Im, and Sungjin Lee. 2019. Demand-based FTL cache partitioning for large capacity SSDs. *IEMEK Journal of Embedded Systems and Applications* 14, 2 (2019), 71–78.
- [9] Jacob Benesty, Jingdong Chen, Yiteng Huang, and Israel Cohen. 2009. Pearson correlation coefficient. In *Proceedings of the Noise Reduction in Speech Processing*. Springer, 1–4.
- [10] Janki Bhimani, Miriam Leeser, and Ningfang Mi. 2015. Accelerating k-means clustering with parallel implementations and GPU computing. In *Proceedings of the High Performance Extreme Computing Conference*. IEEE, 1–6.
- [11] J. Bhimani, J. Yang, Z. Yang, N. Mi, Q. Xu, M. Awasthi, R Pandurangan, and V. Balakrishnan. 2016. Understanding performance of I/O intensive containerized applications for NVMe SSDs. In *Proceedings of the 35th IEEE International Performance Computing and Communications Conference*. IEEE.
- [12] Janki S. Bhimani, Jingpei Yang, Changho Choi, and Jianjian Huo. 2017. Smart I/O stream detection based on multiple attributes. (March 16 2017). US Patent App. 15/344,422.
- [13] Daniel Campello, Hector Lopez, Ricardo Koller, Raju Rangaswami, and Luis Useche. 2015. Non-blocking writes to files. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*. 151–165.
- [14] Hyunjin Cho, Dongkun Shin, and Young Ik Eom. 2009. KAST: K-associative sector translation for NAND flash memory in real-time systems. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. European Design and Automation Association, 507–512.
- [15] Changho Choi. Multi-Stream Write SSD: Increasing SSD Performance and Lifetime with Multi-Stream Write Technology. 2020. Retrieved 18 Dec., 2020 from http://www.flashmemorysummit.com/English/Collaterals/Proceedings/2016/20160809_FC12_Choi.pdf.
- [16] Peter Desnoyers. 2012. Analytic modeling of SSD write performance. In *Proceedings of the 5th Annual International Systems and Storage Conference*. ACM, 12.
- [17] Paul DuBois and Michael Foreword By-Widenuis. 1999. *MySQL*. New riders publishing.
- [18] Stephen G Fischer, Changho Choi, Jason Martineau, and Rajinikanth Pandurangan. 2018. Methods for multi-stream garbage collection. (October 25 2018). US Patent App. 15/821,708.
- [19] Felix Gessert, Wolfram Wingerath, Steffen Friedrich, and Norbert Ritter. 2017. NoSQL database systems: A survey and decision guidance. *Computer Science-Research and Development* 32, 3–4 (2017), 353–365.
- [20] HUEN Hingkwon and Changho Choi. 2019. Method of consolidate data streams for multi-stream enabled ssds. (May 2 2019). US Patent App. 16/219,936.
- [21] HUEN Hingkwon, Changho Choi, Derrick Tseng, and Jianjian Huo. 2020. Multi-stream SSD QoS management. (March 17 2020). US Patent 10,592,171.
- [22] S. Idreos, F. Groffen, N. Nes, S. Manegold, S. Mullender, and M. Kersten. 2012. Monetdb: Two decades of research in column-oriented database. *IEEE Data Engineering Bulletin* 35, 1 (2012), 40–45.
- [23] Jeong-Uk Kang, Jeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The multi-streamed solid-state drive. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Storage and File Systems*. USENIX Association, Philadelphia, PA. Retrieved from <https://www.usenix.org/conference/hotstorage14/workshop-program/presentation/kang>.
- [24] Jesung Kim, Jong Min Kim, Sam H. Noh, Sang Lyul Min, and Yookun Cho. 2002. A space-efficient flash translation layer for CompactFlash systems. *IEEE Transactions on Consumer Electronics* 48, 2 (2002), 366–375.
- [25] Taejin Kim, Sangwook Shane Hahn, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2018. PCStream: Automatic stream allocation using program contexts. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems*.
- [26] Taejin Kim, Duwon Hong, Sangwook Shane Hahn, Myoungjun Chun, Sungjin Lee, Jooyoung Hwang, Jongyoul Lee, and Jihong Kim. 2019. Fully automatic stream management for multi-streamed ssds using program contexts. In *Proceedings of the 17th USENIX Conference on File and Storage Technologies*. 295–308.
- [27] Andrey V. Kuzmin and James G. Wayda. 2016. Multi-array operation support and related devices, systems and software. (January 5 2016). US Patent 9,229,854.
- [28] Sungjin Lee, Dongkun Shin, Young-Jin Kim, and Jihong Kim. 2008. LAST: Locality-aware sector translation for NAND flash memory-based storage systems. *ACM SIGOPS Operating Systems Review* 42, 6 (2008), 36–42.
- [29] Sang-Won Lee, Dong-Joo Park, Tae-Sun Chung, Dong-Ho Lee, Sangwon Park, and Ha-Joo Song. 2007. A log buffer-based flash translation layer using fully-associative sector translation. *ACM Transactions on Embedded Computing Systems* 6, 3 (2007), 18.

- [30] Yixin Luo, Yu Cai, Saugata Ghose, Jongmoo Choi, and Onur Mutlu. 2015. WARM: Improving NAND flash memory lifetime with write-hotness aware retention management. In *Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies*. IEEE, 1–14.
- [31] Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2014. CaSSanDra: An SSD boosted key-value store. In *Proceedings of the 2014 IEEE 30th International Conference on Data Engineering*. IEEE, 1162–1167.
- [32] Changwoo Min, Kangnyeon Kim, Hyunjin Cho, Sang-Won Lee, and Young Ik Eom. 2012. SFS: Random write considered harmful in solid state drives. In *Proceedings of the 10th USENIX conference on File and Storage Technologies*. 12.
- [33] Mahdi Pakdaman Naeini and Gregory F. Cooper. 2018. Binary classifier calibration using an ensemble of piecewise linear regression models. *Knowledge and Information Systems* 54, 1 (2018), 151–170.
- [34] D. Narayanan, A. Donnelly, and A. Rowstron. 2008. Write off-loading: Practical power management for enterprise storage. *ACM Transactions on Storage* 4, 3 (2008), 10:1–10:23.
- [35] Hyun-Woo Park, Soyee Choi, Mijin An, and Sang-Won Lee. 2019. Freezing frozen pages with multi-stream SSDs. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–3.
- [36] Wei Xie, Yong Chen, and Philip C. Roth. 2016. Parallel-DFTL: A flash translation layer that exploits internal parallelism in solid state drives. In *Proceedings of the 2016 IEEE International Conference on Networking, Architecture and Storage*. IEEE, 1–10.
- [37] Fei Yang, Kun Dou, Siyu Chen, Mengwei Hou, Jeong-Uk Kang, and Sangyeun Cho. 2015. Optimizing nosql db on flash: A case study of rocksdb. In *Proceedings of the 2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops*. IEEE, 1062–1069.
- [38] Fei Yang, Kun Dou, Siyu Chen, Jeong-Uk Kang, and Sangyeun Cho. 2015. Multi-streaming RocksDB. In *Proceedings of the Non-Volatile Memories Workshop*.
- [39] Jingpei Yang, Rajinikanth Pandurangan, Changho Choi, and Vijay Balakrishnan. 2017. AutoStream: Automatic stream management for multi-streamed SSDs. In *Proceedings of the 10th ACM International Systems and Storage Conference*. 1–11.
- [40] Pan Yang, Ni Xue, Yuqi Zhang, Yangxu Zhou, Li Sun, Wenwen Chen, Zhonggang Chen, Wei Xia, Junke Li, and Kihyoun Kwon. 2019. Reducing garbage collection overhead in {SSD} based on workload prediction. In *Proceedings of the 11th USENIX Workshop on Hot Topics in Storage and File Systems*.
- [41] Zhengyu Yang, Manu Awasthi, Mrinmoy Ghosh, and Ningfang Mi. 2016. A fresh perspective on total cost of ownership models for flash storage. In *Proceedings of the 2016 IEEE 8th International Conference on Cloud Computing Technology and Science*. IEEE.
- [42] Hwanjin Yong, Kisik Jeong, Joonwon Lee, and Jin-Soo Kim. 2018. vStream: Virtual stream management for multi-streamed SSDs. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Storage and File Systems*.
- [43] Peiyong Zhang and Huanjie Tang. 2020. High-efficient superblock flash translation layer for NAND flash controller. *Electronics Letters* 56, 6 (2020), 278–280.

Received July 2020; revised March 2021; accepted June 2021

Auto-Tuning Parameters for Emerging Multi-Stream Flash-Based Storage Drives Through New I/O Pattern Generations

Jaki Bhimani¹, Adnan Maruf¹, Ningfang Mi, Rajinikanth Pandurangan, and Vijay Balakrishnan

Abstract—In the era of big data processing, more and more data centers in cloud storage are now replacing traditional HDDs with enterprise SSDs. Both developers and users of these SSDs require thorough benchmarking to evaluate and configure the variable parameters of emerging technologies. [2] and [3] are the recent development of the SSD industry, which assists in placing data on SSDs in a smart way to improve application performance and SSD endurance. The challenging part to use multi-stream SSDs is to assign stream IDs to incoming writes, such that each stream consists of data with a similar lifetime. The benefit of the stream management algorithms varies over different workloads. Thus, first, we propose a new framework, called Pattern I/O generator (PatIO), to capture the enterprise storage behavior that is prevailing across various user workloads, virtualization setup, file systems, and volume managers for the database server applications on flash-based storage. Second, using PatIO, we study what type of applications may be benefited by which stream assignment algorithm. Third, we design the framework to automatically tune the variable parameters of different stream identification algorithms of the multi-stream SSDs. Our evaluation shows 20 to 110 percent of the reward function increase, measuring the cumulative impact on application performance and SSD endurance.

Index Terms—Flash memory, I/O pattern generator, benchmarking, multi-stream SSDs

1 INTRODUCTION

OPTIMIZING the operation of modern cloud storage systems for various big data applications is critical. Evaluating the effect of any storage device firmware or hardware amendments using real system deployment requires a lot of resources, time and efforts towards installation and running of different workloads to test. Moreover, many different virtualization and system setup options in a cloud environment also need to be tested for each workload. The research advancement by evaluating a tiny subset of these possible settings and workloads then becomes very limited. Thus, benchmarking is very important for developers and users of evolving cloud storage.

Most traditional I/O benchmarking tools [4], [5] were designed for hard disk drives (HDDs). Hence, when benchmarking storage, the I/O workloads generated by these tools do not resemble the I/O activities of real workloads on flash-based solid-state drives (SSDs). The main problems are that with multiple design choices at the virtualization

and system layer, (1) the data generated by traditional synthetic I/O generators might be too simple, or (2) it demands a lot of time and storage space to generate and store different trace logs for each workload with trace-based I/O generators. Fig. 1 compares the variance of I/O size for reads and writes over NVMe SSDs (top) and HDDs (bottom). We run the TPC-H decision support benchmark with twenty-two different queries executed on eight different input tables of various sizes with Apache Spark application. We observe that I/O sizes running on NVMe SSD are completely different from those running on HDD. The size of both read and write I/Os exhibits a periodic pattern when using NVMe SSD. Large read/write I/Os are periodically clustered together, with some idle intervals between I/O size spikes. Therefore, new benchmark methods that capture realistic I/O activities and require fewer resources, time, and efforts are needed.

Motivated by this, first, we propose a new benchmarking framework, called Pattern I/O generator (PatIO) to capture the enterprise storage behavior that is prevailing across various user workloads, virtualization setup, file systems, and volume managers for different database server applications on flash-based storage. Second, we integrate PatIO with multi-stream SSDs, to study the impact of the various internal stream identification algorithms. Third, we design and integrate the auto-tuning module within multi-stream SSDs to tune the variable parameters of different stream identification algorithms. The main contributions and features of our solution are as follows.

1) *Extract and Generate I/O Patterns*. An I/O layout pattern is the property of an I/O workload, which is the key to the application performance (efficiency) and storage health

- Janki Bhimani and Adnan Maruf are with the School of Computing and Information Sciences, Florida International University, Miami, FL 33199 USA. E-mail: {janki.bhimani, amaru009}@fiu.edu.
- Ningfang Mi is with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115 USA. E-mail: ningfang@ece.neu.edu.
- Rajinikanth Pandurangan and Vijay Balakrishnan are with the Memory Solutions Lab, Samsung Semiconductor Inc., R&D, San Jose, CA 95112 USA. E-mail: rajini.panduri@samsung.com, vijay_bala@yahoo.com.

Manuscript received 20 Feb. 2020; revised 31 Aug. 2020; accepted 27 Dec. 2020.

Date of publication 30 Dec. 2020; date of current version 14 Jan. 2022.

(Corresponding author: Janki Bhimani.)

Recommended for acceptance by M. Kandemir.

Digital Object Identifier no. 10.1109/TC.2020.3048303

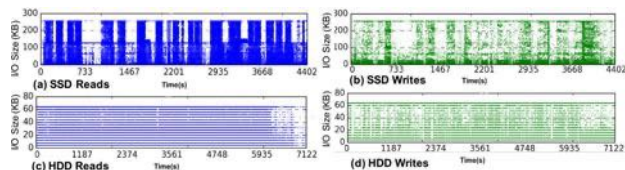


Fig. 1. I/O size of TPC-H-Spark with 50G workload comparing (a) SSD reads, (b) SSD writes, (c) HDD reads, and (d) HDD writes.

(endurance). Multiple dimensions, including disk offset, time, read/write rate (also called data temperature) and I/O size, frame an I/O layout pattern. Each workload may have many different patterns representing different real database activities like compaction and log management. We collect and study I/O patterns of different big data workloads with a different setup for various database server applications using flash-based cloud storage. Specifically, we ran more than 1,000 workloads of each database server application with different SSDs of various capacities (from 100 GB to 1 TB) and different file systems such as xfs and ext4. Our pattern extraction methodology involves a three-step process, i.e., dissect, construct, and integrate. We dissect the overall I/O activities of real workloads to extract distinct repetitive I/O patterns. Then, we identify different input features of an I/O generating engine (e.g., FIO, a popular I/O engine), to construct jobs that generate I/Os resembling different I/O activities of a real workload. We generate unique I/O patterns using various combinations of multiple I/O jobs. We finally construct a pattern warehouse as the collection of these I/O patterns. Different combinations of synthetically generated I/O patterns can reproduce comprehensive characteristics of various real workloads.

2) *Ensure Scalability and Usability*. The second contribution of our work is to make PatIO scalable to generate I/Os over different sizes of storage disks and different storage volumes consisting of multiple SSDs in cloud storage. The user is allowed to specify the storage size and the expected execution time of the desired workload. PatIO can then automatically change all I/O jobs at the low level and modify the necessary input options in I/O patterns on-the-fly for all jobs. To provide an easy-to-use experience, we further develop a graphical user interface (GUI) and an automatic plotting wrapper for PatIO. It decouples the user from the complexity of underlying code modification, integration, compilation, and execution.

3) *Ensure Expandability and Integrability*. We aim to capture a variety of different patterns from samples of I/O workloads that we know in a pattern warehouse. It is also easy to expand our pattern warehouse by adding new patterns based on the new knowledge of applications and workloads. In addition, our I/O generator can be integrated into different environments of the storage industry to fasten research, development, and evolution phases. For example, a possible deployment could be to run PatIO on FIO using FPGAs for next-generation SSD hardware development (e.g., key-value SSDs) or to use PatIO for a firmware configuration such as the proportion of over-provisioning in SSDs.

4) *Practical Application*. We further enhance our framework to evaluate the efficiency and endurance of multi-

stream SSDs. We evaluate the performance of two existing automatic stream assignment algorithms known as auto-stream: SFR and MQ proposed in [6], for different I/O patterns. It helps service providers of cloud storage learn what types of workloads are more benefited by using flash-based multi-stream SSDs. It also helps users of cloud storage to understand if the stream identification is appropriately made, and how their stream assigning algorithms can be further improved to further leverage performance by flash-based SSDs.

5) *Auto-Tuning Module*. Our final contribution is to construct an infrastructure for auto-tuning internal variable parameters of the multi-stream SSDs. In particular, we build a peripheral infrastructure to tune variable parameters for those two existing stream assignment algorithms [6]. In our experiments, we observe that without proper tuning, the benefits of multi-stream technologies may be restrained when some factors like the SSD version, SSD capacity, underlying firmware are changed. Moreover, tuning manually could take a very long time, like a couple of months. Motivated by this, we build an infrastructure on top of PatIO to support automatic tuning for different I/O patterns.

We evaluate our framework by using different containerized workloads running using standalone and simultaneous database applications such as MySQL, Cassandra, and ForestDB. Specifically, we compare I/O characteristics (such as arrival address, I/O size, and read over write ratio), and I/O performance (such as throughput, average latency, tail latencies and Write Amplification Factor (WAF)) of generated workloads with those of real-world workloads. Finally, we discuss the scalability of workloads generated by PatIO to adapt to the SSDs of different capacities.

The rest of the paper is structured as follows. Section 2, discusses the existing techniques. Section 3 presents the PatIO architecture. Section 4 evaluates our technique. The research direction enabled by PatIO towards evolving flash based storage devices is explained in Section 5. We describe and evaluate our auto-tuning module in Section 6. Finally, we draw our conclusions in Section 7.

2 RELATED WORKS

Most benchmarking techniques [5], [7], [8], [9], [10], [11], [12], [13], [14], [15], [16], [17] use samples of proprietary data to first record the overall average statistics (e.g., average I/O rate and average read to write ratio) of real workloads and then reproduce I/Os synthetically based on averages. Such benchmarking tools results in a uniform distribution of I/Os on disk and a constant throughput during the execution. Thus, we argue that although these synthetic I/O generators operate with low overhead and negligible resource requirements, they are not sufficient to capture the working of modern cloud workloads on evolving flash technology. Thus, SSDs behave differently on these traditional synthetic workloads, compared with what they do on real-world workloads.

Apart from widely used synthetic I/O generators, another popular benchmarking technique in the storage industry is *workload replay*. The replay tools [4], [18], [19], [20], [21], record the characteristics of real I/O data for different

TABLE 1

PatIO versus Existing Storage Benchmarking Tools (Bench. - Benchmarking, Req. - Require, Endu. - Endurance, Cap. - Capture, Var. - Variance, Gen. - Generate, Int. - Interface, Across - Acr.)

Bench. Tool	Flash based Bench.	Cap. Var. Acr. SSDs	Cap. Var. Acr. Time	Cap. Endu. of SSDs	Auto Gen. Output Plots	GUI Int.	Req. Trace Logs
PatIO	✓	✓	✓	✓	✓	✓	
Ezpio [14]	✓				✓	✓	
IOA [11]					✓	✓	
Iom. [8]					✓	✓	
SDG. [18]	✓		✓				
hfp. [19]	✓						✓
blkr. [4]							✓
CH [10]	✓				✓		✓
FB [29]	✓		✓				
DB [33]	✓		✓				

granularities like blocks, data chunks, and sectors. By using the recorded logs, these tools can almost exactly replicate I/O activities of real workloads. Recent replay tools [19] have enhanced capability to generate additional data dependency graphs and be able to accurately replay the I/O workload. This technique has high precision. However, capturing all possible workload traces to frame a trace repository is challenging. Storing these traces also demands a large amount of storage resources. One way to generate “real” I/Os with a low storage requirement for characterizing data would be to increase the recording granularity of I/O characteristics and get a short trace that only abstracts the characteristics of a real workload. However, it still requires efforts to run different real application workloads to record traces and needs more storage resources when the number of traces increases.

Application level benchmarks [7], [18], [22], [23], [24], [25], [26], [27] strive to mimic I/O behaviours of specific applications, but require time and efforts for installation, configuration and database loading before running. YCSB [22] is a framework and common set of workloads to evaluate the performance of different “key-value” and “cloud” serving stores. Another widely used database management system (DBMS) benchmark, DBench [23], can evaluate the performance of a plurality of DBMS’s stores both DBMS independent and DBMS specific files in computer memory.

Filesystem level benchmarks [28], [29] spawn several threads or processes doing a particular type of I/O action as specified by the user. They help to answer the trivial question such as, “Which file system is better.” However, our focus is to characterize the performance of SSDs, so it is useful to compare with the benchmarks that report bandwidth and latency when reading from and writing to the disk in various-sized increments without filesystem layer.

Block level benchmarks [19], [28], [30], [31], [32] provide the ability to record and replay block-level I/Os. However, they have heavy overheads to maintain ordering, CPU mappings, and time-separation of I/Os. BlkTrace [30] provided the ability to collect detailed traces from the kernel for each I/O processed by the block IO layer. HFPlayer [19] used the generated dependency graph and can replay the I/O workload in a scaled environment. Buttress [31] used synchronous I/O to replay block traces.

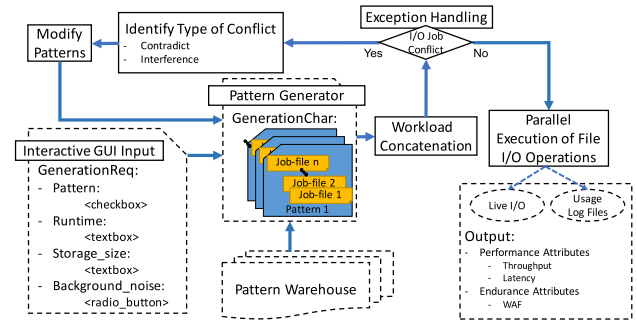


Fig. 2. Block diagram: pattern generation.

In contrast, PatIO does not require to store, read, and follow any I/O trace file while regenerating I/Os. Thus, PatIO is more cost-efficient and less time-consuming compared to existing I/O replay techniques, and more importantly, it is much more precise compared to naive I/O generators (i.e., naive FIO [5]). We finally summarize the features of PatIO and different popular existing benchmarking techniques in Table 1. To the best of our knowledge, this is the first attempt to analyze and improve the impact of variable parameters of the internal algorithms of emerging SSDs such as multi-stream SSDs.

3 PATIO FRAMEWORK

In this section, we discuss the overall architecture and then elaborate on the three components of methodology - dissect, construct, and integrate for PatIO.¹ The driving force of building PatIO is to study the diversities of I/O activities using different data processing workloads on flash-based cloud storage and then regenerate I/O characteristics representing the complex transaction forms like compaction, log management, and key-value store that are performed by different database applications on flash-based SSDs.

In PatIO, we extract the common I/O patterns by observing many different workloads of various applications over different SSDs. We carefully design ready to use I/O workloads to replicate many common I/O patterns observed while running real applications. Particularly, the combination of our I/O patterns replicates the cumulative activities generated by different workloads of any applications. Thus, PatIO strives to capture the common characteristics of a group of similar workloads rather than exactly resembling just one particular workload. PatIO is lightweight, as it does not require to record, store, and retrieve logs of I/O activities. PatIO is also designed to be scalable to generate I/O workloads over different storage sizes. The main contributions and features of our solution are as follows.

3.1 Architecture of PatIO

Fig. 2 shows the architecture of our framework. First, the front end GUI allows the user to configure a workloads’ I/O patterns and its expected execution time and to select the size of the storage disk and the desired level of background noise. The pattern generator then dynamically pulls

1. We use “Disk” interchangeably with “SSD” to represent flash storage throughout this work.

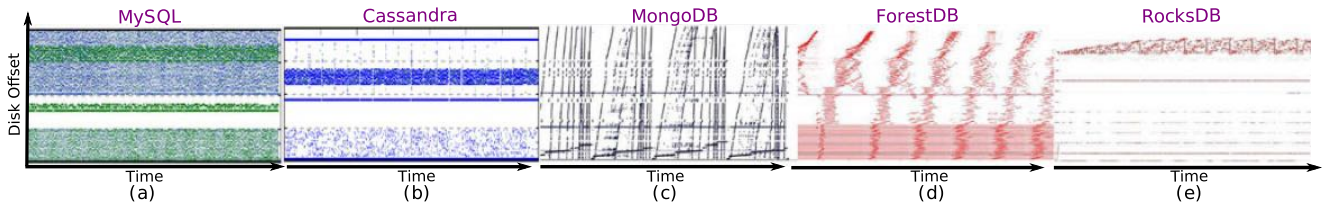


Fig. 3. Disk access patterns over time by real data processing applications.

corresponding files of patterns from the pattern warehouse based on the selected options.

A workload is a combination of single or multiple patterns. Each pattern is a multi-threaded system process and consists of multiple I/O jobs, where a single thread executes each job. Different patterns are executed together to construct a workload. However, before simultaneously executing all jobs of selected patterns, the pattern generator needs to modify the input parameters of these jobs according to the specified execution time and disk size by the user. In addition, the programming commands given by jobs of different processes may have conflicts, e.g., simultaneously writing different values at a specific SSD address. Then, the pattern generator also needs to perform a workload modification to ensure that all job files of the selected workload run correctly during the parallel execution. This whole process of modifying I/O jobs is called workload concatenation.

Fig. 2 shows the process of workload concatenation by a loop of events around *Pattern Generator*, which contains exception handling, identifying conflicts, and modifying patterns. Sometimes, job modification to resolve a conflict may cause new conflicts. Thus, workload concatenation is repeated until there are no more conflicts. Later, in Section 3.4, we explain details about job modification and types of conflicts. Finally, according to the concatenated workload, the I/O engine generates I/Os that will be performed by FIO [5] on back-end SSDs to resemble I/O activities of real parallel applications. *PatIO* also provides detailed reports and graphs to show I/O performance (e.g., throughput, latency, and tail latency) and SSD endurance, such as Write Amplification Factor. All results can be stored as a backup log for future analysis.

3.2 Study of Real I/O Patterns: Dissect

An *I/O Pattern* is a Cluster of I/O Activities of a Workload that has Similar Characteristics. Here, we present our observations on I/O patterns of real applications, which inspires our design of *PatIO*. In our study of the I/O activities of various real applications on SSDs, we consider instrumenting general purpose applications such as Kmeans clustering and PageRank as well as database applications such as MySQL and RocksDB. However, we observed that for generic applications, the processing time is dominated by computation, and intermediate shuffling data generated by them is small and fits in main memory. The data dependencies within such intermediate data that are cached are served from memory. Moreover, these applications perform most of their I/O activities only at the beginning to read the data into memory and at the end to write the final outputs. Thus, we mainly discuss the I/O intensive database workloads whose execution time is dominated by I/O processing.

We study the I/O activities of at least 10 different workloads for each of the 100 different applications on various models and capacities of SSDs. Fig. 3 shows the I/O access patterns for some representative real SQL and NoSQL database applications on SSDs. The workload configurations of these real applications are listed in Table 2. We observe that a real application exhibits variance in I/O activities on SSDs (also called disk) over time. For example, some applications perform their I/Os in a uniform horizontal stripe wise fashion, see MySQL and Cassandra in Figs. 3a and 3b. While some other applications show a periodic pattern of I/O layout over time, such as MongoDB and ForestDB in Figs. 3c and 3d. Also, there exist applications like RocksDB in Fig. 3e that present a horizontal stripe wise pattern. One I/O stripe of RocksDB (see the upper region of Fig. 3e) further exhibits a phase-wise pattern of I/O layout where I/O activities slowly start spreading over the disk and then construct a uniform horizontal strip when the workload has run for a prolonged duration. Thus, the diversity in I/O access patterns motivates us to develop a new I/O generator that can capture these I/O behaviors and dynamically generate I/O workloads for different SSD devices.

First, we *dissect* the overall I/O activities of different user workloads, virtualization setup, file systems, and volume managers for various real database server applications on flash-based storage. We identify the prevailing attributes in distinct visual I/O patterns. The I/O characteristics include I/O sizes, I/O densities, ratios of read to write, and I/O inter-arrival time. We analyze the distribution of these I/O characteristics across different address space of flash-based SSDs and the variation of these I/O characteristics over workload execution time. For example, MongoDB (see Fig. 3c) comprises of different I/O patterns, such as straight horizontal lines representing overwrites on the same disk offset, and inclined vertical lines across the disk representing a form of sequential writes. To extract different I/O patterns, we perform data classification using the K Nearest Neighbour (K-NN) pattern classifier with different distance measures (such as euclidean, Manhattan, Chebychev, and

TABLE 2
Workload Configurations of Different Real Applications (KV - key/value, col. - columns)

Application	Workload	Size	Operations	Type
MySQL	TPCC	4200 warehouse size	100 connections	SQL Transactions
Cassandra	Cassandra stress	10 million records	15 minutes	R/W: 70% /30%, fixed(1024) col.
MongoDB	YCSB	220 million records	50 million operations	100% update, 24/1000-KV
ForestDB	DB_bench	200 million records	10 hours	100% update, JSON objects
RocksDB	DB_bench	560 million records	250 million operations	100% update, 24/1000-KV

TABLE 3
Building I/O Patterns: The Input Features for Jobs of Different I/O Patterns

I/O Pattern	Feature_Set{}
Random I/O with Density Variance (RIDV)	-rate_iops, -offset, -bsrange, -thinktime, -thinktime_blocks
Sequential Writes with Multiple Jobs of Different Offset (SWMJDO)	-rate_iops, -size, -numjobs, -offset, -blocksize_range, -offset_increment
Bars of R/W (BRW)	-rate_iops, -numjobs, -offset, -runtime, -size
Bamboo Sticks Different Slopes (BSDS)	-rate_iops, -offset, -startdelay, -rw_sequencer
Fountain Scatter Horizontal (FSH)	-rw_sequencer, -rate_iops, -numjobs, -offset_increment, -blocksize_range
Bamboo Sticks Horizontal Density Variance (BSHDV)	-rate_iops, -bsrange, -startdelay, -rw_sequencer
Horizontal Overwrites (HO)	-rw_sequencer, -startdelay, -rate_iops, -random_distribution=zipf
Raindrops	-thinktime, -thinktime_blocks, -rw_sequencer, -rate_iops, -numjobs, -offset, -runtime, -size
Sprinkler	-rw_sequencer, -rate_iops, -numjobs, -offset, -offset_increment, -blocksize_range
Bamboo Sticks Vertical Density Variance (BSVDV)	-rate_iops, -offset, -bsrange, -startdelay, -rw_sequencer, -size
Backward Steps (BS)	-rw_sequencer, -rate_iops, -numjobs, -offset, -runtime, -wait_for_previous, -offset_increment, -blocksize_range
Angular Chopping (AC)	-rw_sequencer, -rate_iops, -numjobs, -offset, -runtime, -size
Vertical Chopping (VC)	-thinktime, -thinktime_blocks, -rw_sequencer, -rate_iops, -offset, -runtime, -size
Bamboo Different Alignment (BDA)	-thinktime, -thinktime_blocks, -rw_sequencer, -rate_iops, -numjobs, -runtime, -size
Horizontal Shower (HS)	-rw_sequencer, -rate_iops, -numjobs, -offset

Percent disagreement) and then study classification results with different K in K-NN to distinguish various I/O patterns following the majority.

3.3 Pattern Warehouse: Construct

Pattern warehouse is a collection of I/O patterns used to construct different I/O workloads. Our framework is expandable because we can add new patterns into the pattern warehouse once we obtain the knowledge of other applications and workloads. Our pattern warehouse currently includes 15 different workload patterns. It provides some recommended pattern combinations to resemble real applications, like MySQL, Cassandra, and MongoDB.

Multiple I/O generating jobs constructs each I/O pattern. A real application often exhibits variance in I/O activities across storage space over time. We observe that real workload I/O patterns can be grouped into different categories, such as horizontal stripe wise, periodic, phase-wise, and abrupt. To capture this variance, we develop different I/O jobs. A job is responsible for rendering I/Os for part of the I/O workload pattern to represent some specific I/O layout. Each job is composed of a set of I/O generating features of the FIO engine. Thus, integrating these I/O jobs together can help to capture the diversity of a real I/O workload pattern.

We construct 15 different I/O patterns as listed in Table 3. For example, patterns *RIDV* and *SWMJDO* are of horizontal strip-wise fashion. *BRW* is a horizontal stripe wise with alternative read and write intensive phases. *BSDS*, *Sprinkler*, and *FSH* provide periodic I/O patterns. *BSHDV* is a phase-

wise I/O pattern. *HO* and *Raindrops* both fall under the abrupt category. Some real I/O workloads were observed to have I/O patterns that are a combination of different categories such as RocksDB 3 as discussed in Section 1. In order to replicate such patterns, we further construct five I/O patterns, namely *BSVDV*, *BS*, *AC*, *VC* and *BRW*, that represent the combination of horizontal stripe wise and periodic I/O fashion. The I/O pattern *BDA* is a combination of periodic and phase-wise types, and *HS* is a combination of horizontal stripe wise and phase-wise categories. Thus, we ensure that pattern warehouse consists of all distinct patterns that we majorly observe in real I/O workloads. We can always add new patterns to our pattern warehouse when required.

One of the challenging problems is identifying features that could be used to construct a particular I/O pattern. We solve this problem by studying and analyzing combinations of different features and then setting appropriate values of features for each I/O pattern. Table 3 lists the obtained combinations of features we also use some other common options, such as *-random_number_generator*, *-initial_seed*, *-iodepth*, *-ioengine*, *-rw*, *-device*, *-if-else*, *-for_loops*, *-while_loops*, *-kill_job* to manage runtime operations of I/O jobs. We name I/O patterns according to their visual appearance like *sprinkler*, *raindrops*, *backward steps*.

Next, we explain some representative I/O characteristics that we identify and use to emulate different I/O patterns.

I/O Holes. We observe that many applications do not perform I/Os during some time intervals or within some disk

offset ranges. For example, Fig. 3b, shows the Cassandra workload that has a horizontal blank space band, where no I/Os are performed to a particular disk offset range. We call such a blank space as *I/O hole*. There could be two types of I/O holes, temporal I/O holes and disk offset I/O holes. A temporal I/O hole may be caused by a system stall for waiting for other resources like CPU, I/O bus, or may happen when the upper layers in I/O stack such as cache or memory are sufficient to serve the desired request. On the other side, a disk offset I/O hole may be caused by wear leveling activities or disk space allocation through application transactions. Modeling such I/O holes is critical to performance because during these I/O holes, overall I/O throughput may fluctuate. Furthermore, a benchmarking tool for flash-based SSDs, that can replicate such I/O holes can better estimate endurance. Thus, we capture I/O holes of different shapes and sizes by setting options like `thinktime`, `thinkblocks`, `startdelay`, `offset increment` for each job.

Byte Density. The measure of how many bytes are stored within a particular range of storage addresses is called Byte density. We observe that in many real applications, different disk spaces are accessed with different byte density. For example, when a MySQL database application stores its metadata in some disk space, it might be accessed more frequently than the other disk space. Moreover, we observe that byte density may also vary across different workload execution time. For example, in MongoDB, depending on the keys affected by the “update” operation, it may result in modifying a different number of indexes in the collection. Thus, the number of I/O activities can be sparse or dense depending upon the number of indexes modified during the workload’s execution. It is vital to capture byte density because the variation in byte density is the primary source of I/O latency variations and latency tails. Thus, we use various I/O distributions, such as `zipfian`, `pareto`, `uniform` to capture byte density in each I/O job.

I/O Jumps. A pure sequential I/O should span continuously over consecutive disk addresses. However, we observe that real workloads sometimes leave empty disk addresses between small sequential writes, e.g., skipping 16 KB after writing every 128 KB sequentially. We then say that the I/O patterns of these applications exhibit periodic *I/O jumps* (i.e., addresses left unwritten) while performing sequential reads or writes. Such an I/O jump can allow sequential I/Os to span over a wide range of disk offsets in a short period. I/O jumps result in inclined vertical lines across the disk, as observed in MongoDB, see Fig. 3. We use options like `sequencer` and its `offset` to generate a sequential I/O sequence with I/O jumps.

Pattern Feature Setting. As mentioned above, Table 3 lists all 15 I/O patterns with corresponding list of features for each. Due to the limited space, we cannot explain the logical derivation for deciding the `feature_set` of all patterns in Table 3. Here, we use the pattern, called Bamboo Different Alignment (BDA), as a representative to explain our logical process of `feature_set` derivation. This pattern is inspired by dissecting the MongoDB application when running different YCSB workloads. Fig. 3c, shows the real I/O layout of one of the workloads. We observe a repetitive pattern with a stretch of partially sequential writes over the whole disk space. We say these I/Os as “partially sequential” because they show

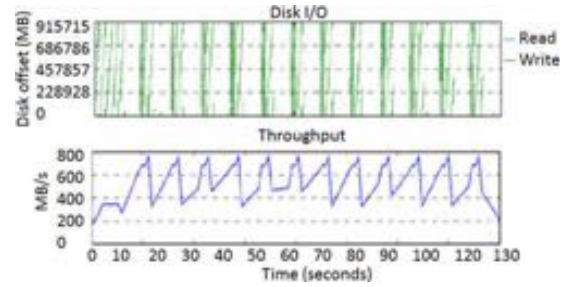


Fig. 4. I/O pattern layout generated by PatIO for Bamboo Different Alignment (BDA) pattern.

an I/O jump after every block of writes to range over the whole disk space in a short period. Apart from that, we consider to use the features `-thinktime` and `-thinktime_blocks` to control periodicity and I/O activities happening in each period. By setting different values for `-rwsequencer` and `-rate_iops`, different slopes of alignment can be achieved. We have preset default values for each of these features within each I/O pattern in the workloads that we design. These values are automatically varied according to the necessary concatenation of multiple I/O patterns, length of the desired workload, and the SSD size.

Fig. 4 shows the resultant BDA pattern for 120 seconds on 960 GB SSD. As seen from the top plot of the figure, the I/O layout consists of periodic I/Os, where each period has a dense region at the beginning followed by the sparser region. Thus, this I/O pattern is constructed by using two jobs. The corresponding features of each job allow it to generate periodic I/Os with different rates to generate denser and sparser regions. Given these two jobs with different I/O rates, we can observe that the throughput (see the bottom plot in Fig. 4) of this generated pattern exhibits variance over time. Such throughput variations well match the throughput variations in real applications.

3.4 Pattern Generator: Integrate

The pattern generator is the central module of PatIO, which is responsible for communicating with the interactive GUI input, pattern warehouse, and I/O execution modules. This module *integrate* different combinations of synthetically generated I/O patterns to reproduce the comprehensive characteristics of various real workloads and system setup for the database server applications. Specifically, the pattern generator gets the user input from the interactive GUI input module. It then fetches the corresponding I/O pattern files from the pattern warehouse. These I/O jobs are then adapted according to the user-specified storage disk size and execution time. Among all the features of the jobs, we first identify a subset of features that could be affected by the change in disk size. Then, the features in this subset are modified by a linear scaling, as shown in Equation (1). For example, *I/O range* which is set to 400-500 GB for 500 GB drive is changed to 800-1000 GB for 1 TB storage disk size.

$$New_Feature_i = default_Feature_i \cdot \frac{New_Size}{default_Size} \quad (1)$$

for \forall Jobs i .

Similarly, the execution time of each job for all the patterns needs to be changed according to the desired execution

time given by the user. Finally, we execute jobs of all selected I/O patterns in parallel.

Online Conflict Management. For parallel execution, some jobs may have conflicts with others. As discussed before, a pattern is executed as a multi-threaded system process. When multiple patterns are required to execute simultaneously, programming commands that are given to the I/O generating engine by one process's threads may affect threads of other processes. Thus, before executing all the jobs of selected patterns, the pattern generator performs careful workload concatenation of all the job files. It is essential to identify and handle these conflicts. Our exception handling module identifies conflicts by maintaining a hash table of features and I/O jobs. If there is a conflict, those jobs are modified according to the type of conflict. The modified I/O jobs are then concatenated again until there are no further conflicts. Here, we use two common types of conflicts as examples to show corresponding modifications performed to resolve them.

Contradiction. Jobs of different patterns might set different values of the same feature. We call this type of conflict as *contradiction*. For example, one job might request I/O engine to set I/O size feature to 4K for a particular disk offset. However, at the same time, another job of a different pattern might want to set I/O size of 64 KB for the same disk offset. For FIO, we notice that both of these I/O jobs may stall for a long time or be dropped off when such a contradiction occurs. We resolve this contradiction by introducing some time delay between the operation of these two jobs. As a result, in the above example, we allow the first job to perform 4 KB I/Os and later let the second job run its 64 KB I/Os on the same disk offset.

Interference. Some actions taken by a job of one pattern might unintentionally influence jobs of the other patterns. For example, a `killall` command in a job of `pattern_x` might also kill jobs of all the other concatenated patterns. Thus, we need to maintain a list of such features and identify if any jobs are using these features. If yes, then we need to modify these jobs to ensure such an interfering command only affects the jobs of the desired pattern. That is, we would identify the thread ID of the jobs of each pattern and kill only the threads of the concerned pattern rather than using the default `kill all` command. The types of conflicts and their resolutions may vary with different I/O engine. However, it is crucial to observe such behavior as it vastly impacts I/O layout on disk.

Parallel Executor. After resolving all conflicts (i.e., no more conflicts in the concatenated workload set), we execute the generated synthetic I/O workload and measure the performance of I/O activities over the storage space. All the workloads generated by our framework are capable of generating logs during the runtime and record the performance in terms of I/O bandwidth, IOPS, throughput, and latency.

3.5 GUI Interface and Process of Using

In order to provide an easy-to-use experience, we develop a GUI interface for `PatIO`. We mainly have two use scenarios - 1) if the user wants to generate the I/O activities for one of our pre-defined applications. As of now, we provide a direct option to generate I/Os resembling the five most popular database applications such as MySQL, Cassandra,

TABLE 4
Hardware Configuration

CPU Type	Intel(R) Xeon(R) CPU E5-2640 v3
CPU Speed	2.60 GHz
CPU #Cores	32 hyper-threaded
CPU Cache Size	20480 KB
Main Memory	128 GB
OS	Ubuntu 16.04 LTS
OS Kernel Version	4.4.0-13generic
File System	ext4
Storage	No-stream and Multi-stream NVMe SSD 960GB and 480GB
Docker Version	1.11
VMware Workstation	12.5.0
FIO Version	2.2

MongoDB, ForestDB, and RocksDB. The user can use corresponding checkboxes to select one or multiple of these real applications in our GUI. 2) if the user wants to generate the I/O activities for some other applications. Then, we assume that the user should have some idea from their experience or plots of previously collected traces that what type of I/Os are they looking to generate. Depending upon their requirements, they can select one or multiple I/O layout patterns in our GUI, e.g., horizontal overwrites, bars of read/write, and backward steps. We have 15 different patterns with a visual snapshot of the disk layout for each to choose from in our GUI. Then the user defines the desired runtime (i.e., execution time) of an I/O workload and the size of the storage space exposed to I/Os. Additionally, the GUI also allows selecting a different level of background noise, which may be incurred by various background I/O activities of the SSDs such as garbage collection, wear-leveling, etc. Finally, the user clicks run, and `PatIO` accordingly generates I/Os, instruments the performance, collects traces, and plots various generally used graphs such as average throughput over time, the cumulative distribution of the tail latencies, and instantaneous write amplification factor of SSD. Thus, besides taking the input options of the desired workload, the GUI is also responsible for linking an option in the widget with its corresponding *PatternID* and send this option to the back-end pattern generator module.

4 EVALUATION

In this section, we evaluate `PatIO` by comparing I/O characteristics and performance of generated workloads with real-world workloads of different database applications such as MySQL, Cassandra, and ForestDB. Table 4 gives the detailed hardware configuration of our platform on which we develop `PatIO` and run real application workloads. `PatIO` is built using python. It uses inbuilt advance libraries of python like matplotlib, NumPy, and Tkinter. Each pattern in pattern warehouse contains a bash program that

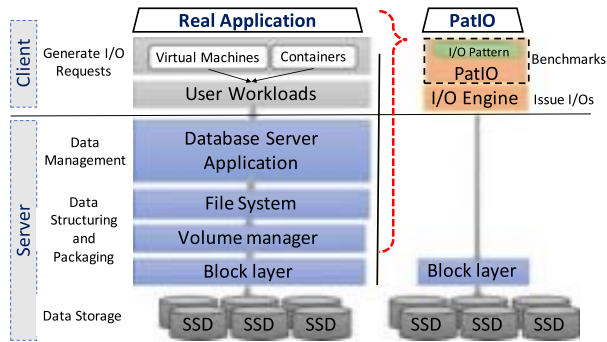


Fig. 5. I/O stack of PatIO in comparison to that of real application.

can be used to construct I/O jobs. We use the FIO engine to generate I/Os.

Fig. 5 shows the I/O stack of PatIO in comparison to that of real application. As shown in Fig. 5, PatIO generates an I/O pattern that can capture real I/O activities (see the left part in the figure) of different user workloads (e.g., YCSB) running in docker containers on the client-side for various database server applications (e.g., MongoDB) running in the datacenter on the server-side that can use different file systems (e.g., ext4) and volume managers (e.g., LVM). More importantly, the operations of different database server applications using various cloud setup of user workloads and system settings at the file system and volume manager layers are abstracted by PatIO. Thus, PatIO requires less time and resources for benchmarking.

We study the I/O activities of different user workloads and applications (e.g., YCSB, Cassandra-stress, and DB_bench) running on parallel virtual machines and containerized infrastructures with different database servers (e.g., MySQL, Cassandra, ForestDB, MongoDB, RocksDB) operating in the data center. As shown in Table 2, each application workload can have its configuration of the number of transactions, compaction rate, and read-write ratio. We also study different combinations of applications operating directly on the local machine and in containerized docker environments, e.g., MySQL+Cassandra with a different number of containers for each application.

4.1 Characteristics Comparison

First, we compare the characteristics of a real workload and a synthetic workload generated by PatIO by measuring their statistical central-tendency like Mean, Median, and Mode. We also compare the spread of data from the central tendency, such as standard deviation and coefficient of variance.

Workload Characteristics. Different characteristics are observed from a workload such as I/O layout on a storage disk, I/O size, and read-write ratio. We perform experiments with 1,000+ workloads of different applications. As a representative, we here present results for some of them. The configurations of real applications are given in Table 2. Fig. 6a compares I/O arrivals on disk space over time for real and PatIO workloads. We see that the statistical results of central tendency (like Mean and Median) for real and PatIO workloads are very similar. Here, we use unit positive and negative standard deviation to measure the spread of data from the central tendency and confidence in statistical conclusions. We also observe that the real and PatIO

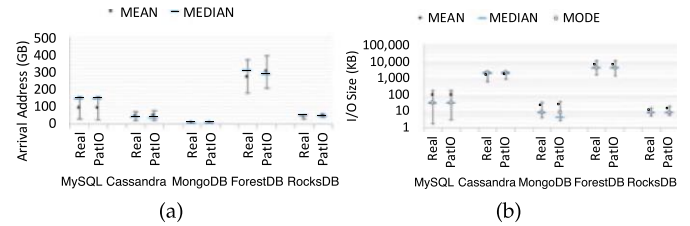


Fig. 6. (a) Mean, Median, and Standard Deviation of I/O arrivals on disk space over time for real and PatIO workloads, (Note: Mode for Arrival Address is not plotted as I/Os to same block does not necessarily imply I/Os to the same address.) and (b) Mean, Median, Mode, and Standard Deviation of I/O Size for real and PatIO workloads. (Note: y -axis is logarithmic scale, so Standard Deviation does not look to be equally distributed on either side of mean).

workloads of all the applications show similar standard deviations. Fig. 7 compares the coefficient of variance of I/O arrivals on disk space over time for real and PatIO workloads. We see mostly error between real and PatIO workloads remains small.

I/O Size. I/O size is another important characteristic that affects performance. Because the sizes of I/Os vary over time for a real workload, just reproducing I/Os with the same size (e.g., average I/O size) is not sufficient. Thus, we argue that it is critically important to emulate the variance of I/O size over execution time. Fig. 6b shows the comparison of the statistics of I/O size over time for real and PatIO workloads. Besides Mean and Median, we further use Mode to represent the size of the majority of I/Os. We can see that the modes of real and PatIO workloads also match well in Fig. 6b. We further observe that PatIO can reconstruct the variance of a real workload as seen from the standard deviation and coefficient of variance. While comparing all different metrics of measurement, the maximum error percentage is less than 25 percent, which indicates a good resemblance between real workloads and PatIO.

Read-Write Temperature. Besides the above statistics comparisons, we also compare the characteristics over the runtime of real and PatIO workloads. Fig. 8 shows the read to write ratio over runtime as a representative by plotting the moving averages taken over every 30 seconds until 15 minutes. Here, we show the results of MySQL and Cassandra. We see that the generated workload can reproduce the actual read/write temperatures.

4.2 Performance Comparison

Throughput. We further compare the throughput (i.e., the number of I/Os performed on disk per second) of the aggregate generated workload using PatIO over variants of 1,000

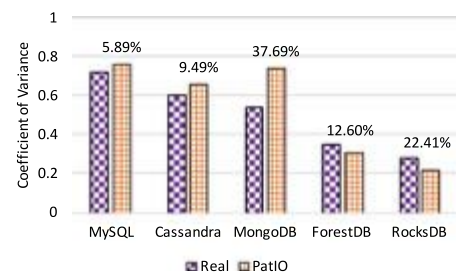


Fig. 7. Coefficient of variance of I/O arrivals on disk space over time for real and PatIO workloads with the error (%) mentioned above the bars.

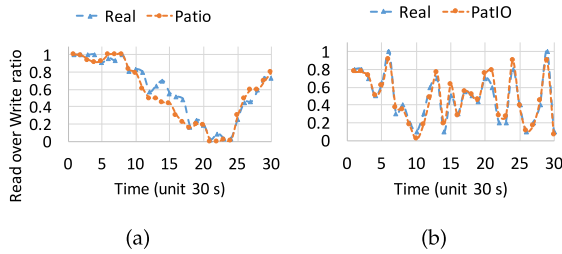


Fig. 8. Read to write ratio over runtime of workloads a) MySQL and b) Cassandra.

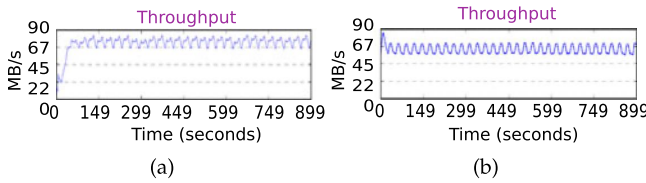


Fig. 9. Throughput variation over time for, a) Real Cassandra workload and b) Generated Cassandra workload.

different workloads of Cassandra with the throughput of a sample chosen randomly while running real application with these 1,000 different workloads. The goal of our `PatIO` generated workload is to capture the high-level I/O feature that persists among multiple workloads of an application. The throughput results in Fig. 9 shows that the real workload does not have a constant throughput over the execution. Our synthetic workload shows the same wavy nature of the throughput (see Fig. 9), as opposed to the constant throughput that is resulted by Naive FIO. We notice that the throughputs in the first few seconds are different because the real workload needs to spend some time to initial cache construction while the generated workload assumes that data required by the application is readily available. Also, for a real application, there are more intermediate layers in I/O stack when compared to `PatIO`, as seen from Fig. 5, which increases the initial latency and decreases the initial throughput. Once the cache is constructed, this latency due to intermediate layers is hidden in parallel tasks. However, the initial performance during this short period is often neglected because it is well known that all storage disks require some initial time for ramp-up. We also observe that the real Cassandra result has varied across time, and `PatIO` can exhibit similar performance variations.

4.3 Overall Validation

We consider the cross correlation² to compare the performance of running workloads generated by `PatIO` with respect to that of running real workloads. We vary the total number of operations (i.e., transactions) performed to generate 50 different workloads of each application. Fig. 10 shows the average of cross-correlations between different real and the corresponding generated 50 workloads while running individual applications (e.g., Cassandra and MySQL) and mixed applications (e.g., MySQL+MongoDB). As a baseline, we also plot the cross-correlation of the workloads generated using naive FIO generator. The naive FIO uses the Mean of different workload characteristics.

2. Cross correlation is a measure of similarity of two series as a function of the displacement of one relative to the other.

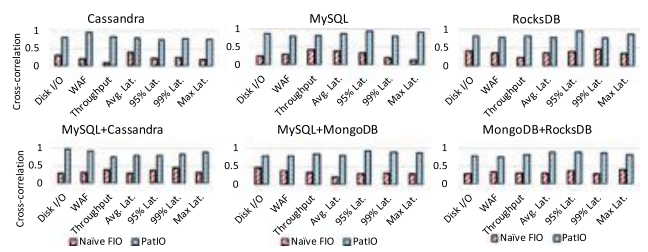


Fig. 10. Comparing the cross-correlation between real and synthetic workloads using a lag of number of samples used for training for traditional FIO that uses average statistics (Naive FIO), and `PatIO`.

TABLE 5
Comparing the Operational Storage Space (MB) Consumed and Execution Time (minutes) for Replay [30], and `PatIO`

	Cassandra		MySQL		RocksDB	
	(MB)	(min)	(MB)	(min)	(MB)	(min)
Replay	64,512	420	48,128	360	90,112	600
<code>PatIO</code>	13.87	9.04	10.42	6.38	18.11	13.51

Each plot in Fig. 10 shows the correlation of the Disk I/O distribution, the endurance SSD measured using Write Amplification Factor, and the performance measured using throughput, average latency, and tail latency. To measure WAF during runtime, periodically, we instrument physical NAND writes of SSD by using the Self-Monitoring, Analysis, and Reporting Technology (S.M.A.R.T) tool commands such as "nvme smart-log" and "smartctl" [34], and instrument the logical writes to each physical SSD from block layer. We do not propose any SSD firmware amendments in this work. All the other performance matrices are measured at the application layer. We observe that the synthetic workloads generated by our `PatIO` are highly correlated with real ones, with the cross-correlation large than 0.8 for all the workloads. The naive FIO has a comparatively lower correlation than `PatIO` because workloads generated by the naive FIO fail to capture the intrinsic diversities and variations of real applications.

Table 5 shows that when compared to traditional I/O replay tools [30], `PatIO` consumes much smaller amount of operational storage space and execution time. This is because `PatIO` is designed to emulate and regenerate the characteristics of different real workloads rather than storing time logs of I/O activities. Also, the architecture of `PatIO` bypasses many intermediate I/O stack layers as shown in Fig. 5, which allows it to execute much faster.

4.4 Scalability of `PatIO`

One of our contributions is that `PatIO` is scalable to generate I/Os over different sizes of storage disks. Thus, we use the Horizontal Shower (HS) I/O pattern (see Table 3) as an example to investigate `PatIO`'s scalability. Fig. 11 shows I/O characteristics (e.g., I/O distribution and I/O starting disk offset) and performance (e.g., throughput) when running the generated HS workload in SSDs with different capacities, i.e., (a) 480 GB and (b) 960 GB. First, we observe that the I/O workloads generated by `PatIO` scale with the

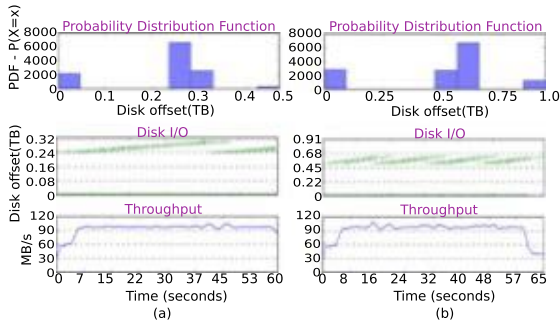


Fig. 11. Effect of drive size: a) 480 GB SSD and b) 960 GB SSD.

SSD capacity in terms of I/O layout, see the first two rows in Fig. 11. We also observe that the throughput remains the same under two different capacities as expected, because the workload does not saturate the I/O bandwidth on both SSDs. Summarizing, workloads generated by PatIO behave similarly as real ones in terms of both I/O characteristics and performance for back-end storage of different sizes.

5 PRACTICAL APPLICATIONS

Multi-stream SSD [2], [3] is the recent development of the SSD industry, which allows us to have multiple append points (erase blocks) at the same time while writing to an SSD. This advancement assists in placing data on SSDs in a smart way such that we have less garbage collection and hence less write amplification factor. Multi-stream functionalities are added to Linux mainline kernel in 2017 with the corresponding NVMe drivers available online.³ To take advantage of the features offered by multi-stream SSDs, it is challenging to identify the stream IDs. For each write/update, a stream ID needs to be assigned such that each stream consists of data with a similar lifetime of being valid. [6] proposed two automatic stream management algorithms (auto-stream), named *SFR* and *MQ*, to assign stream IDs. The *SFR* algorithm utilizes three attributes (i.e., sequentiality, frequency, and recency) for stream detection. It maintains the rank of data blocks in a table and allocates stream IDs according to the rank. The *MQ* (Multi-Queue) algorithm utilizes access frequency and recency to maintain multiple queues and uses each queue to represent the rank of data. The calculated rank is used to assign stream-IDs to data. In [6], it is observed that the benefit of these stream management algorithms varies over different workloads. However, it is difficult to study what type of applications may be benefited by which stream assignment algorithm, because there are many varieties of applications and different possible combinations of the variable parameter within each stream assignment algorithm. Moreover, quantifying the aggregated performance benefit over the wide range of the performance parameters, such as high throughput, low latency, high SSD endurance,⁴ and low write amplification factor may consume a large amount of resources and require a lot of time and efforts. As PatIO can mimic the

3. <https://elixir.bootlin.com/linux/latest/source/drivers/nvme/host/core.c>

4. SSD endurance is the total amount of data that an SSD is guaranteed to be able to write under warranty, and high SSD endurance indicates high device lifetime.

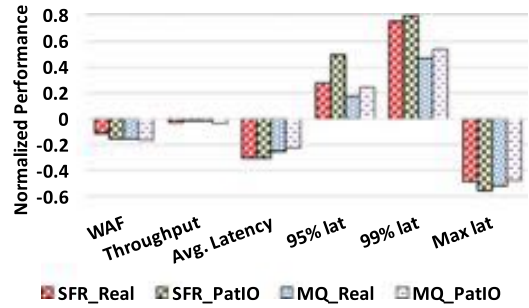


Fig. 12. Comparing the application performance while using real and generated workloads running Cassandra application.

I/O behavior of the real application with negligible spatial and temporal overhead (as discussed in Section 4). Thus, we next discuss how PatIO can be integrated with the multi-stream SSDs to help study the performance variation of each stream identification algorithm with respect to various applications.

5.1 Integrating PatIO With Multi-Stream SSDs

PatIO can be used to study the impact of any particular software, firmware, or hardware improvements over multiple facets such as throughput, latency, and write amplification factor. Particularly, first, we install the new firmware and hardware of multi-stream SSDs in our server. Next, we ensure that the block layer of the Linux OS is able to read and write to the multi-stream SSDs through the NVMe device driver. Finally, we run PatIO on the client-side to issue I/Os directly to the block layer of the server, as shown in Fig. 5. We use multi-stream NVMe SSD and no-stream legacy NVMe SSD of the same capacity of 480 GB. We measure application performance in terms of average throughput, average latency, and different tail latencies. We also measure SSD endurance in terms of the average of write amplification factor calculated using the ratio of the total physical NAND writes to the logical application writes within 5-minute intervals.

5.2 Performance of Auto-Stream

We first compare the performance of the workload generated by PatIO to the performance of the real workload. Fig. 12 plots the normalized performance results (i.e., (multi-stream - legacy)/legacy) of a multi-stream SSD using *SFR* and *MQ* under both real and PatIO generated workload of the Cassandra application. The results using legacy SSD without streaming are considered as the baseline. We use the workload configuration of Cassandra as mentioned in Table 2 for the legacy NVMe SSD (no streaming) and multi-stream NVMe SSD to obtain the “real” performance results. The positive bars in Fig. 12 reflect that using a multi-stream SSD, the measured performance is higher than that of a legacy SSD and vice-versa for negative bars. Our analysis across different performance metrics helps us examine if the generated workload by PatIO shows the same traits as a real workload. We observe that PatIO is able to capture the performance trend of either improvement or deterioration exactly. That is, while using a real application, if using multi-stream SSD with *SFR* or *MQ* algorithm resulted in performance improvement, then

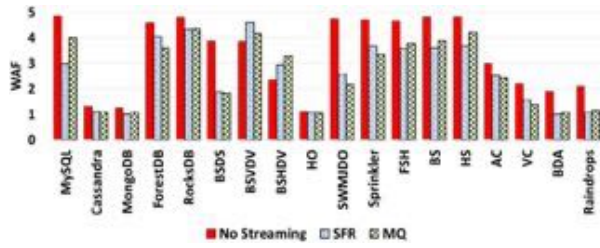


Fig. 13. Comparing WAF of legacy drive with a multi-stream drive using SFR and MQ stream assignment algorithms.

similar performance improvement is also seen with the workload generated by PatIO. Fig. 12 shows that our generated workload closely mimics all performance metrics. For example, WAF decreases with both the auto-stream algorithms of multi-stream SSDs compared to the no-streaming legacy SSDs for the real Cassandra workload. We can see the same impact from the PatIO generated Cassandra workload. Thus, PatIO is useful to analyze the impacts of evolving storage technologies such as auto-stream.

We next use PatIO, for running individual I/O patterns listed in the Table 3 to study in-depth performance benefit of multi-stream SSDs with SFR and MQ algorithms over legacy SSDs (no streaming) for each I/O pattern. We also combine different I/O patterns to emulate various real applications and study the performance impact. Fig. 13 shows write amplification factor for 5 different workloads, such as MySQL, Cassandra, MongoDB, ForestDB and RocksDB, and 13 other I/O patterns (in Table 3). We observe that multi-stream SSD with both the SFR and MQ algorithms can reduce WAF for most of the workloads compared to legacy. Lower WAF means less internal writes during garbage collection, which leads to better SSD endurance. Thus, we say that multi-stream technology may improve SSDs’ lifetime. There are two exceptional I/O patterns, such as BSVDV and BSHDV, under which using multi-stream SSD incurs an increase in WAF. More importantly, our PatIO helps to identify such exceptions. This also shows that PatIO has great potential to help the specialist in the design of multi-stream SSDs and auto-stream algorithms to improve their algorithm, firmware, or hardware in order to handle such exceptions.

Fig. 14 further indicates that apart from increasing the lifetime of SSD, multi-stream technology may also help reduce the latency of workloads. We see that different I/O patterns have different impacts of streaming on their latency. Both SFR and MQ achieve lower latency for some

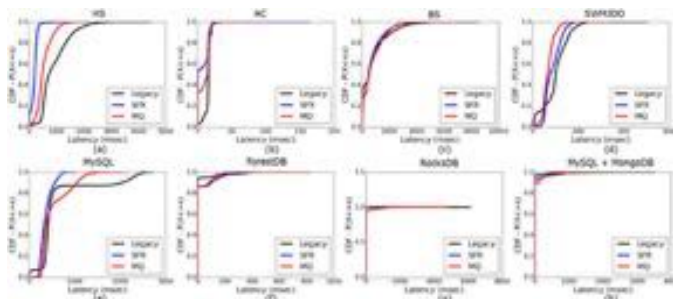


Fig. 14. Latency Cumulative Distribution Function (CDF) of different workloads using a legacy SSD and multi-stream SSD with stream assignment by using SFR and MQ algorithms.

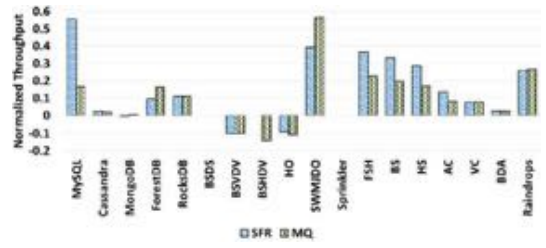


Fig. 15. Normalized throughput of multi-stream drive using SFR and MQ stream assignment algorithms with respect to legacy.

patterns, like HS and SWMIDO. While for some others like AC and MySQL, SFR does a better job than MQ. In addition, SFR or MQ might be good for some individual workloads, but not be that beneficial when we have multiple simultaneous workloads, like MySQL+MongoDB.

Finally, we analyze application throughput while using legacy (no-streaming) SSD and multi-stream SSD. Fig. 15 shows the normalized throughput of SFR and MQ with the throughput of legacy as a baseline. It depicts that among different generated application workloads and I/O patterns, most of them have better throughput while operating on multi-stream SSD than legacy SSD. However, some particular I/O patterns (like BSVDV, BSHDV, and HO) need special attention. The workloads with these patterns may not show a good response to multi-stream technology. They may need some modifications to improve their underlying stream assignment. Note that for I/O patterns BSDS, and Sprinkler the throughput using multi-stream and legacy SSDs is the same, thus the normalized throughput is shown in Fig. 15 is null.

5.3 Benefits of Performance Engineering With PatIO

To sum up, PatIO can explore a wide range of different workloads in a short time compared to the time required in real application installation, configuration, and running. PatIO can assist us in comparing different stream assignment algorithms using I/O patterns to evaluate the impacts of any algorithm, firmware, or hardware modifications on various performance metrics. Additionally, PatIO can identify intricate details, e.g., which I/O pattern in a workload which is responsible for performance degradation.

6 AUTO-TUNING MODULE

Our evaluation found that the benefits of multi-stream SSDs over legacy varied with the SSD version and the capacity of multi-stream SSDs. Indeed, [6] presents that both two-stream assignment algorithms (i.e., SFR and MQ) have a set of tunable parameters, such as *chunk_size_sector*, *decay_sec* and *freq_aging_sec* for SFR and *chunk_size_sector* and *adjust_ref_cnt* for MQ. We believe that to achieve optimal performance, these variable parameters need to be properly tuned for a change in the physical firmware like the size or the model of an SSD drive. Furthermore, while identifying the best value for these parameters, different I/O applications should be considered to avoid over-fitting to any particular workload. Additionally, different users may have requirements on different performance parameters, e.g., latency, throughput, or write amplification. Thus, it is also important to take these user requirements into account

when tuning variable parameters. Let the parameters for any particular algorithm (e.g., SFR) be $\alpha_k, \beta_k, \gamma_k$, etc. Examples of performance parameters that we desire to measure and improve are 50th, 90th, 99th, and 99.99th percentile of latency (Δl), WAF (ΔW) and throughput (Δt). Weight factor ω_i is further used to differentiate the importance of each performance parameter. Eq. (2) represents the weighted sum of different performance parameters.

$$W_j = \frac{-\omega_1 \Delta l_{50} - \omega_2 \Delta l_{90} - \omega_3 \Delta l_{99} - \omega_4 \Delta l_{99.99} - \omega_5 \Delta W + \omega_6 \Delta t}{\omega_1 + \omega_2 + \omega_3 + \omega_4 + \omega_5 + \omega_6}, \quad (2)$$

where, $\forall i \in [0, +\infty)$ and $\omega_i \in [0, +\infty)$. Thus, Eq. (3) shows that the resultant performance parameters are the function of the internal algorithm parameters of multi-stream SSDs.

$$f(\alpha_k, \beta_k, \gamma_k, \dots) = W_j, \quad (3)$$

where $\forall \alpha_k \in [0, Max_\alpha], \beta_k \in [0, Max_\beta]$, and $\gamma_k \in [0, Max_\gamma]$.

Finally, we want to maximize the weighted "Reward Function" for n workload as shown in Eq. (4).

$$W_{avg} = \frac{1}{n} \sum_{j=1}^n W_j. \quad (4)$$

Objective.

Find the best $\alpha_k, \beta_k, \gamma_k$, etc. such that,

$$g = Max \left\{ \frac{1}{n} \sum_{j=1}^n W_j = f(\alpha_k, \beta_k, \gamma_k, \dots) \right\}. \quad (5)$$

6.1 Techniques for Solving Optimization Problem

The simplest approach towards solving this convex optimization problem is the brute force evaluation of the objective function by generating all possible $\alpha_k, \beta_k, \gamma_k$, etc. This method gives us the best possible internal parameters, which ensures the best performance and endurance of the storage device. However, due to the high complexity of the problem, such a brute force approach is time-consuming. Thus, we solve our optimization function by using the multi-armed bandit model of reinforcement learning to auto-tune parameter values that can result in a local maxima.⁵ Our *agent* is an internal algorithm (e.g., SFR) of a multi-stream SSD, which needs to take *actions* towards the best performance by either increasing or decreasing the value of each variable parameter. For example, SFR has *three-arms*, as it contains three variables. The *reward* of each action is calculated using the reward function g (see Eq. (5)). Our *agent* aims to maximize the reward until it encounters the local maxima.

We build the required peripheral framework shown in Fig. 16 with PatIO to enable auto-tuning. This framework is constructed to explore the best values of tunable parameters for an auto-stream algorithm, such as SFR and MQ. This framework particularly uses the same SSD in the legacy

5. A real-valued function f defined on a domain X , is said to have a local (or relative) maximum point at the point x^* if there exists some $\epsilon > 0$ such that $f(x^*) \geq f(x)$ for all x in X within distance ϵ of x^* .

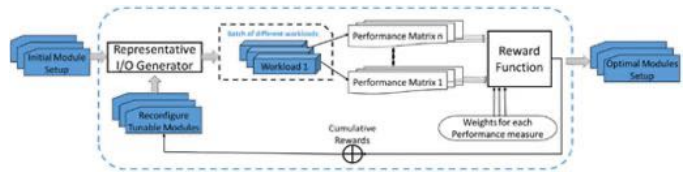


Fig. 16. Auto-tuning the variables of stream assignment algorithm.

mode (i.e., without stream) and then runs an auto-stream algorithm in the multi-stream mode to measure the performance changes between legacy and multi-stream. The goal is to identify the values for variable parameters of that auto-stream algorithm in order to obtain the best possible performance using multi-stream SSDs. Our framework uses PatIO to generate $\binom{15}{1} + \binom{15}{2} + \dots + \binom{15}{15} = 2^{15}$ different workloads with 15 I/O patterns shown in Table 3 to ensure that the parameter tuning is not biased or over-tuned towards any particular workload. The initial value and the minimum and maximum of each variable parameter are defined in the "Initial Module Setup". Then, as shown in Fig. 16, PatIO reads the constraints and initial values from the initial setup file and generates a set of different workloads. These workloads are run one by one to get their own performance metrics that consist of various performance parameters like throughput, average latency, tail latency, WAF.

6.2 Results

To evaluate the effectiveness of our parameter tuning module, we first compare the time consumed in reaching the convergence point and the maximum reward after convergence while using the brute-force method and our auto-tune module. We run the experiments five times and take the average to present the results. Fig. 17a shows that the time for auto-tuning to reach local maxima is much shorter than the time required by brute-force to reach global maxima. From Fig. 17b, we see that difference in the achieved maximum reward with brute-force and auto-tuning is not significant. Thus, using our auto-tune module along with PatIO, we can quickly improve the performance and endurance of multi-stream SSDs.

We further analyze the variations of the rewards over runtime when using our auto-tune module. Fig. 18 shows the average reward value (W_{avg}) of the initial point (I), two intermediate points (II, III), and the final converged point (IV), for SFR and MQ. The variable parameter values for each of these points are further listed in Table 6. We notice the overall performance reward (W_{avg}) increases 60 percent for SFR and 110 percent for MQ from the initial point (I) to the final converged point (IV) in Fig. 18.

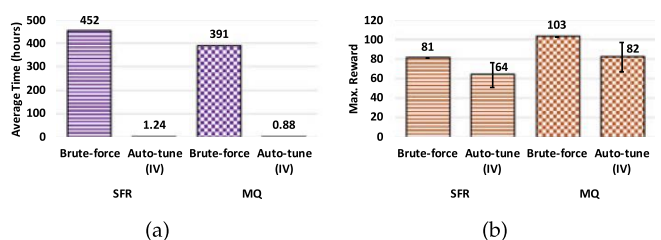


Fig. 17. Comparing Brute-force and auto-tune converged points (a) Average Time and (b) Maximum Reward.

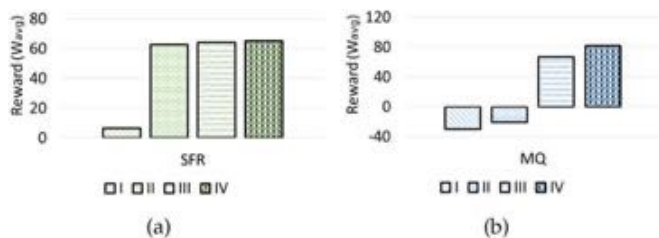


Fig. 18. Average Rewards while auto-tuning variable parameters of (a) SFR and (b) MQ.

TABLE 6
Intermediate Points While Auto-Tuning

Label	SFR	MQ
I	chunk_size=4096, decay_sec=1200, freq_aging_sec=36000	chunk_size_sector=2048, adjust_ref_cnt=2
II	chunk_size=1024, decay_sec=300, freq_aging_sec=18000	chunk_size_sector=2048, adjust_ref_cnt=4
III	chunk_size=2048, decay_sec=100, freq_aging_sec=10000	chunk_size_sector=1024, adjust_ref_cnt=8
IV	chunk_size=2048, decay_sec=2000, freq_aging_sec=42000	chunk_size_sector=512, adjust_ref_cnt=6

We note that besides PatIO, our auto-tune module is also complementary to other synthetic workload generators such as FIO and trace-based I/O generators such as block trace replay. However, we observe that the internal parameters selected using simple synthetic workloads are not good enough to obtain better performance when running a real application on these multi-stream SSDs. On the other hand, the storage space and time required to use trace-based I/O generators to configure the internal parameters of multi-stream SSDs are very high and not always feasible. Thus, compared with those existing synthetic workload generators and trace-based I/O generators, our PatIO provides a more realistic and highly usable solution. Using our auto-tune module with PatIO, our performance engineering team was able to significantly reduce the configuration time of multi-stream SSDs from a couple of months to a couple of hours. Our novel approaches of (a) benchmarking with the combined activities of multiple workloads of any application and (b) tuning the internal variables of the modern SSDs to further improve the performance are very important to the storage community with new emerging storage devices.

7 CONCLUSION

To comfort benchmarking in a modern cloud storage with flash-based SSDs, we develop PatIO, which can generate I/Os that closely resemble real data processing workloads. PatIO captures the common characteristics of a group of similar workloads rather than exactly resembling one particular workload. PatIO thus can resemble a wide range of realistic I/O workloads. PatIO is also lightweight, as it does not require to record, store, and retrieve logs w.r.t. the

timestamp of various I/O activities. We developed a GUI interface for PatIO to make it easy to use. We evaluated PatIO by comparing workload characteristics and performance of synthetic workloads with real workloads on the same system platform. We currently have 15 different I/O patterns in our pattern warehouse that are capable of reproducing 1,000+ real workloads. We also deployed PatIO to two auto-stream algorithms and evaluated the current advancement of multi-stream technology in terms of its benefits to application performance and SSD endurance. Finally, we proposed a practical technique to automatically tuning variable parameters of the existing stream assignment algorithms for any change in storage capacity or SSD models. In the future, we plan to extend our PatIO warehouse to add new patterns capturing I/O activities of the other compute intensive workloads, changes in system parameters, such as NVM write buffer size, queue depth, and garbage collection algorithm. We plan to design a module that automatically identify important common characteristics and accordingly self-generates new I/O patterns from a set of given workloads using machine learning techniques in combination with statistical computations. We also plan to explore other global convergence techniques that may incur lower overhead and guarantee better performance.

ACKNOWLEDGMENTS

This work was initiated during Janki Bhimani's internship at Samsung Semiconductor Inc. [1]. This work was supported in part by the National Science Foundation Awards CNS-2008324 and CNS-2008072 and in part the National Science Foundation Career Award CNS-1452751.

REFERENCES

- [1] J. S. Bhimani, R. Pandurangan, V. Balakrishnan, and C. Choi, "Methods and systems for testing storage devices via a representative I/O generator," U.S. Patent App. 15/853,419, Mar. 21, 2019.
- [2] J.-U. Kang, J. Hyun, H. Maeng, and S. Cho, "The multi-streamed solid-state drive," in *Proc. 6th USENIX Conf. Hot Topics Storage File Syst.*, 2014, Art. no. 13.
- [3] J. Bhimani *et al.*, "FIOs: Feature based I/O stream identification for improving endurance of multi-stream SSDs," in *Proc. IEEE 11th Int. Conf. Cloud Comput.*, 2018, pp. 17–24.
- [4] Blkreplay. Accessed: Aug. 21, 2019. [Online]. Available: <http://manpages.ubuntu.com/manpages/xenial/man1/blkreplay.1.html>
- [5] J. Axboe, "Fio-flexible I/O tester synthetic benchmark," Accessed: Jun. 13, 2015. [Online]. Available: <https://github.com/axboe/fio>
- [6] J. Yang, R. Pandurangan, C. Choi, and V. Balakrishnan, "Auto-Stream: Automatic stream management for multi-streamed SSDs," in *Proc. 10th ACM Int. Syst. Storage Conf.*, 2017, Art. no. 3.
- [7] T. G. Armstrong, V. Ponnakanti, D. Borthakur, and M. Callaghan, "LinkBench: A database benchmark based on the Facebook social graph," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2013, pp. 1185–1196.
- [8] I/O meter. Accessed: Jan. 9 2019. [Online]. Available: <http://www.iometer.org>
- [9] General tips on disk benchmarking. Accessed: May 2, 2016. [Online]. Available: <http://beyondtheblocks.reduxio.com/8-incredibly-useful-tools-to-run-storage-benchmarks>
- [10] Cloudharmony. Accessed: Oct. 13, 2019. [Online]. Available: <https://github.com/cloudharmony/block-storage>
- [11] I/O analyzer - open-source by Intel. Accessed: Nov. 7, 2019. [Online]. Available: <https://labs.vmware.com/flings/i-o-analyzer>
- [12] D. Vanderkam, J. Allaire, J. Owen, D. Gromer, P. Shevtsov, and B. Thieurmel, "Dygraphs: Interface to 'Dygraphs' interactive time series charting library," *R Package Version 0.4*, vol. 5, p. 7, 2015. Accessed: Jan. 6, 2020. [Online]. Available: <http://CRAN.R-project.org/package>

- [13] Cloud computing bare metal storage testing. Accessed: Feb. 19, 2016. [Online]. Available: https://community.oracle.com/community/cloud_computing/bare-metal/blog/2017/05/19/block-volume-performance-analysis
- [14] EzFIO. Accessed: Oct. 13, 2016. [Online]. Available: <http://www.nvmexpress.org/ezfio-powerful-simple-nvme-ssd-benchmark-tool/>
- [15] TKperf. Accessed: Sep. 28, 2019. [Online]. Available: <https://www.thomas-krenn.com/en/wiki/TKperf>
- [16] Automating FIO tests with Python. Accessed: May 28, 2015. [Online]. Available: <https://javigon.com/2015/04/28/automating-fio-tests-with-python/>
- [17] FIO tests. Accessed: Jun. 18, 2018. [Online]. Available: https://github.com/javigon/fio_tests
- [18] R. Gracia-Tinedo, D. Harnik, D. Naor, D. Sotnikov, S. Toledo, and A. Zuck, "SDGen: Mimicking datasets for content generation in storage benchmarks," in *Proc. 13th USENIX Conf. File Storage Technol.*, 2015, vol. 15, pp. 317–330.
- [19] A. Haghdoost, W. He, J. Fredin, and D. H. Du, "On the accuracy and scalability of intensive I/O workload replay," in *Proc. 15th USENIX Conf. File Storage Technol.*, 2017, pp. 315–327.
- [20] V. Tarasov *et al.*, "Extracting flexible, replayable models from large block traces," in *Proc. 10th USENIX Conf. File Storage Technol.*, 2012, vol. 12, Art. no. 22.
- [21] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Generating realistic impressions for file-system benchmarking," *ACM Trans. Storage*, vol. 5, no. 4, 2009, Art. no. 16.
- [22] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proc. 1st ACM Symp. Cloud Comput.*, 2010, pp. 143–154.
- [23] S. Subramanyam, "System and method for testing multiple database management systems," U.S. Patent 5701 471, Dec. 23, 1997.
- [24] Q. Zheng, H. Chen, Y. Wang, J. Zhang, and J. Duan, "COSBench: Cloud object storage benchmark," in *Proc. 4th ACM/SPEC Int. Conf. Perform. Eng.*, 2013, pp. 199–210.
- [25] D. R. Jiang, T. V. Pham, W. B. Powell, D. F. Salas, and W. R. Scott, "A comparison of approximate dynamic programming techniques on benchmark energy storage problems: Does anything work?," in *Proc. IEEE Symp. Adaptive Dynamic Program. Reinforcement Learn.*, 2014, pp. 1–8.
- [26] A. Adir, R. Levy, and T. Salman, "Dynamic test data generation for data intensive applications," in *Proc. Haifa Verification Conf.*, 2011, pp. 219–233.
- [27] A. Traeger, E. Zadok, N. Joukov, and C. P. Wright, "A nine year study of file system and storage benchmarking," *ACM Trans. Storage*, vol. 4, no. 2, 2008, Art. no. 5.
- [28] N. Zhu, J. Chen, T.-C. Chiueh, and D. Ellard, "TBBT: Scalable and accurate trace replay for file server evaluation," *ACM SIGMETRICS Perform. Eval. Rev.*, vol. 33, no. 1, pp. 392–393, 2005.
- [29] R. McDougall and J. Mauro, "Filebench tutorial, sun microsystems," Accessed: Apr. 17, 2019. [Online]. Available: <http://www.nfsv4bat.org/Documents/nasconf/2005/mcdougall.pdf>
- [30] btoreplay - Block trace replay. Accessed: Jul. 22, 2019. [Online]. Available: <https://linux.die.net/man/8/btoreplay>
- [31] E. Anderson, M. Kallahalla, M. Uysal, and R. Swaminathan, "Buttress: A toolkit for flexible and high fidelity I/O benchmarking," in *Proc. 3rd USENIX Conf. File Storage Technol.*, 2004, p. 4.
- [32] Z. Liu, F. Wu, X. Qin, C. Xie, J. Zhou, and J. Wang, "TRACER: A trace replay tool to evaluate energy-efficiency of mass storage systems," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2010, pp. 68–77.
- [33] K. Kanoun, H. Madeira, and A. Jean, "A framework for dependability benchmarking," in *Proc. Workshop Dependability Benchmarking*, 2002.
- [34] nvme-smart-log. Accessed: Aug. 12, 2019. [Online]. Available: <http://manpages.ubuntu.com/manpages/xenial/man1/nvme-smart-log.1.html>

Janki Bhimani is an assistant professor at Florida International University, Miami, FL, USA.

Adnan Maruf is currently working toward the PhD degree in computer science at Florida International University, Miami, FL, USA.

Ningfang Mi is an associate professor at Northeastern University, Boston, MA, USA.

Rajinikanth Pandurangan is a senior staff engineer at Samsung Memory Solutions Lab – R&D, San Jose, CA, USA.

Vijay Balakrishnan is the director of Datacenter Performance and Ecosystem Team at Samsung Memory Solutions Lab, San Jose, CA, USA.

▷ **For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.**

New Performance Modeling Methods for Parallel Data Processing Applications

JANKI BHIMANI, NINGFANG MI, MIRIAM LEESER, and ZHENGYU YANG, Northeastern University, USA

Predicting the performance of an application running on parallel computing platforms is increasingly becoming important because of its influence on development time and resource management. However, predicting the performance with respect to parallel processes is complex for iterative and multi-stage applications. This research proposes a performance approximation approach *FiM* to predict the calculation time with *FiM-Cal* and communication time with *FiM-Com* of an application running on a distributed framework. *FiM-Cal* consists of two key components that are coupled with each other: (1) a Stochastic Markov Model to capture non-deterministic runtime that often depends on parallel resources, e.g., number of processes, and (2) a machine-learning model that extrapolates the parameters for calibrating our Markov model when we have changes in application parameters such as dataset. Along with the parallel calculation time, parallel computing platforms consume some data transfer time to communicate among different nodes. *FiM-Com* consists of a simulation queuing model to quickly estimate communication time. Our new modeling approach considers different design choices along multiple dimensions, namely (i) process-level parallelism, (ii) distribution of cores on multi-processor platform, (iii) application related parameters, and (iv) characteristics of datasets. The major contribution of our prediction approach is that *FiM* can provide an accurate prediction of parallel processing time for the datasets that have a much larger size than that of the training datasets. We evaluate our approach with NAS Parallel Benchmarks and real iterative data processing applications. We compare the predicted results (e.g., end-to-end execution time) with actual experimental measurements on a real distributed platform. We also compare our work with an existing prediction technique based on machine learning. We rank the number of processes according to the actual and predicted results from *FiM* and calculate the correlation between the actual and predicted rankings. Our results show that *FiM* obtains a high correlation in the range of 0.80 to 0.99, which indicates considerable accuracy of our technique. Such prediction provides data analysts a useful insight of optimal configuration of parallel resources (e.g., number of processes and number of cores) and also helps system designers to investigate the impact of changes in application parameters on system performance.

CCS Concepts: • **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability;

Additional Key Words and Phrases: Performance modeling, queuing theory, Markov model, distributed systems, execution time, parallel calculation, communication network, prediction

This work was partially supported by National Science Foundation (NSF) award CNS-1452751, CNS-1717213, and AFOSR grant FA9550-14-1-0160.

Authors' address: J. Bhimani, N. Mi, M. Leaser, and Z. Yang, Northeastern University, 360 Huntington Ave, Boston, MA, 02115, USA; emails: {bhimani, ningfang}@ece.neu.edu, {mel, yangzy1988}@coe.neu.edu.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.

1049-3301/2019/06-ART15 \$15.00

<https://doi.org/10.1145/3309684>

ACM Reference format:

Janki Bhimani, Ningfang Mi, Miriam Leeser, and Zhengyu Yang. 2019. New Performance Modeling Methods for Parallel Data Processing Applications. *ACM Trans. Model. Comput. Simul.* 29, 3, Article 15 (June 2019), 24 pages.

<https://doi.org/10.1145/3309684>

1 INTRODUCTION

High-Performance Computing (HPC) systems are ubiquitous in processing data for myriad applications involving huge datasets. How to achieve the best performance with an optimal configuration of parallel resources (e.g., number of processes and number of cores) is a challenging research problem. Currently, researchers run their application codes on a representative dataset, fix application parameters, and try different configurations of parallel resources to determine the optimal one. However, if we want to find optimal application parameters, then the investigation needs to consider all possible combinations of application parameters and parallel resources. Such an investigation becomes very expensive, requiring a significant amount of time and hardware resources. Besides, on parallel computing platforms, using more parallel resources does not always guarantee performance improvement. Hence, it is beneficial if we can approximate the optimal performance in terms of parallel resources and application parameters. The prediction of expected performance before the porting of an actual implementation on a hardware platform can save significant time and hardware resources spent in experimentally finding the optimal performance. The emergence of huge datasets as workloads and parallel computing has emphasized the importance of predictive analysis, and performance bottleneck identification. Designing efficient prediction tools thus becomes critically important to system designers and application programmers [24].

As motivation, we show an example with an iterative k -means clustering application running on a framework using Message Passing Protocol (MPI [19, 35]) in Figure 1. We observe that the calculation time decreases when we have more parallel MPI processes; however, the time to communicate data increases. Such an observation implies that speeding up parallel calculation time may not guarantee overall application speed-up. Also, the decrease in calculation time levels off after 70 parallel processes. Increasing the number of parallel processes further consumes more system resources, but may not improve overall application runtime. Thus, the capability of predicting such an optimal point (e.g., 70 in Figure 1) is vital to system designers for making good design choices.

In this article, we develop a new performance modeling approach, named FiM, to estimate both computation and communication times of iterative, multi-stage data processing applications using MPI. Each node is a CPU that has multiple cores, and each core can support multiple MPI processes. One of the nodes accesses an application dataset and determines the distribution of processing among the processes of other nodes. All these nodes then perform parallel computations using multiple processes. Such a parallel phase is known as one stage in our model. At the end of each stage, all processes synchronize to decide on the termination or launch next stage. In this research, we concentrate on predicting parallel processing time of such an iterative and multi-staged application running with global synchronizations. One of the key innovations in our work is that FiM relies only on small datasets for training but can predict the execution times for larger datasets. More specifically, this article aims to answer the following questions through our prediction models.

- Can we quickly estimate the parallel calculation and communication times of an application to identify the optimal number of processes?
- Can we use small datasets as training to predict performance of applications operating in parallel on large datasets?

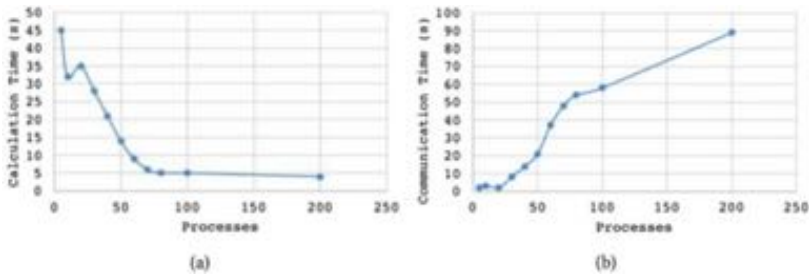


Fig. 1. Latency variation across different number of parallel processes for (a) calculation time and (b) communication time.

- How does the number of processes impact calculation and communication time?
- Is the application communication or compute bound?

To answer these questions, we introduce FiM, which consists of two main components: (1) FiM-Cal and (2) FiM-Com. The goal of FiM-Cal is to predict the calculation time by using a stochastic Markov model and a machine-learning model. The stochastic Markov model is built using the probabilistic technique to estimate the impact of an increase in the number of parallel processes. We first develop the base case of the parallel paradigm and then derive a generic model that applies to any number of parallel processes as well as any number of dependent stages (e.g., iterations) of an application. The base case of the Markov model is calibrated using the minimum number of system parameters. The machine-learning model is then designed to extrapolate the calibrated parameters for the Stochastic Markov model to adapt to changes in application parameters such as datasets.

The goal of FiM-Com is to predict the communication time using a set of simulation queuing models. Here our motive is to get a quick estimate using a simplified prediction model. Such an estimate of communication time along with calculation time can provide instant insight to users. Thus, our FiM approach can use the minimum possible calibration parameters to quickly predict the expected computation and communication time as well as the optimal number of processes for platform configuration. While comparing actual and predicted time, the worst prediction error of the overall application runtime by FiM is observed to be less than 20%. The source code of our calculation and communication time prediction model is available at GitHub (<https://github.com/bhimanijanki/FiM>). A preliminary version of the article, titled “FiM: Performance Prediction for Parallel Computation in Iterative Data Processing Applications,” was published in *Proceedings of the IEEE International Conference on Cloud Computing (CLOUD’17)* [8].

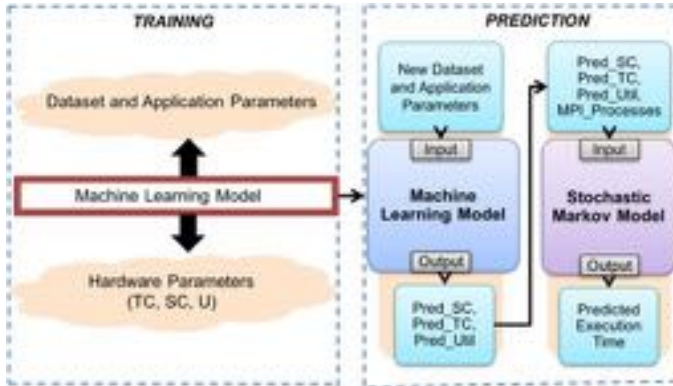
The remainder of this article is organized as follows. We present the two FiM components in Section 2 and Section 3.2, respectively. We evaluate our models on a distributed memory platform, see Section 4. In Section 5, we discuss some related work. Section 6 presents our conclusions and future works.

2 FIM-CAL: CALCULATION PREDICTION

In this section, we present *FiM-Cal*, an analytical approach to predicting the calculation time of an application running on a distributed multi-process platform. FiM-Cal consists of two key components: a stochastic Markov model and a machine-learning model. We first use the stochastic Markov model to represent the computational processing of an application in a parallel MPI framework. Then we design a machine-learning algorithm to estimate the parameters related to the system for calibrating our stochastic Markov model. This parameter extrapolation enables our model

Table 1. Notations Used in *FiM-Cal*

Notation	Description
S_j^i	Markov chain state with i active and j passive processes
P_{s_i}	Stage completion probability of i^{th} stage
P_{ij}	State transition probability of moving from i active to j active processes
P_{act}	Probability P_{11} of 1 process model
P_{p2a}	Probability P_{01} of 1 process model
P_{a2p}	Probability P_{10} of 1 process model
P_{pass}	Probability P_{00} of 1 process model
F	Frequency (GHz)
TC	Total cycles
SC	Total stall cycles
U	Utilization per process
T_i	Total time taken by stage i
α	Sensitivity constant
β	Regression constant
y_i	Dependent variables
\vec{X}_i	Vector of independent variables

Fig. 2. Prediction procedure of *FiM-Cal*.

to predict an application's calculation time when we have a different number of parallel processes or variable application parameters (such as dataset size) without any system state instrumentation. Table 1 lists the notations used in this article for *FiM-Cal*. Figure 2 shows the overall workflow of our proposed *FiM-Cal*. We will introduce the details of each component in Figure 2 in the remainder of this section.

2.1 Stochastic Markov Model

Our stochastic Markov model is designed to model computational processing for an application running on a system with parallel multi-core CPUs deployed using MPI. Such a stochastic model allows us to capture a non-deterministic runtime that often depends on parallel resources, e.g., number of processes. If there exists a global synchronization call in an application, then all processes wait until the barrier. The processing of an application is partitioned into multiple stages

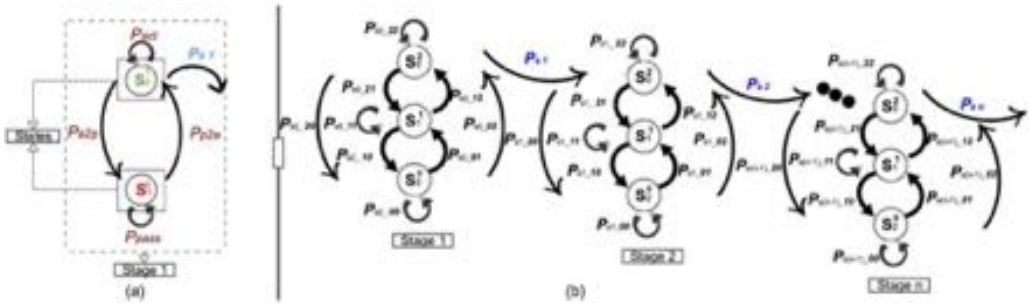


Fig. 3. Modeling (a) the base case: single process with single stage and (b) generic case: two processes with multiple stages.

with respect to this global synchronization, such that each stage corresponds to a parallel phase until all processes have completed their tasks and are in an active state to proceed to the next stage. In this section, we first introduce our base case, which models a single stage for a single-process system, and then show its extension to the generic case with multiple processes and multiple stages.

2.1.1 Base Case. The base case model is built to represent a single process for a single stage application, see Figure 3(a). In our model, each process is considered to be either in *active* or *passive* state. As shown in Figure 3(a), when only a single process runs in the computing platform, we have two states for a stage such that state S_1^1 represents that one process is active while S_0^1 represents that one process is passive. We also introduce transition probabilities (e.g., P_{act} , P_{a2p} , P_{pass} , P_{p2a}) of switching between two states or staying in the same state, as well as the stage completion probability (P_{s1}) of transferring from one stage to another. In the active state, the process performs constructive work and typically changes from the active state to the passive state when it is blocked by an event that would create a latency stall. Such a latency stall might be caused due to a cache miss that takes many CPU cycles. In this work, we do not model memory latencies, contentions, inter-dependencies and deadlocks individually for each process but rather treat the combined effect as a process remaining passive.

The probabilities in the base case model are parameterized by instrumenting the system details, which is further used to derive the probabilities of the generic case. To parameterize the probabilities of the base case, we use the `perf` tool [16] to instrument the required data, including the hardware clock rate (F), system CPU utilization factor per process (U), total number of cycles required for execution (TC), and stall cycles (SC). In particular, we run an application with a single stage on a single process and use the `perf stat` command to collect and report the required data as listed above.

In the base case (i.e., single-process and single-stage), the probability to remain in the active state (P_{act}) is primarily determined by the proportion of time that the process is performing useful work. Therefore, we use Equation (1) to get P_{act} ,

$$P_{act} = U, \tag{1}$$

where U is utilization per process. As shown in Figure 3(a), when the process is in an active state (i.e., S_0^1), there are three possible events for its next transition: (1) Remain in S_0^1 with probability P_{act} , (2) transition to with probability P_{a2p} , and (3) complete the stage with probability P_{s1} . Now, if there are TC total cycles to be processed for the given dataset, then processing is completed only after completing the last cycle. This gives the probability of completion as $1/TC$. Probability P_{a2p}

for the process to transit to the passive state can then be calculated as shown in Equation (2),

$$P_{a2p} = 1 - (P_{act}) - (1/TC). \quad (2)$$

The probability of the process remaining passive (P_{pass}) is primarily determined by the ratio of stall cycles (SC) to total cycles (TC) as shown in Equation (3),

$$P_{pass} = SC/TC. \quad (3)$$

We can determine the probability of switching from passive to active (P_{p2a}) by applying the control flow equation to the passive state (S_1^0) as shown in Equation (4),

$$P_{p2a} = 1 - (P_{pass}). \quad (4)$$

We finally get the stage completion probability (P_{s1}) by applying the control flow equation to the active state (S_0^1) as shown in Equation (5),

$$P_{s1} = 1 - (P_{act}) - (P_{a2p}). \quad (5)$$

Note that, for the base case with one stage and only one possible active phase, P_{s1} is the same as $1/TC$, because all active cycles can be spent only in one active state (S_0^1). Later, in generic cases, we discuss the calculation of P_{s1} , which is then not the same as $1/TC$.

2.1.2 Generic Cases. Now we consider generic cases where we can have multiple processes operating on an application with multiple stages. This generic behavior can be modeled as an extension of the base case. The processing of an application may have multiple inter-dependent parallel stages. For example, an iterative application with 500 iterations can be divided into 500 parallel stages such that each stage represents an iteration and is entered only after the completion of all prior stages. Thus, the first stage corresponds to the parallel calculation phase by all processes in the first iteration and is followed by the remaining stages in the same order. In general a simple non-iterative and single stage application can be treated as having 1 iteration and 1 stage while predicting its runtime using our proposed technique.

Figure 3(b) shows an underlying Markov model for an application with two processes using n stages, each with two parallel processes. The entire workflow of an iterative, multi-stage, multi-process application can be mapped with a chain of n parallel stages, and $P_{s1}, P_{s2}, \dots, P_{sn}$, are the completion probabilities for all n stages, see Figure 3(b). Note that these stage completion probabilities are non-uniform and dependent on all the completion probabilities of prior stages as well as intra-state transition probabilities of that stage. Also for every stage, all its state transition probabilities (P_{ij}) depend on the completion probability of the prior stage. Thus, the value of P_{ij} in a stage is different from that of P_{ij} in another stage even for the same i and j . Furthermore, a single stage can only complete when all of its processes are active, i.e., not being blocked by any events. A calculation phase of an application is completed when tasks assigned to all processes are completed in the last stage. Thus, the completion probability of an application is P_{sn} .

To model an iterative, multi-stage paradigm with multiple processes, we use multiple states within each stage to represent activities (active or passive) of all processes. Consider t processes with i active processes and j passive processes, where $0 \leq i \leq t$, $0 \leq j \leq t$ and $t = i + j$. Each stage consists of a total of $M = t + 1$ states. Thus, the transition probabilities of jumping from any one of these M states to other states or itself can be divided into three types: (1) probability to remain in the same state (e.g., P_{22}, P_{11} and P_{00} in Figure 3(b)), (2) probability to increase active processes (e.g., P_{01}, P_{12}, P_{02} in Figure 3(b)), and (3) probability to increase passive processes (P_{10}, P_{21}, P_{20} in Figure 3(b)). Given M states, we have M probabilities to remain in the same state, $\sum_{i=1}^{M-1} i$ probabilities to increase active processes, and $\sum_{i=1}^{M-1} i$ probabilities to increase passive processes.

Thus, the total number of probabilities to be calculated for a single stage with M states is equal to $M + 2 \sum_{i=1}^{M-1} i = M^2$.

2.1.3 Solving the Generic Model. We solve such a generic Markov model and derive its probabilities by relating them to the preliminary transition probabilities of the base case. That is, once we have transition probabilities for $M = 2$ (base case), we can calculate all probabilities for a generic case with $M > 2$. In Reference [27], a mathematical relation between the transition probabilities of a Markov model with two states and a Markov model with more than two states has been derived. We use their method to relate transition probabilities of the cases with $M = 2$ and $M > 2$. Initially, base case probabilities are calculated.

Specifically, let us consider to derive generic model probabilities for an application with $M = 3$ from the base case probabilities of $M = 2$. For $M = 3$, the application must have two parallel processes (see Figure 3(b)). At any given time during the execution, there are three states ($M = 3$): (1) Both of these processes can be active (state S_0^2), (2) one process can be active and the other passive (state S_1^1), and (3) both can be passive (state S_2^0). The probability of two active processes to continue as active is $P_{22} = P_{act} * P_{act}$. Similarly, the probability of two passive processes to continue as passive is $P_{00} = P_{pass} * P_{pass}$. If one process that was active in previous stage remains active and the other process that was passive remains passive, or if the active process becomes passive and the other process that was passive becomes active, then in both of the above cases the model stays in state S_1^1 . Thus, the probability of such state transition is $P_{11} = (P_{act} * P_{pass}) + (P_{a2p} * P_{p2a})$. P_{01} is state transition probability from S_2^0 (i.e., both the passive processes) to S_1^1 (i.e., one active process and the other passive process).

Same as above, the state transition probabilities for any M can be derived. These derived equations can be mathematically reduced to a more generic form. Equations (6) to (8) give the state transition probabilities for $M > 2$, where i corresponds to the number of active processes in the previous state and j corresponds to the number of active processes in the targeted state. For example, in Figure 3(b), P_{21} indicates the state transition probability of moving from a state with 2 active processes (S_0^2) to a state with 1 active process (S_1^1). Substituting appropriate i and j in Equations (6) to (8) for $M = 3$, all the probabilities explained and derived above for 2 processes such as P_{00} , P_{11} , P_{22} , P_{01} , P_{02} , P_{12} , P_{10} , P_{20} , and P_{21} can be obtained. These equations represent the stochastic process of a Markov chain and can be calculated by mathematical induction after solving the Markov chain with a finite number of states. Particularly, as shown in Equations (6) to (8), we use the probabilities (P_{act} , P_{a2p} , P_{pass} , and P_{p2a}) that are obtained by the base case (Section 2.1.1) to calculate the state transition probabilities in generic cases.

If $i == j$, then

$$P_{ii}(t) = \sum_{k=0}^{\min\{i,t-i\}} \binom{i}{k} \cdot \binom{t-i}{t-i-k} \cdot (P_{act}^{i-k}) \cdot (P_{a2p}^k) \cdot (P_{pass}^{t-i-k}) \cdot (P_{p2a}^k). \quad (6)$$

If $i < j$, then

$$P_{ij}(t) = \sum_{k=0}^{\min\{i,t-j\}} \binom{i}{k} \cdot \binom{t-i}{t-j-k} \cdot (P_{act}^{i-k}) \cdot (P_{a2p}^k) \cdot (P_{pass}^{t-j-k}) \cdot (P_{p2a}^{j-i+k}). \quad (7)$$

If $i > j$, then

$$P_{ij}(t) = \sum_{k=0}^{\min\{j,t-i\}} \binom{i}{i-j+k} \cdot \binom{t-i}{t-i-k} \cdot (P_{act}^{j-k}) \cdot (P_{a2p}^{i-j+k}) \cdot (P_{pass}^{t-i-k}) \cdot (P_{p2a}^k). \quad (8)$$

Additionally, after capturing the state transition probabilities of the first stage, we calculate the stage completion probability P_{S_1} using Equation (9) and then use P_{S_1} as an incoming probability for calculating the state transition probabilities of stage 2, and so on. This chaining process captures an iterative and multi-stage application running with multiple processes. Finally, P_{S_n} for the n th stage is calculated by Equation (9),

$$\begin{aligned} P_{S_n} &= P(X_s = n \mid X_{s-1} = n - 1) \\ &= 1 - \sum_{j=0}^{j=i} (P_{ij} \mid X_{s-1} = n - 1) \end{aligned} \quad (9)$$

for $i = \text{Max}(\#\text{Processes})$.

We further use Equation (10) to calculate the time (T_n) spent in performing parallel calculation for n stages, given the completion probability (P_{S_n}) and CPU frequency (F),

$$T_n = \frac{1}{(P_{S_n}) \cdot (F)}. \quad (10)$$

Consequently, our stochastic Markov model can predict the computation time required to process any particular dataset using different levels of parallelism such as different number of processes in MPI. Next, we present our machine-learning technique that assists to extrapolate the data (such as, F , U , TC , and SC) required for calibrating the base case model.

2.2 Machine-learning Model

Our stochastic Markov model allows us to predict the calculation time of an application when we have a different number of processes in the system. However, the required hardware parameters (i.e., TC , SC , U) need to be instrumented for every new dataset and a new setting of application parameters. This limits the scope of the model to predict for a particular set of datasets and fixed application parameters. Most analytical models suffer from this lack of flexibility. Therefore, we develop our machine-learning model to avoid additional instrumentation for a new dataset or a new set of application parameters. For a new application, we need first to train to derive a new model. However, for new sets of application parameters and new datasets of the same application, the derived model can be used to predict the runtime. To reduce the complexity of the machine-learning model, we also assume that the application calculation time is dependent on the fewest possible hardware parameters. Our evaluation results, shown in Section 4, demonstrate the feasibility of this assumption by showing the fairly accurately predicted results obtained by our approach that is good to give a quick approximation. Here we introduce a two-stage machine-learning model that emulates hardware behaviors without performing actual instrumentation for required hardware-related data. Such a hybrid emulation of hardware is the key to allowing the approach to be able to predict parameters for datasets with sizes much larger than those of the training datasets.

2.2.1 Regression Mapping. The focus of regression is to find the relationship between a dependent variable (such as the hardware parameters that we want to emulate) and one or more independent variables (such as application parameters and datasets). This analysis estimates the conditional expectation of a dependent variable given values of all related independent variables. We find that the generalized linear regression model performs the best when compared to others (quadratic, Poisson model, and gradient decent) for modeling all desired hardware parameters (U , TC , and SC). We show the validation of a linear regression model in Section 4.3.

The linear regression equation for learning variable y_i is shown in Equation (11), where \vec{X}_i is a vector of p -independent variables related to application parameters and datasets, $\vec{\beta}_i$ consists of a vector of $p + 1$ constants, and n is total number of scalar-dependent variables. Suppose for the

k -means application, elements of \vec{X}_i would consist of the number of desired clusters (K), the number of iterations (I), and size (N). For our model, we have three scalar-dependent variables, i.e., U , TC , and SC , which can be predicted after building this linear regression model. Thus, we have three equations for y_1 , y_2 , and y_3 with $n = 3$,

$$\begin{aligned} y_i &= \beta_{i0} + \beta_{i1}x_{i1} + \cdots + \beta_{ip}x_{ip} \\ &= \vec{\beta}_i(1 + \vec{X}_i^T), \text{ for } i = 1, 2, \dots, n. \end{aligned} \quad (11)$$

This linear regression model is used to find values for constants $\vec{\beta}_i$ using the training data for which both dependent variables and independent variables are known. We obtain the regression curve and regression constants $\vec{\beta}_i$ by building our machine-learning model in MATLAB.

ALGORITHM 1: Calibration of α

```

1 Input:  $\epsilon, \tau, y_i, \vec{X}_i$ , Output:  $\alpha$ 
2 Initialize:  $\vec{\beta}_i, TC, SC, U$  using regression mapping,  $\alpha = 0$  and iter = 0
3 if  $0 \leq U \leq 1, TC > 0, SC > 0, SC < TC$  then
4   Predict comp. time using stochastic Markov model
5   Calculate RMS error (Actual, Predicted)
6   if iter == 0 then
7     Calculate error (Actual - Predicted)
8     Decide OP = + or -, depending on positive or negative error
9   if RMS error <  $\tau$  then
10    return
11  else
12     $\alpha = \alpha \{OP\} \epsilon$ 
13    iter ++
14    Calculate  $TC, SC$ , and  $U$  using Equation (12)
15    goto line 3
16 else
17   Neglect bad values
18   goto line 4

```

2.2.2 Iterative Improvement Model. We found non-negligible errors between the actual and predicted calculation times when we pair the linear regression model described above with our stochastic Markov model, to predict calculation time of large datasets based on small training datasets. To handle this issue, we develop an iterative improvement model that uses a sensitivity parameter α to tune the constant factors $\vec{\beta}_i$ with respect to the predicted results (i.e., calculation time) of our stochastic Markov model as shown in Equation (12). Note that in this equation the constants in vector $\vec{\beta}_i$ are obtained from the regression between hardware parameters and application parameters, but constant α is obtained by using both Markov model and regression model as described in Algorithm 1,

$$\begin{aligned} y_i &= \alpha(\beta_{i0} + \beta_{i1}x_{i1} + \cdots + \beta_{ip}x_{ip}) \\ &\text{for } i = 1, 2, \dots, n. \end{aligned} \quad (12)$$

Initially, we use hardware parameters (U , TC , and SC) and application parameters of training data with regression mapping (Equation (11)) to obtain constants of vector $\vec{\beta}_i$. The hardware parameters are passed to our stochastic Markov model to predict calculation time. In the first iteration, the error between actual and predicted time is used to decide the adjust direction of α , see lines 6 to 8. That is, if the actual value is greater than the predicted one, then the algorithm increases α and vice versa. In the following iterations, Algorithm 1 increases or decreases α by a small value ϵ (e.g., $\epsilon = 1e-5$), and the hardware parameters (U , TC , and SC) are predicted using constants of vector $\vec{\beta}_i$ and α with Equation (12) (see line 14). Then the computation time is predicted using U , TC , and SC as the inputs to our stochastic Markov model (see line 4). The adjustment process continues until the root mean square (RMS) error becomes smaller than a predefined threshold (e.g., $\tau = 0.01$), see line 9. The value of τ needs to be set such that we avoid underfitting as well as overfitting the training data. From our experiments, we observe that $\tau = 0.01$ is a good value on an average across different choices of applications and training datasets. Thus, the algorithm adjusts the value of α until the predicted calculation time becomes close to the actual time. We tune α while training the model and then use the exact tuned α value throughout the prediction stage. Note that our machine-learning model is used to calculate the constants of vector $\vec{\beta}_i$ and α in the training phase, which are after that used for extrapolation of hardware parameters.

In summary, Figure 2 shows the overall procedure of our prediction model *FIM-Cal*, which includes the training and prediction phases. In the prediction phase, our machine-learning model extrapolates the dependent variables (such as hardware parameters: SC , TC , U) for new datasets and new sets of application parameters. These predicted hardware details can then be used as an input to our stochastic Markov model to predict the calculation time.

3 FIM-COM: COMMUNICATION PREDICTION

Apart from computation time [7, 8], applications spend some time in the communication of data to various processes. With the increase in the number of processes, this communication time keeps increasing [37]. Therefore, it is essential to roughly estimate communication time, i.e., the runtime of *data transfer*. For applications running on a distributed multi-process system, the data transfer time for processes lying on the same node is different from that between processes lying on different nodes. The framework considered in this work consider all inter-node communication.

3.1 Communication Patterns

It is non-trivial to predict communication time, due to various communication patterns and the non-deterministic latency of the communication network. There exist many, more complex models to predict accurate network communication time [10, 20, 23, 30], but here our motive is to get a quick estimate using a simplified prediction model. In this work, we investigate three types of collective communication patterns including downlink (scatter and broadcast) and uplink (gather). We build different queuing models to capture those patterns when running a data processing application such as k -means clustering on a multi-process system with MPI. Because the data transfer time along with computation time is significant, our model aims to provide very quick rough estimates of communication time to make fast decisions.

Specifically, we consider communications from the one to many nodes as the *downlink*. Such a downlink communication can have either the scatter pattern or the broadcast pattern. Under the scatter pattern, the data are distributed among the multiple nodes by the one node such that each node gets a unique part of the data. Under the broadcast pattern, the data are broadcast to all nodes, and each node receives the same copy of the data. The scatter communication is usually undertaken by communicating data to one after another process in MPI. We also consider communications

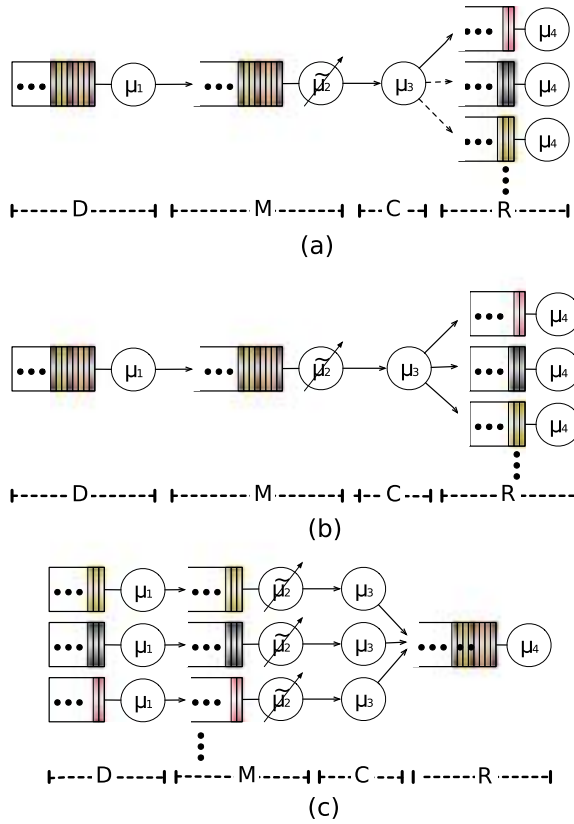


Fig. 4. Queuing models for the (a) scatter, (b) broadcast, (c) gather communication patterns.

from multiple nodes to the one node as the uplink gather pattern. When gathering, each node sends their own data to a shared buffer, and the managing node reads those data one by one from the buffer.

3.2 Communication Models

We develop a set of queuing models to capture these three collective communication patterns. Such queuing models can give a high-level abstraction of real communication systems in terms of packet arrival rates, delay/waiting time, and packet transfer rate, which helps to estimate the overall data communication time. Note that we try to keep these queuing models simple to enable fast estimation.

Figure 4 presents our queuing models to represent the scatter, broadcast, and gather patterns, respectively. Typically, communication time depends on various network properties such as initialization cost, maximum network bandwidth, network load, and the amount of metadata (e.g., headers, acknowledgments, and flags). Therefore, in each of these models, we use four components, Data transfer (D), Metadata (M), Cost of network initialization (C), and Receive (R) in Figure 4, in series to capture different properties of network communication. Each job in the queue of D, M, C, and R represents a data packet e.g., a pixel for k -means clustering.

First, D is used to capture the actual data transfer rate through the communication network, where mean service rate μ_1 indicates the available network bandwidth. We assume that all data

packets are available in memory. Thus, all jobs are assumed to wait in the queue of part D before data transfer begins.

The second part M, in the models of Figure 4 captures the effect of metadata on overall communication time. Such metadata can include headers, acknowledgments, addresses, offsets, padding, and flags that need to be transferred over the network along with the original data packets as part of the network protocol. The time consumed by such metadata transfers depends on the number of actual data packets and instantaneous network load. To capture this, we design component M with a variable mean service rate $\tilde{\mu}_2$, which depends on the instantaneous queue length of part M. The arrivals to this queue are the departures from part D.

Part C in the model represents network initialization. Usually, the cost for network initialization is a network latency that is added to overall communication time. We use a delay server with mean service rate μ_3 to emulate such network latency. The last part of our models (R in Figure 4) is used to estimate the data fetching time on the receiver side. We use part R to capture the receiving, while the first three parts (D, M, and C) capture the sending. We assume that the receiving process is homogeneous across different nodes. Therefore, multiple servers and queues are used to represent multiple receivers, and the mean service rate of each server is μ_4 .

We present the corresponding models for scatter and broadcast in Figure 4(a) and (b), to predict communication times for the downlink, i.e., from the one node to multiple nodes. Under the scatter pattern, the one node sends different pieces of data to other nodes one by one. We use the round robin policy to distribute jobs into queues in part R, and thus there is only one solid arrow in R to indicate data transfer to one process at a time, see Figure 4(a). In Figure 4(b), there are multiple solid arrows to R, which represents that we duplicate jobs (data points) and distribute them to all queues of R in parallel to capture the broadcast pattern. The third model, shown in Figure 4(c), represents the gather pattern of the uplink communication, where multiple nodes send their data to one node. We have multiple servers running in parallel in parts D, M and C. The First-In-First-Out (FIFO) discipline is used in the last queue to emulate data fetching by the managing node.

3.3 Communication Model Calibration

The main tasks in communication model calibration are (1) to determine an appropriate queuing model and (2) to derive service processes for all servers in the model based on its collective communication pattern. Our models only need to be calibrated when we have a new application or run on a new hardware platform. We then leverage the knowledge of actual transfer processes on a new platform to calibrate the service rates of all servers in the models. The calibrated models can be used directly to predict communication times for new datasets, new application parameters (e.g., number of iterations in k -means) and different number of MPI processes.

In particular, we calibrate μ_1 using the maximum network bandwidth obtained from the provided network configuration of a cluster (e.g., 10Gb/s backplane). We observe from conducting experiments on multiple applications for both 10Gb/s and faster 56Gb/s TCP/IP backplane that the actual data transfer rates are often within 90% to 100% of the maximum network bandwidth, with a uniform distribution. Therefore, we calculate the lower bound (ζ_{min}) and the upper bound (ζ_{max}) of transfer rate per unit job (i.e., each data point) and derive the service rate μ_1 in part D using Equation (13),

$$\mu_1 = Uniform(\zeta_{min}, \zeta_{max}). \quad (13)$$

To get the transfer time of the metadata, we first measure the actual total communication time for sending a unit job (a single data point) and then deduct the measured runtime of the other three parts, i.e., network bandwidth, network initialization, and receiver's data fetch time. We find that the derived transfer rates of metadata are logarithmic to the instantaneous network load (refer

Table 2. Platform Configurations Labeled as C1 to C5 (2-E52670—Two Multi-core, Hyper-threaded Intel Xeon E5 2670 CPU's @ 2.60GHz and 256GB of RAM) (2-E52650—Two Multi-core, Hyper-threaded Intel Xeon E5 2650 CPU's @ 2.00GHz and 128GB of RAM)

	C1	C2	C3	C4	C5
CPU	2-E52670	2-E52670 2-E52650	2-E52670 4-E52650	2-E52670 6-E52650	2-E52670 8-E52650
Cores	32	64	96	128	160
Network	10Gb/s Ethernet backplane TCP/IP				
Shared FS	NFS				
OS	Linux				

Section 4.3). Therefore, we use Equation (14) to generate $\tilde{\mu}_2$ for metadata, where χ represents the number of jobs currently waiting in the queue of M. The tilde on μ_2 represents the variable mean service rate of component M, which depends on the instantaneous queue length of part M (χ) to symbolize variance as explained above,

$$\tilde{\mu}_2 = \log(\chi). \quad (14)$$

In part C, μ_3 captures the network initialization latency using a delay server. Specifically, the sending node first activates itself (`comselfcreate`) and then activates the receiving nodes (`comworldcreate`). The MPE tool [19] is used to collect the `mpilog` log files. We observe that the initialization latency follows an exponential distribution (refer Section 4.3). Thus, we use Equation (15) to get the service rate μ_3 , where η is the mean of the service rate distribution,

$$\mu_3 = \exp(\eta). \quad (15)$$

Finally, we observe that the data receiving rate is exponentially distributed (refer Section 4.3). Additionally, we consider a multiplicative factor ϕ to model the number of receivers that simultaneously fetch data. For example, ϕ is equal to one under the scatter pattern, since only one node can receive data at any time. For the broadcast pattern, ϕ is equal to the number of nodes. Thus, Equation (16) is used to draw μ_4 for all the servers in part R, where ψ is the mean service rate distribution,

$$\mu_4 = \phi * \exp(\psi). \quad (16)$$

4 EVALUATION

We evaluate our FiM prediction approach (a combination of FiM-Cal and FiM-Com) by comparing the predicted results (e.g., end-to-end execution time) with actual experimental measurements on a real distributed platform. We also compare our prediction approach with an existing work, named RBASP [5], which is a regression-based approach to extrapolate execution time. In our evaluation, we consider the end-to-end execution time of data processing applications as the sum of the runtime spent in communicating the required data to parallel processing units, performing the calculations in parallel, and transferring the results back to the managing node. We assume that no overlap exists between the calculation and communication in the application implementation. We use the Discovery Cluster at Northeastern University [1] to build our experimental platform. Table 2 describes five parallel platform configurations we used in our evaluation, where each CPU belongs to different nodes.

We evaluate our approach with six NAS Parallel Benchmarks (NPB—version *NPB3.3.1-MPI*) [2], with the large size dataset of *Problem Class C*. Table 3 lists the six benchmarks we used in our

Table 3. NPB Benchmarks

BT	Block Tri-diagonal solver	Compute Bound
EP	Embarrassingly Parallel	Compute Bound
SP	Scalar Penta-diagonal solver	I/O Bound
LU	Lower-Upper Gauss-Seidel solver	I/O Bound
IS	Integer Sort	Memory Bound
CG	Conjugate Gradient	Memory Bound

evaluation. For each benchmark, we train our model using three small size datasets of *Problem Class S*. Note that we use the trained model to predict the performance for datasets of *Problem Class C*, which are much larger than the datasets of *Problem Class S*. We reprogram the NPB benchmarks to implement iterative, multi-stage paradigm versions in MPI using C. The time spent for computation in all iterations (or multiple phases) is the total computation time. For each iteration, we measure the time from the start of parallel processes to the completion of all the processes. We also evaluate FiM with two iterative data processing applications: k -means [6] and Pagerank. For each of these applications, we run experiments on 15 different datasets and choose one dataset as a representative to show the results, i.e., N_L (the large dataset with 13 million data points). For both applications, we train our model using three small datasets, i.e., N_S (the small datasets each with 3 thousand data points).

k-means Clustering (KM): Our k -means clustering implementation [6] takes color images as input datasets. We cluster pixels in an image based on five features, including three *RGB* channels and the position (x, y) of each pixel. We choose random data points to initialize each cluster centroid and then use the Euclidean distance to calculate the nearest cluster for each data point. The parameters of k -means include a number of desired clusters (K), number of iterations (I), and size of input dataset (N).

Pagerank (PR): The Pagerank application takes a network of directed vertexes and edges as an input dataset. The output of the Pagerank application is a probability distribution representing the weights of each vertex (page). We choose a damping factor of 0.85 and initialize all vertexes (pages) with the same probability weights. The parameters of this application include a number of vertexes (V), number of iterations (I), size of input dataset (N), and network nature (dense or sparse).

4.1 Performance Evaluation

In our evaluation, we consider a regression-based approach named RBASP [5] to compare with our FiM approach. RBASP is well known for its simplicity and accuracy in extrapolating execution time of multi-process applications. In our FiM approach, we combine our stochastic Markov modeling with machine-learning regression to predict calculation time and use our queueing models to predict communication time. Our prediction model with this combination of popular techniques helps to predict accurately in most cases and gives a quick estimation. We choose RBASP to compare our prediction accuracy, because, similarly to FiM, it also gives a quick prediction and can estimate runtime of datasets larger than the training datasets. RBASP is a pure regression-based approach; unlike FiM, it directly estimates total time (calculation + communication) using regression. RBASP model predicts the execution time (y) of a given parallel application on p processes by using several instrumented runs of an application on q processes, where $q \in \{1, \dots, p_0\}$ and $p_0 < p$. By varying the values of independent variables (x_1, x_2, \dots, x_n) , this model aims to calculate coefficients $(\beta_0, \dots, \beta_n)$ by the linear regression fit for $\log_2(y)$ (Equation (17)), where $g(q)$ can

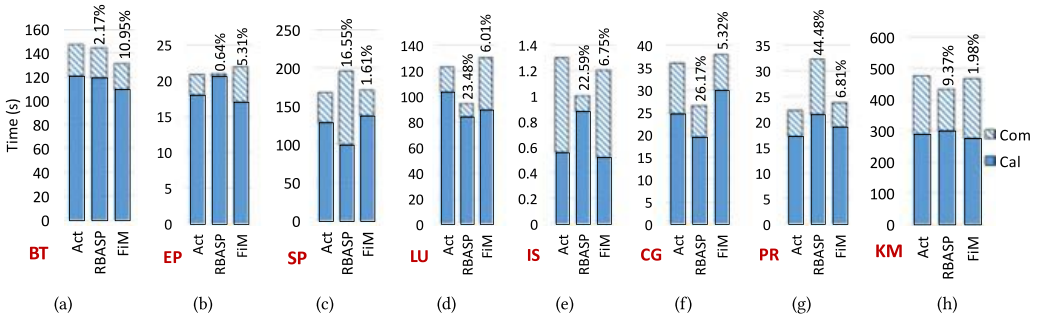


Fig. 5. Actual and predicted execution time using FiM and RBASP with the relative prediction error listed on top of each bar.

Table 4. Summary of Results for All Applications (Rel. Er.-Relative Error) (Act.-Actual) (App.-Application) (Opt.-Optimal)

App.	Best Rel. Er. %		Average Rel. Er. %		Worst Rel. Er. %		Opt. # MPI Process		
	RBASP	FiM	RBASP	FiM	RBASP	FiM	RBASP	FiM	Act.
BT	1.98	1.04	2.17	10.95	66.45	18.07	136	360	360
EP	0.44	2.13	0.64	5.31	76.74	15.32	309	252	248
SP	1.98	0.21	16.55	1.61	94.21	19.14	212	64	56
LU	0.14	0.06	23.48	6.01	79.15	26.88	82	28	20
IS	1.73	1.3	22.59	6.75	90.94	34.36	70	70	70
CG	2.63	2.99	26.17	5.32	77.65	40.73	156	102	102
PR	2.02	0.91	44.48	6.81	57.65	31.46	21	56	64
KM	1.61	0.42	9.37	1.98	30.43	10.87	64	192	192

be either a linear function or a quadratic function,

$$\log_2(y) = \beta_0 + \beta_1 x_1 + \dots + \beta_n x_n + g(q). \quad (17)$$

While reproducing the RBASP model, we use $p_0 = 1, 2, 4$ as the training set and predict the performance with two forms of $g(q)$ function as suggested in Reference [5]. The RBASP approach directly predicts the execution time using regression, which requires performing training with data points processed for a different number of multiple processes. In contrast, FiM extrapolates the hardware parameters for a given computing platform and then uses these hardware parameters as the inputs to the stochastic Markov model for predicting execution time for a different number of processes. FiM does not need to be trained again when we change the number of processes used in a given computing platform.

Figure 5 shows the predicted results using RBASP and FiM for six NPB benchmarks and two iterative data processing applications. We run these experiments on the C5 platform (see its configuration in Table 2), using the actual optimal number of processes listed in Table 4. As shown in Figure 5, our FiM approach achieves a pretty good agreement between the predicted and actual results across all the six benchmarks and two applications. We also observe that RBASP has lower relative prediction error than FiM for only BT and EP. Both BT and EP are compute-intensive benchmarks, so a pure regression technique is sufficient to predict their execution time. However, prediction using only regression is not good enough for I/O and memory intensive applications as observed from RBASP prediction results in Figure 5 for the rest of the applications. As shown in

Figure 5, for all the remaining benchmarks and applications, FiM performs better. Many real-time applications are I/O or memory intensive, for whom simple regression model is not sufficient to give accurate predictions. For the results shown in Figure 5, the calculation time reported includes all background delay that occurs due to I/O latency to access the data that are required to perform the computations. The time required for data transfer among different parallel processes is communication time. We also observe that the prediction error of FiM remains less than 20% for individual prediction of calculation time and communication time. Furthermore, RBASP's prediction is limited to the fixed application parameters on which the model is trained, because if the application parameters are changed, then RBASP needs to be re-trained. In contrast, FiM can predict execution time without any prior training for new application parameters, because we do not regress the execution time directly, but instead, we train our model to learn the change in system counters like total cycles and stalled cycles when application parameters are changed. Thus, this also advanced our approach to avoid over-fitting to the execution time of the training datasets.

Table 4 lists the best, average and worst prediction errors as well as the actual and predicted optimal numbers of processes using RBASP and FiM. Apart from having a lower average error, also having a tight prediction error range is important for these prediction approaches, because such a range can be used to provide a quick approximation before conducting actual experiments. We observe that FiM has a relatively tight prediction error range from the best to the worst, compared to RBASP. Despite the lower prediction error under the best case, RBASP obtains higher prediction errors in the worst case for all benchmarks and applications. This is because the pure regression model used in RBASP has poor adaptability to changes in the values of attributes (e.g., number of processes). FiM provides tighter error bounds, which is very important for such a quick estimation modeling technique. We further observe that the optimal number of processes predicted by FiM is very close to the actual one. We also rank the number of processes according to the actual and predicted results from FiM and calculate the correlation¹ between the actual and predicted rankings. We obtain a high correlation in the range of 0.80 to 0.99, which indicates considerable accuracy of our FiM estimation technique.

We further use the Chi-square goodness-of-fit test [29] to evaluate how well the predicted runtime distribution matches with the actual measured runtime distribution with varying application parameters and the number of processes. Our null hypothesis is that the predicted data are not consistent with actual distribution. The significance level of our test is 0.1, meaning that the deviation of the predicted value is not more than 10%. The p -value² that we obtain for our test is equal to 0.08. Since the p -value (0.08) is less than the significance level (0.1), we cannot accept the null hypothesis. This indicates reasonably high confidence of less than 10% error between the actual and predicted values.

4.1.1 Individual Prediction Results. We carefully study the effectiveness of our model for predicting execution time with the increasing number of processes. Figure 6 shows the results for FiM per data point (e.g., per pixel in k -means) when operated with a dataset consisting of 13 million data points after getting trained using three datasets with less than 1,000 data points. These experiments are performed using the C5 hardware configuration of Table 2. We conduct all experiments for 1,000 times and measure the average runtime as well as the minimum and maximum actual runtime obtained in these 1,000 runs, which are shown by the range bars. From Figure 6(a), we observe that the actual calculation time keeps decreasing when we have more simultaneous processes running. We also observe that 192 is the optimal number of processes. Increasing the

¹A correlation between actual and predicted ranks describes the degree of agreement between them. Correlation ranges between -1 and 1 with 1 being the best; higher correlation signifies better accuracy of predicted results.

²The p -value is a statistical measure of the deviation of the actual distribution from the hypothesis.

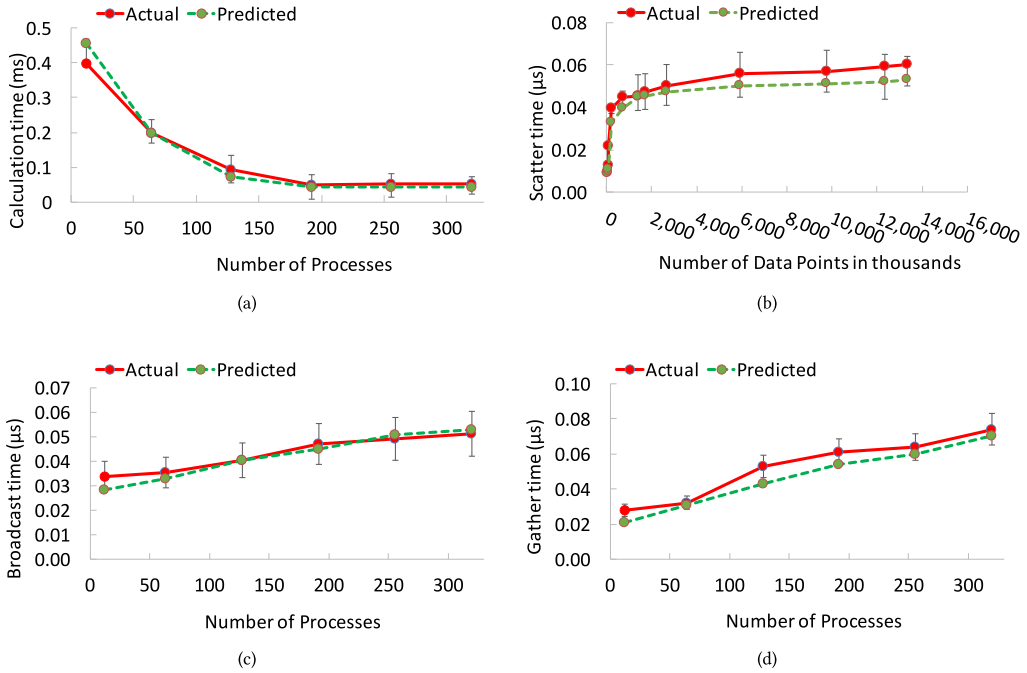


Fig. 6. Actual and predicted time - (a) Calculation (b) Scatter (c) Broadcast (d) Gather.

number of processes further above 192 does not give any performance improvement. Our model can accurately predict this optimal number of processes.

Figure 6(b) shows the results of FiM-Com for the scatter pattern. For scatter of “ n ” data points over “ p ” processes, “ n/p ” data points need to be transferred to each process. For scatter, each process receives a unique subset of the dataset and thus, with the increase in the number of processes, the number of data points to be transferred in total does not increase. Hence, the runtime of the scatter communication depends mainly on the size of input datasets and not on the number of processes. So, we evaluate the FiM-Com scatter model under different datasets of different sizes. The results are shown in Figure 6(b) are obtained using 192 processes and for datasets with the different number of data points mentioned on the x -axis in thousands. The smallest dataset consists of one thousand data points and the largest dataset consists of 13 million data points. We observe that, when we have relatively small datasets, the scatter time per data point increases rapidly with the increase in dataset size until 600 thousand data points scattered to 192 processes. However, for larger datasets, the scattering time per data point remains almost constant or increases very slowly. Each data point is of 20B, thus for less than or equal to 600 thousand data points scattered over 192 processes, less than or equal to 64KB is required to be transferred to each process. Note that the eager protocol message size is set to 64KB with environment variable `MP_EAGER_LIMIT`. Accordingly, the default receive buffer size is increased with environment variable `MP_BUFFER_MEM`. Hence, for a data transfer smaller than 64KB, the communication network follows the eager protocol. However, for a large data transfer, the communication network follows the rendezvous protocol [35] to perform sender-receiver handshake. Our scatter model is designed to capture the effect of this protocol shift.

Figure 6(c) shows the results of FiM-Com for broadcast as a function of the number of processes. We see that data transfer of each data point consumes more time as the number of processes

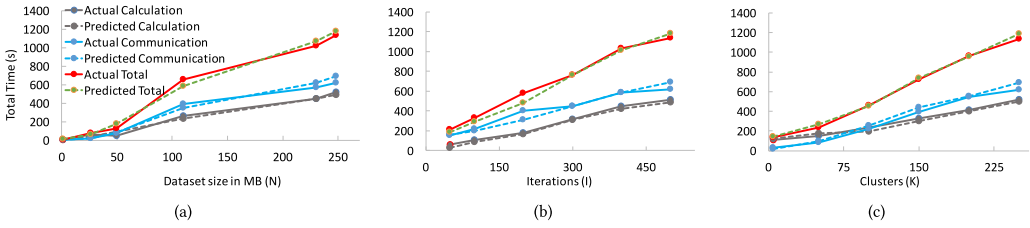


Fig. 7. Sensitivity analysis - (a) Dataset size (b) Number of Iterations (c) Number of Clusters.

increases. This is because for broadcast, with each additional process, the number of data points to be transmitted also increases unlike scatter. The increase in data packets incurs an additional load that in turn increases the average broadcast time of each data point. Last, the results under the uplink gather pattern are plotted in Figure 6(d) with respect to the number of processes for each data point. The gathering time increases linearly with an increase in the number of processes due to the increased congestion in the communication network, with more data points to be received by the data collecting node. From Figure 6, we observe that our FiM models accurately predict the calculation and communication times under different communication patterns. Also, the predicted results mostly remain in the range bars (i.e., the minimum to the maximum) of the actual observations. All these graphs show that FiM estimates are very accurate.

4.2 Sensitivity Analysis

One significant contribution of our modeling techniques is to accurately predict for large datasets by using only small datasets to train and calibrate the models. In our experiments, we use three small datasets as the training ones to collect data for model calibration. For a new application, we need first to train to derive a new model. However, for new sets of application parameters and new datasets (including both small and large ones) of the same application, the derived model can be used to predict the runtime. Therefore, we perform a sensitivity analysis of different dataset sizes and application parameters. We argue that if the user has the flexibility to choose application parameters for achieving optimal performance, our model then can provide useful guidance by helping the user to decide appropriate parameters.

For example, k -means processing with more iterations and more clusters can provide better clustering results and accuracy but consume more time. Therefore, it would be useful to use FiM to estimate the execution time with respect to the increase in the number of iterations and number of clusters to determine how much extra latency is needed to achieve better accuracy. The results of execution time for the k -means algorithm as a function of (a) dataset size, (b) the number of iterations, and (c) number of clusters are plotted in Figure 7. Figure 7 also shows the actual and predicted calculation and communication time. We experiment with different hardware configurations as shown in Table 2, and present the results of the C5 configuration here. In these experiments, we also use the predicted optimal number of processes listed in Table 4. For each plot in Figure 7, we do a sensitivity analysis on one parameter and fix the remaining two parameters with dataset size of 250MB, 500 iterations and 250 clusters. We observe that our models can accurately predict the execution time even when the datasets become large, see Figure 7(a). Note that we only use small datasets to train our models. Figure 7(b) shows a linear increase of execution time with increasing number of iterations. Figure 7(c) further shows execution time with respect to the increase in number of clusters. Summarizing from Figure 7, we can see that FiM consistently achieves predicted results in good agreement with actual ones under different application parameters like dataset size, number of iterations and number of clusters.

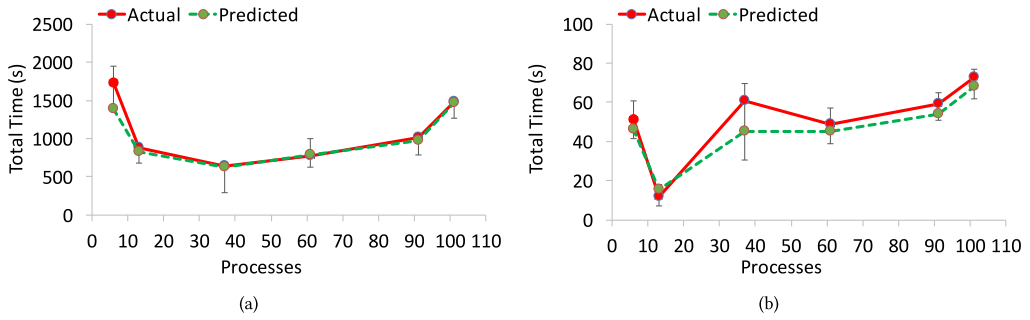


Fig. 8. Sensitivity analysis w.r.t. number of MPI processes for (a) *k*-means and (b) Pagerank.

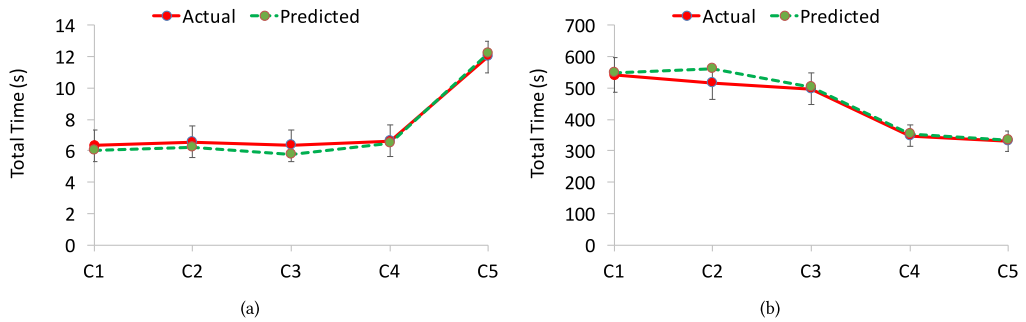


Fig. 9. Sensitivity analysis w.r.t. distribution of cores under (a) a small dataset with 40 vertices and (b) a large dataset with 4,039 vertices for Pagerank application.

We further evaluate the prediction of our models under different hardware configurations. A distributed computing platform deployed using MPI offers different choices in the number of parallel processes and the distribution of cores (e.g., C1-C5 listed in Table 2). Figure 8 shows the actual and predicted execution times with respect to the number of parallel processes for (a) *k*-means and (b) Pagerank, respectively. We can see that the best performance (i.e., the shortest execution time) is achieved in the middle range of processes, e.g., 38 for *k*-means and 12 for Pagerank. FiM is able to predict the optimal performance for both applications accurately. Predicted results match well with actual ones across the different number of processes.

Figure 9 plots the predicted results for Pagerank under five different hardware configurations listed in Table 2. To be able to predict across such heterogeneous platforms, we calibrate our model for data compute and data transfer among each type of available hardware. We observe that the first three configurations (C1, C2, and C3) achieve a shorter execution time for a small dataset (see Figure 9(a)). When we have large datasets, C4 and C5 with more distributed cores are better (see Figure 9(b)). FiM can accurately predict such performance trends, i.e., Pagerank becomes more scalable on the higher number of distributed cores for larger datasets. These estimation results can thus provide us with insightful data regarding the scalability of an application on a multi-core computing platform.

Here, we evaluate our technique by individually varying each variable parameter. From the results presented above, we see that this sensitivity analysis helps to observe that prediction accuracy of our model remains intact with changing application parameters, number of parallel processes and different hardware platforms.

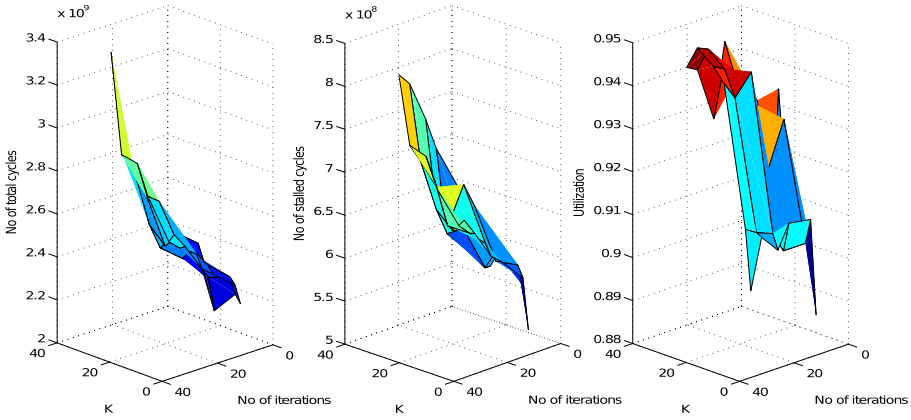


Fig. 10. Regression mapping (K - number of clusters, I - number of iterations).

4.3 Validation

In this section, we present the validation for considering the linear regression model to extrapolate hardware variables such as TC , SC , and U . In particular, we show results using k -means clustering as a representative. Recall, for k -means, we chose linear regression to extrapolate hardware variables as a function of application parameters (I , N , and K), see Section 2. There are a variety of regression models that can be used. It is not straightforward to choose the right regression model that is best suited to our requirement. Even a complex model might over-fit the training data, generating a substantial prediction error on other datasets. However, a simple model may underestimate the learning trends and produce incorrect predicted results [39]. Considering these two cases, we first investigate the learning trend of three hardware variables (TC , SC , and U) under different settings of application parameters. We choose the linear regression model after examining other regression models such as a piece-wise linear model, Poisson models and quadratic models with different degrees of the polynomial.

We investigate the learning trend of three hardware variables under different settings of application parameters. We show the hardware variables such as TC , SC , and U for k -means clustering as a representative. Figure 10 depicts the resulting surface of each hardware variable as a function of application parameters (e.g., I and K for the k -means application). Similar results can be obtained for other combinations of application parameters, such as (I , N) and (N , K). In Figure 10, linear surfaces can be found for different hardware variables. Thus, we conclude that hardware variables TC , SC , and U , linearly depend on the increment in application parameters (such as K , I , and N). These results confirm the use of the linear regression mapping in our machine-learning approach.

We also investigate the calibration of our communication models, i.e., the training results of the service rates for each server in the queuing models, see Section 3.2. In Figure 11, we plot the CDFs of the measured communication service times (i.e., $1/\mu_2$, $1/\mu_3$ and $1/\mu_4$) for the three components (i.e., metadata transfer (M), network initialization (C) and data fetch time (R)) in the communication. The predicted service times drawn from our calibrated service processes (i.e., Equations (14), (15), and (16)) are also plotted in Figure 11. We can see that the predicted service time distributions well capture the actual service times.

The Chi-square goodness-of-fit test [29] is used to evaluate how the three derived statistical distributions (i.e., uniform, logarithmic and exponential) for μ_2 , μ_3 , and μ_4 fit the actual measurements. The chi-squared test is used to determine whether there is a significant difference between

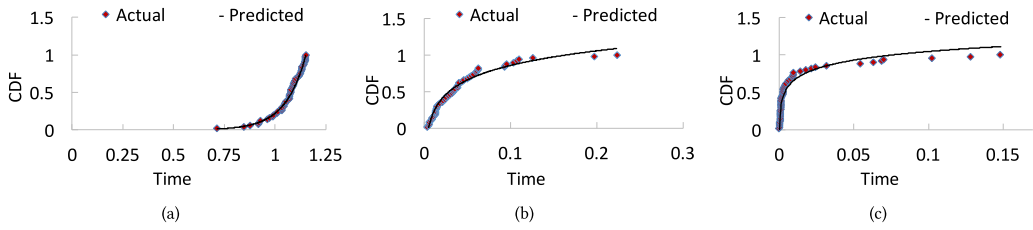


Fig. 11. Cumulative distributed functions (CDFs) of the actual and predicted service times (per job) for (a) metadata transfer, (b) network initialization, and (c) data fetching.

the expected distribution and the observed distribution. Our null hypothesis is that the derived statistical distributions are not consistent with actual distribution. The significance level of our test is 0.15, meaning that the maximum deviation of the predicted distribution is not more than 15%. The p -values³ that we obtain for service rates μ_2 , μ_3 , and μ_4 are equal to 0.08, 0.11, and 0.14, respectively. Since the p -values are less than the significance level (0.15), we reject the null hypothesis. This indicates reasonably high confidence of not more than 15% error between the actual and predicted values.

5 RELATED WORK

Modeling helps to shorten the development cycle by providing the necessary insights to obtain optimal performance. Performance modeling can be approached in several different ways, including empirical evaluation [17, 18], simulation [9, 26, 28] and analytical modeling [4, 15]. Empirical evaluation is a technique for gaining knowledge about a system from observations of experiments. This requires the exact implementation as well as similar hardware to the target, because results are based on observed ground truth. Empirical techniques were popular in the past when computer hardware was stable over long periods. They give result fast and accurately for the datasets similar to training.

Simulators model hardware such as memory hierarchy, communication buses, parallel ports, and accelerators [31, 33, 40]. They evaluate each block of the given codes, similar to the manner that it is executed on the target machine. Thus, they require source codes while performing prediction. Although simulators like SimpleScalar [3] and CACTI [38] can predict with high accuracy, they also consume a long time to give predicted results. Their slow running time and infrastructure cost are major drawbacks.

Analytical modeling is the technique of building a set of equations to show the high-level abstraction of the behavior of an application and a hardware architecture. This type of modeling can be evaluated quickly and easily as it is reduced to the form of the set of equations to be solved. Analytical models can be flexible and scalable but are comparatively less accurate than empirical and simulation-based models, because they lack accurate hardware machine models. The major drawback of analytical modeling is that it limits the scope of prediction. These models will give high prediction errors if tested with parameters that were not captured when building the model equations. An innovative idea is the combination of the above models, e.g., COMPASS [25]. It gives good accuracy but requires the compilation of source code for each new test dataset as well as the conversion of source codes to ASPEN [36]. COMPASS requires the hardware machine model to be formed not only for training but also for testing, which is quite time consuming, especially for large datasets.

³The p -value is a statistical measure of deviation of the actual distribution from the hypothesis.

Predicting the performance of any parallel processing platform consists of two major phases, i.e., calculation and communication. The Markov chain modeling using probabilistic distribution assists in predicting the calculation load of multi-process and multi-core architectures [12, 27, 32]. These approaches model systems in the form of equations, where changes to the code or data require changes to the equations. The above process can be time-consuming when we desire to predict for a range of parameters. Some analytical models [13, 14, 21, 22, 34] require the conversion of source code into a control flow chart for the ease of framing equations. Our model predicts the performance of an application on a range of input parameters without requiring a new set of equations, as FiM uses a machine-learning model to learn hardware parameters. Ad hoc analytical models and structured analytical models have been developed to predict the network communication behavior of an application [11]. They use predefined models such as timeline diagrams showing various network overheads. This type of models cannot efficiently capture different patterns like scatter, broadcast and gather. The concept of using queuing theory to model such communication and predict the expected communication time is a novel idea described in our article.

6 CONCLUSIONS

We present a novel performance modeling technique (FiM) to predict the execution time of iterative multi-stage data processing applications running on parallel computing paradigm. We combine different modeling techniques, such as stochastic Markov modeling, machine-learning techniques and queuing theory, to predict the end-to-end execution time. FiM estimates the time required for both data calculation and data communication across a range of input datasets, application configurations and parallel hardware parameters such as number of processes. We demonstrate that FiM helps system designers and application programmers choose optimal hardware parameters and application parameters. More importantly, our prediction models are parameterized using small datasets but can predict accurately for large datasets. We evaluated our approach using NAS Parallel Benchmarks and real iterative data processing applications. We rank the number of processes according to the actual and predicted results from FiM and calculate the correlation between the actual and predicted rankings. Our results show that FiM obtains high correlation in the range of 0.80 to 0.99, which indicates considerable accuracy of our technique. In the future, we plan to expand the scope of our prediction model to investigate other communication patterns like all-to-all communication. We will also capture scenarios where there is an overlap between the communication and computation phases. Large-scale computing platforms, including GPUs, will further be considered as a target environment.

REFERENCES

- [1] [n.d.]. Information Technology Services—Research Computing. Retrieved from <https://www.northeastern.edu/rc/>.
- [2] [n.d.]. NASA Advanced Supercomputing Division, NAS Parallel Benchmarks. Retrieved from <http://www.nas.nasa.gov/publications/npb.html>.
- [3] T. Austin, E. Larson, and D. Ernst. 2002. SimpleScalar: An infrastructure for computer system modeling. *Computer 2* (2002), 59–67.
- [4] David H. Bailey and Allan Snaveley. 2005. Performance modeling: Understanding the past and predicting the future. In *Proceedings of the European Conference on Parallel Processing*. Springer, 185–195.
- [5] Bradley J. Barnes, Barry Rountree, David Lowenthal, Jaxk Reeves, Bronis De Supinski, and Martin Schulz. 2008. A regression-based approach to scalability prediction. In *Proceedings of the 22nd Annual International Conference on Supercomputing*. ACM, 368–377.
- [6] Janki Bhimani, Miriam Leeser, and Ningfang Mi. 2015. Accelerating K-means clustering with parallel implementations and GPU computing. In *Proceedings of the High Performance Extreme Computing Conference (HPEC'15)*. IEEE, 1–6.
- [7] Janki Bhimani, Ningfang Mi, and Miriam Leeser. 2016. Performance prediction techniques for scalable large data processing in distributed MPI systems. In *Proceedings of the IEEE 35th International Performance Computing and Communications Conference (IPCCC'16)*. IEEE, 1–2.

- [8] Janki Bhimani, Ningfang Mi, Miriam Leeser, and Zhengyu Yang. 2017. FiM: Performance prediction for parallel computation in iterative data processing applications. In *Proceedings of the IEEE 10th International Conference on Cloud Computing (CLOUD'17)*. IEEE, 359–366.
- [9] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arka Prava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH Comput. Arch. News* 39, 2 (2011), 1–7.
- [10] Henri Casanova. 2001. Simgrid: A toolkit for the simulation of application scheduling. In *Proceedings of the 1st IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001*. IEEE, 430–437.
- [11] WenGuang Chen, JiDong Zhai, Jin Zhang, and WeiMin Zheng. 2009. LogGPO: An accurate communication model for performance prediction of MPI programs. *Sci. Chin. Ser. F: Inf. Sci.* 52, 10 (2009), 1785–1791.
- [12] Xi E. Chen and Tor M. Aamodt. 2009. A first-order fine-grained multithreaded throughput model. In *Proceedings of the IEEE 15th International Symposium on High Performance Computer Architecture (HPCA'09)*. IEEE, 329–340.
- [13] Gopinath Chennupati, Nandakishore Santhi, Robert Bird, Sunil Thulasidasan, Abdel-Hameed A. Badawy, Satyajayant Misra, and Stephan Eidenbenz. 2017. A scalable analytical memory model for CPU performance prediction. In *Proceedings of the 8th International Workshop on High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation, PMBS (PMBS@SC'17)*. Springer, 114–135.
- [14] G. Chennupati, N. Santhi, S. Eidenbenz, and S. Thulasidasan. 2017. An analytical memory hierarchy model for performance prediction. In *Proceedings of the 2017 Winter Simulation Conference (WSC'17)*. 908–919.
- [15] Gopinath Chennupati, Nanadakishore Santhi, Stephen Eidenbenz, Robert Joseph Zerr, Massimiliano Rosa, Richard James Zamora, Eun Jung Park, Balasubramanya T. Nadiga, Jason Liu, Kishwar Ahmed, and Mohammad Abu Obaida. 2017. *Performance Prediction Toolkit (PPT)*. Los Alamos National Laboratory (LANL). Retrieved from <https://github.com/lanl/PPT>.
- [16] Arnaldo Carvalho de Melo. 2010. The new linux perf tools. In *Proceedings of Linux Kongress*, vol. 18.
- [17] Steven Fortune and James Wyllie. 1978. Parallelism in random access machines. In *Proceedings of the 10th Annual ACM Symposium on Theory of Computing*. ACM, 114–118.
- [18] Phillip Gibbons, Yossi Matias, and Vijaya Ramachandran. 1998. The queue-read queue-write PRAM model: Accounting for contention in parallel algorithms. *SIAM J. Comput.* 28, 2 (1998), 733–769.
- [19] A. Chan, W. Gropp, and E. Lusk. User guide for MPE: Extensions for MPI programs. Technical report, ANL-98. Argonne National Laboratory.
- [20] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. 2010. LogGOPSim: Simulating large-scale applications in the LogGOPS model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM, 597–604.
- [21] Tanzima Z. Islam, Jayaraman J. Thiagarajan, Abhinav Bhatle, Martin Schulz, and Todd Gamblin. 2016. A machine learning framework for performance coverage analysis of proxy applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 46:1–46:12.
- [22] Nikhil Jain, Abhinav Bhatle, Michael P. Robson, Todd Gamblin, and Laxmikant V. Kale. 2013. Predicting application performance using supervised learning on communication features. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. ACM, 95:1–95:12.
- [23] Curtis L. Janssen, Helgi Adalsteinsson, Scott Cranford, Joseph P. Kenny, Ali Pinar, David A. Evensky, and Jackson Mayo. 2012. A simulator for large-scale parallel computer architectures. *International Journal of Distributed Systems and Technologies (IJ DST)* 1, 2 (2010), 57–73.
- [24] Darren Kerbyson, Henry Alme, Adolfo Hoisie, Fabrizio Petrini, Harvey Wasserman, and Mike Gittings. 2001. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing*. ACM, 37–37.
- [25] Seyong Lee, Jeremy Meredith, and Jeffrey Vetter. 2015. COMPASS: A framework for automated performance modeling and prediction. In *Proceedings of the 29th ACM on International Conference on Supercomputing*. ACM, 405–414.
- [26] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. 2002. Simics: A full system simulation platform. *Computer* 35, 2 (2002), 50–58.
- [27] Reshmi Mitra, Bharat S. Joshi, Arun Ravindran, Arindam Mukherjee, and Ryan Adams. 2012. Performance modeling of shared memory multiple issue multicore machines. In *Proceedings of the 2012 41st International Conference on Parallel Processing Workshops*. IEEE, 464–473.
- [28] Avadh Patel, Furat Afram, Shunfei Chen, and Kanad Ghose. 2011. MARSS: A full system simulator for multicore x86 CPUs. In *Proceedings of the 48th Design Automation Conference*. ACM, 1050–1055.
- [29] D. Cox. 2002. Karl Pearson and the chi-squared test. In *Goodness-of-Fit Tests and Model Validity*. Springer, 3–8.
- [30] Juan-Antonio Rico-Gallego, Juan-Carlos Diaz-Martín, and Alexey L. Lastovetsky. 2016. Extending τ -Lop to model concurrent MPI communications in multicore clusters. *Fut. Gener. Comput. Syst.* 61 (2016), 66–82.

- [31] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. 2011. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.* 38, 4 (2011), 37–42.
- [32] Rafael H. Saavedra-Barrera and David E. Culler. 1991. *An Analytical Solution for a Markov Chain Modeling Multi-threaded*. Technical Report. Citeseer, Berkeley, CA.
- [33] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A pre-rtl, power-performance accelerator simulator enabling large design space exploration of customized architectures. In *Proceedings of the ACM SIGARCH Computer Architecture News*, Vol. 42. IEEE Press, 97–108.
- [34] Allan Snavey, Laura Carrington, Nicole Wolter, Jesus Labarta, Rosa Badia, and Avi Purkayastha. 2002. A framework for performance modeling and prediction. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC'02)*. IEEE, 1–17.
- [35] Marc Snir. 1998. *MPI—The Complete Reference: The MPI Core*. Vol. 1. MIT Press.
- [36] Kyle Spafford and Jeffrey Vetter. 2012. Aspen: A domain specific language for performance modeling. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 84.
- [37] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM* 52, 4 (2009), 65–76.
- [38] Steven Wilton and Norman Jouppi. 1996. CACTI: An enhanced cache access and cycle time model. *IEEE J. Solid-State Circ.* 31, 5 (1996), 677–688.
- [39] Ming Yuan and Yi Lin. 2006. Model selection and estimation in regression with grouped variables. *J. Roy. Statist. Soc. B* 68, 1 (2006), 49–67.
- [40] G. Zheng, Gunavardhan Kakulapati, and L. V. Kale. 2004. BigSim: A parallel simulator for performance prediction of extremely large parallel machines. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium, 2004*.

Received January 2018; revised December 2018; accepted January 2019

MULTI-CLOCK: Dynamic Tiering for Hybrid Memory Systems

Adnan Maruf*, Ashikee Ghosh*, Janki Bhimani*, Daniel Campello[†], Andy Rudoff[‡], and Raju Rangaswami*

*Knight Foundation School of Computing and Information Sciences, Florida International University

[†]Google, [‡]Intel Corporation

Abstract—

The rapid growth of in-memory computing powered by data-intensive applications has increased the demand for DRAM in servers. However, a DRAM-based system can be limiting for modern workloads because of its capacity, cost, and power consumption characteristics. Hybrid memory systems, which consist of different types of memory, such as DRAM and persistent memory, can help address many of these limitations. One promising direction that has been explored in the recent literature involves introducing persistent memory devices as a second memory tier that is directly exposed to the CPU. The resulting tiered memory design must address the fundamental challenge of placing the right data in the right memory tier at the right time while minimizing overhead. We present MULTI-CLOCK, an efficient, low-overhead hybrid memory system that relies on a unique page selection technique for tier placement. MULTI-CLOCK’s page selection captures both page access *recency* and *frequency*, and enables moving pages to appropriate tiers at the right time within hybrid memory systems. We implemented a Linux-based, NUMA-aware version of MULTI-CLOCK that is entirely transparent and backward compatible with any existing application. Our evaluation with diverse real-world applications such as graph processing and key-value stores shows that MULTI-CLOCK can improve the average throughput by as much as 352% when compared with several state-of-the-art techniques for tiered memory.

I. INTRODUCTION

Over the last several decades, DRAM performance and capacity have followed Moore’s Law and thus kept up with advances in CPU technology. However, DRAM-based memory systems have two significant drawbacks — cost and power consumption. These drawbacks impact their usage in both enterprise servers and mobile systems. Most new generation applications are inherently memory-intensive, whereby workloads demand access to high-performance yet low-cost memory systems [1]–[4]. A complementary technological change is the imminent availability of higher capacity and lower power consuming byte-addressable persistent memory (PM) technologies [5]–[7]. These new memories offer latency and bandwidth for byte-addressable access that are within an order of magnitude of those for DRAM [8], [9], with power consumption being lower by 4-29x compared to DRAM [8]. These characteristics make the use of PM to extend the main memory attractive. When using PM as the main memory, its persistence capability becomes irrelevant, thereby entirely avoiding its biggest performance overhead [10].

One appealing use of persistent memory is as a new tier in a hybrid multi-tier memory system with tiers ordered from

high performance - low capacity to *low performance - high capacity*. This approach allows applications to access their data directly from persistent memory without first paging into DRAM. However, managing persistent memory simply as additional available memory (i.e., static tiering) could compromise the effectiveness of the tiered memory system. Once an application has exhausted higher performance tier resources, future allocations for that application or any other application on the system will have to be serviced from lower performance tiers. Additionally, such allocations continue to reside in lower performance tiers regardless of how important the data becomes over its lifetime. Thus, the primary challenge in building an efficient hybrid memory system is the dynamic placement of pages in appropriate tiers. From a system design standpoint, addressing this challenge translates to understanding the relative importance of pages, identifying misplaced pages in either tier and moving such pages to their optimal tier, all while controlling the overhead of these operations. The major contributions of our work are:

- We design an efficient page selection method for dynamic page movement across memory tiers. This method enables identifying pages suitable for specific memory tiers in an online fashion, thereby adapting to the workload.
- We propose MULTI-CLOCK, a solution based on dynamic tiering that overcomes the limitations of static tiering and extends the system’s memory with improved performance without sacrificing DRAM capacity.
- We develop a real-system prototype implementation of MULTI-CLOCK using Linux version 5.3.1 by extending the kernel’s page reclamation algorithm to include its dynamic page migration logic.
- We evaluate the performance of our prototype using diverse workloads including graph analytics and key-value stores to compare MULTI-CLOCK with existing solutions.

We evaluate MULTI-CLOCK against Nimble [11], AutoTiering [12], and Memory-mode [7]. Our evaluation with YCSB workloads [13] using a Memcached [3] backend and with GAPBS [14], a graph processing benchmark, shows that MULTI-CLOCK provides up to 132% higher performance compared with static tiering and up to 352% compared with other state-of-art solutions such as Nimble [11], AutoTiering [12], and Memory-mode [7]. From these experiments, we find that the page selection mechanism in dynamic tiered memory systems is of critical importance. We also demonstrate that

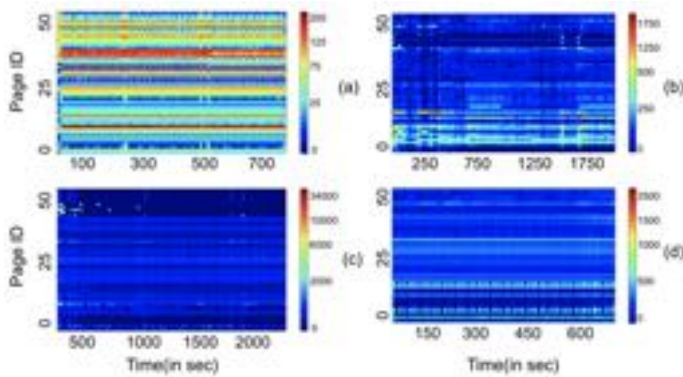


Fig. 1. **Heat-map of pages access frequencies** depicts access frequencies of the sampled pages in (a) RUBiS OLTP, (b) SPECpower (OLTP), (c) Dacapo *xalan* and (d) Dacapo *lusearch*.

the state-of-the-art page selection mechanisms do not consider page access frequency distributions for identifying page migration candidates, and we demonstrate that doing so is vital for optimizing performance in a tiered memory system.

II. MOTIVATION

Integrating persistent memory (PM) devices into existing systems force a rethink of the system architecture. Due to the relatively high read and write latency and low bandwidth, using only PM as the main memory is not ideal. On the other hand, a hybrid memory system with DRAM and PM can deliver both high throughput combined and increased capacity. However, designing a memory hierarchy with PM to improve the performance of applications is non-trivial.

One promising approach of utilizing PM is to configure both DRAM and PM as separate tiers in a multi-tier memory system. With tiering, data residing in the byte-addressable PM is treated as resident in the main memory and directly addressable by the CPU. Tiers represent disjoint sets of memory frames. The operating system identifies which frames belong to each memory type and assigns them to their proper tier. Tiers can be arranged in a specific order, following the characteristics of the different types of memory from Higher Tier - *high performance - low capacity* to Lower Tier - *low performance - high capacity*, to service memory allocations.

In this section, we discuss the diversity in the access patterns of pages across applications. We also discuss why the careful selection of candidate pages for specific tiers based on both the frequency and recency is pivotal for performance. We also discuss the existing solutions for the DRAM-PM tiered memory systems and their limitations.

A. Diversity in Page Access Patterns

Let us consider a simple tiered memory system wherein pages are first allocated (or get "born in") in the DRAM tier. When the system starts running low on free space in DRAM, the system starts demoting less frequently accessed pages to the PM tier to free up DRAM space for new allocations. Without an available promotion mechanism, a demoted page

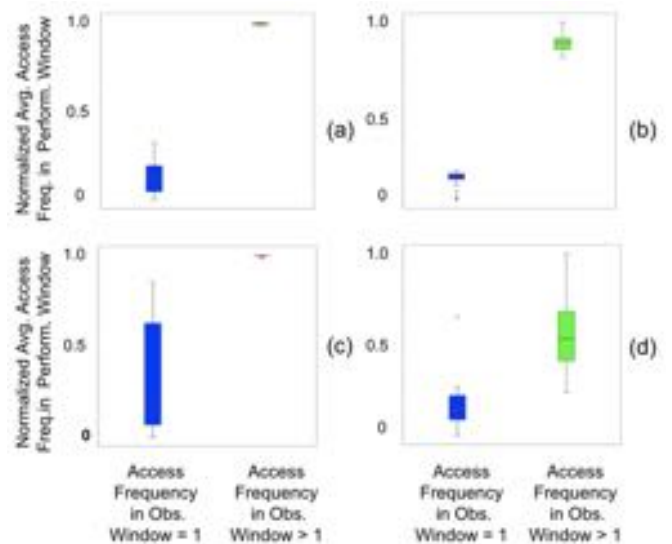


Fig. 2. **Distribution of access frequencies for different page types** depicts the distribution of access frequencies in the performance windows for the two types of pages: pages that were accessed only once during the observation window and pages that were accessed multiple times in the observation window. Workloads for the experiment: (a) RUBiS OLTP, (b) SPECpower (OLTP), (c) Dacapo *xalan* and (d) Dacapo *lusearch*.

would reside in the PM tier for the rest of its lifetime. If a significant number of demoted pages get frequently accessed post-demotion, a complementary promotion mechanism that allows demoted pages to move back to the DRAM tier may result in better system performance. However, a tiered system with the facility of promoting pages from PM to DRAM can improve performance only if promoted pages are accessed relatively more frequently for a reasonable amount of time afterward. To evaluate the potential for a promotion mechanism in improving workload performance, we recorded the access patterns of pages in memory over time within applications. To keep the overhead tractable, we randomly sampled pages from memory, assigned them unique identifiers, and traced the accesses to these sampled pages.

In Figure 1, the heatmap depicts the frequency of accesses of the sampled pages for the execution periods of four workloads from different benchmarks: (a) RUBiS OLTP benchmark [15], (b) SPECpower (OLTP) [16] running at 80% of the maximum throughput, (c) Dacapo *xalan* (XML to HTML conversation) and (d) Dacapo *lusearch* (searching keyword over a corpus of data using lucene) [17]. On the Y axis, 50 sampled pages are sorted in ascending identifier order. The x axis represents execution time. Each block of the heatmap shows the intensity of the access frequency for a particular page for a particular time segment. The heat maps indicate fairly diverse access patterns for the sampled pages. Some pages show frequent accesses throughout the execution period. We denote these pages by *DRAM friendly pages* which should always reside in DRAM. Other pages have very infrequent accesses over the entire execution time. The total number of accesses from

Tiering Technique	Page Access Tracking	Page Selection		NUMA Aware	Space Overhead	Generality	Evaluation	Usability Limitation	Key Insight
		Promotion	Demotion						
Static-Tiering	N/A	N/A	N/A	Yes	N/A	All	PM	None	Straight forward
Thermostat	Software Page Fault	N/A	Frequency	No	Yes	Huge Page	Emulator (KVM)	Not Open Source	Poisoning huge pages
AutoNUMA-Tiering	Software Page Fault	Recency	N/A	Yes	Yes	All	PM	Config. NUMA Paths	NUMA balancing
AutoTiering	Software Page Fault	Recency	Frequency	Yes	Yes	All	PM	Config. NUMA Paths	Maintain N-bit history for demotion
Nimble	Reference Bit	Recency	Recency	No	No	All	Emulator	Config. Launcher	Optimize huge page migrations
AMP	Reference Bit	Recency+ Frequency+ Random	Recency	No	Yes	Huge Page	Emulator (QEMU)	No KMEM DAX Support	Hybrid page selection
MULTI-CLOCK	Reference Bit	Recency+ Frequency	Recency	Yes	No	All	PM	None	Low overhead Recency/Frequency

TABLE I
Comparison of existing memory tiering techniques.

these pages is very low compared to the total access count during the execution. Thus, the tier residence of these pages does not significantly impact the overall performance. Apart from these two types of pages, we see that certain pages can significantly benefit a tiered memory system. These *Tier friendly pages* show bimodal access behavior whereby for some time segments they get accessed at a much higher rate than other time segments. If these pages can be identified by analyzing their access patterns and moved to the DRAM tier when they start to get accessed at a higher rate, the overall application performance can potentially be improved. Thus, our core motivation for a dynamic tiering system is driven by two main observations: (a) the importance of pages changes over time, and (b) at any given time, the importance of different pages in the system can vary significantly.

Next, we investigate the importance of frequency of accesses along with the recency for identifying *Tier friendly pages*. Recent works such as Nimble [11] select pages only based on the recency since capturing frequency on the real system with minimal overhead is challenging. To understand the access frequency of pages, we divide the whole execution period of the workloads that were used in the experiment in Figure 1 into multiple sets of *observation windows* followed by *performance windows*. We divide sampled pages that were accessed into two defined categories: pages that were accessed only once during that particular observation window and pages that were accessed multiple times. Finally, we measure their accesses in the next performance window, and we follow the same procedure for all (observation window, performance window) pairs. In the frequency distribution shown in Figure 2, we can notice that pages that were accessed multiple times in the observation windows are accessed with a much higher frequency on average in the performance windows compared to the pages that were accessed only once. This suggests that pages with higher frequency in some observation windows have a higher probability of getting accessed in the next performance window.

B. Persistent Memory in Memory-mode

Persistent memory in Memory-mode is a natively system-supported solution for using PM as memory. It is implemented in recent memory controllers that support PM and by recent operating systems that support PM DIMMs [7]. In Memory-mode, DRAM is directly mapped as the cache for data stored in PM and used as the last level cache in addition to the L1/L2/L3 caches. The system recognizes only the PM as memory. In a multi-socket system, DRAM can only act as a cache for the PM DIMMs on the same socket [18]. The primary limitation with using PM in Memory-mode is that the available DRAM capacity is unusable by the operating system and thus applications as well.

C. Memory Caching and Tiering

The classical caching problem when used with memory hierarchies in computer systems is distinct from the dynamic memory tiering problem. With caching, every item needs to be fetched from the higher-performing tier (i.e., DRAM) before accessing it. With tiering, in addition to the high performance (DRAM) memory tier, there's a second (lower-performing) memory tier that is also directly accessible. Due to the small performance gap between the high-performing and the lower-performing tiers, items can be directly fetched from the lower-performing tier without significant performance loss. Thus, the core problem to address here is placing the right data in the right memory tier, online.

Caching-aware applications (e.g., compilers) can organize prefetching and increase memory access efficiency during execution. In the future, if tiers of memory get individually exposed to applications, it is conceivable that applications can achieve prefetching of data from PM to DRAM via OS hints. MULTI-CLOCK provides a currently usable method where the kernel can automatically identify the hot items and can serve them from the higher memory tier. This technique is entirely oblivious to applications. Furthermore, dynamic migration implemented in systems such as MULTI-CLOCK is complementary to prefetching-based techniques and can also

be effective in systems where prefetching is not feasible or accurate.

D. Existing Tiered Memory Systems and Their Limitations

Table I shows the comparison of the existing and MULTI-CLOCK tiering system. A straightforward way to tier is static tiering, whereby a memory page, once mapped to a tier, may not get reassigned to a different tier during its lifetime. However, this is inefficient; when an application wins the race to allocate memory from a higher tier, and such space is exhausted, future allocations will be downgraded to use lower tiers during their entire lifetime, regardless of how the *importance* of the contained data changes over time.

[Software Page Fault Based Page Access Tracking.] Thermostat [19] focused on tracking huge pages by poisoning the page table entry (PTE) and triggering a software page fault, and migrating cold pages to the lower memory tier. AutoNUMA-tiering [20] and AutoTiering [12] are based on AutoNUMA [21]. Similar to Thermostat, these solutions use a software page fault technique called hint page fault to track the page access and use recency to identify hot pages for promotion. Although the software page fault techniques can provide high accuracy in page access tracking, it is costly to track all the pages as every page fault has to be handled before accessing the page. Moreover, these techniques also require additional memory to store each page’s individual scan time on which its page hotness classification depends. AutoTiering designs a conservative approach (AutoTiering-CPM) to migrate pages to the best NUMA node. In addition, AutoTiering maintains an n-bit vector for each page to determine the page coldness and designs a progressive approach (AutoTiering-OPM) to demote cold pages to lower tier. We could not evaluate Thermostat as its source code was not available. We evaluate both AutoTiering-CPM and AutoTiering-OPM to compare the performance with MULTI-CLOCK. AutoTiering-CPM is designed using AutoNUMA-tiering, and thus we did not explicitly compare with AutoNUMA-tiering.

[Reference Bit Based Page Access Tracking.] Nimble [11] focuses on transparent huge page (THP) migration, enables multi-threaded concurrent migration, and two-sided page exchange to improve the overall page migration performance. However, Nimble uses the existing page profiling technique of the Linux kernel to exchange the top most recently accessed pages in the lower tier with the least recently accessed pages in the upper tier. Nimble is evaluated on an emulator, and applications need to run through Nimble’s *launcher* to utilize its page migration techniques. As Nimble mainly focuses on the optimization of the overall page migration process, we separated its hot/cold page identification technique and implemented a single threaded Nimble page selection mechanism in a real system for the singular purpose of comparing against MULTI-CLOCK’s page selection mechanism. MULTI-CLOCK itself is implemented as a built-in kernel feature, and hence, applications do not require to follow any purpose-built launcher mechanism for using MULTI-CLOCK.

AMP [22] proposes a tiered memory system that focuses on page selection mechanisms based on the popular cache replacement algorithms, including least-recently-used (LRU), least-frequently used (LFU), and random selection. AMP is designed, implemented, and evaluated using an emulator. AMP uses one node, only for DRAM allocations, and the other node only for PM allocations, which is unrealistic in a two socket NUMA machine wherein each node typically has its own DRAM, PM, and CPUs [7]. Moreover, AMP is implemented on Linux kernel version 4.15, which does not support the required KMEM DAX driver (available from kernel v5.1) to use PM as the main memory in a tiered system. Furthermore, the core design principle of AMP requires it to scan and profile all the memory pages from both DRAM and PM tier, which is impractical in the kernel on a real system as the number of in-memory pages can grow to hundreds of millions for the workloads we evaluated. Hence, for multiple practical reasons, we could not deploy AMP on a real system for evaluation.

Identifying the hot/cold data in virtual memory management may cause a high overhead. For low overhead, efficient tracking, the Linux kernel implements CLOCK, which is the approximation of the popular LRU cache replacement policy. As tracking every in-memory page access is not feasible, LFU is considered impractical for general virtual memory management. The CLOCK algorithm does not consider the frequency of the access. In a tiered memory system, as we have shown in Section II-A, it is important to capture both recency and frequency for hot/cold page identification. Hence, in this paper, we try to solve the following two novel research questions for tiered memory systems:

- *RQ1: How to identify hot pages for promotion based on recency and frequency?*
- *RQ2: How to design a simple and low overhead yet efficient system in the kernel?*

III. MULTI-CLOCK

A fundamental problem with static tiering is the mismatch of page access performance requirements with tier performance capabilities. Dynamic memory tiering mechanisms address this problem with a solution that dynamically migrates important pages to higher tiers and less important pages to lower tiers. The principal hypothesis of designing MULTI-CLOCK is that the pages that are recently accessed more than once are more likely to be accessed in the *near* future. MULTI-CLOCK determines the relative importance of pages within and across tiers by running a modified version of Linux’s Page Frame Reclamation Algorithm (PFRA) (which is based on the CLOCK algorithm) to each memory tier separately.

MULTI-CLOCK is implemented based on the well-known CLOCK because of its low overhead and effectiveness. However, MULTI-CLOCK does not use CLOCK exactly as it is. The CLOCK algorithm approximates LRU by checking for references when scanning the list of pages and moving any referenced page to the head of the list. MULTI-CLOCK uses a new approach to identify important pages in the lower tier. In addition to the active and inactive lists, MULTI-CLOCK

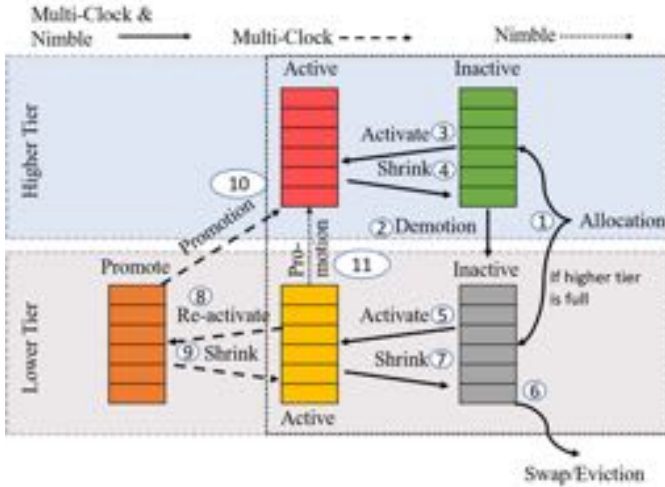


Fig. 3. MULTI-CLOCK architecture. The new data structures that we add to each tier and the interaction between these data structures. The arrows show the movement of pages across different page lists. The solid arrow represents both MULTI-CLOCK and Nimble, the dashed arrow is for MULTI-CLOCK only, and the dotted arrow is for Nimble only. The label numbers do not represent any particular order of the operations.

introduces a new *promote* list to select the candidate pages for promotion. MULTI-CLOCK completely reworks how pages are moved across the lists in both tiers, thus introducing a new lifecycle of pages.

New pages are allocated from the DRAM. Once the DRAM is full, pages are allocated from the PM tier. Every page in the system is arranged in one of its tier’s three lists according to their degree of hotness/coldness of accesses. The DRAM tier in the system does not use a promote list since there is no higher-performing tier to migrate pages to. With these changes, MULTI-CLOCK is able to capture both recency and frequency. Hence, although MULTI-CLOCK is based on the CLOCK algorithm, it is different from CLOCK and has significant algorithmic contributions and unique implementation challenges that we elaborate next in this section and in Section IV.

A. Life Cycle of a Page

Figure 3 depicts the overall arrangement of lists in the two tiers on the system and the possible movement of pages within and across these tiers for both MULTI-CLOCK and Nimble. Every list is scanned at various points in time to make decisions regarding migrations. In MULTI-CLOCK, a recently allocated page starts in the inactive list as shown in the Figure 3(1). The inactive list of a higher-performing tier maintains candidate pages for *demotion*, i.e., migration to a lower-performing tier. A page is said to be referenced if any type of access (i.e., read or write) occurs to the page. Both inactive and active lists make a differentiation between pages that were referenced and those that were not referenced since the last scan.

During a scan, if a page has been marked referenced since the previous scan is encountered, it is then marked as not

referenced and moved to the head of the list. On the other hand, if the page was not referenced, it is moved according to which list it belongs to: (a) if it belongs to the active list, it is moved to the inactive list as shown in Figure 3 (4) and (7), and (b) if it belongs to the inactive list is then migrated to its lower tier, and if none exists, evicted out of memory (Figure 3 (6)). This movement of pages out of a list is referred as the shrink of the source list. At the same time, when access occurs to a page in the inactive list and that page was marked as referenced, this page is *activated* by being moved to the active list’s head, where it starts out by being marked as not referenced (Figure 3 (3) and (5)). A similar process is followed when a page is *re-activated* and is moved from the active list to the promote list’s head (Figure 3(8)), where it becomes a candidate for *promotion* (i.e., migration to a higher-performing tier) as shown in Figure 3(10).

With this arrangement, the system is able to classify pages into three categories: *hot*, *warm*, and *cold*. Hot pages navigate the lists within a tier and eventually reach the promote list where they become candidate pages to migrate to the higher-performing tier. On the other extreme, cold pages remain in the inactive list where they become candidates for migration to a lower-performing tier when the tier experiences memory pressure. Thus, MULTI-CLOCK makes decisions on how to place each page within an appropriate tier and within an appropriate list according to their access frequency and recency.

The key difference in the architecture of MULTI-CLOCK and Nimble is shown in Figure 3. The life cycle in Nimble involves the page only residing in the dotted box on the right side of the Figure 3. Nimble does not have any promote list, and thus it cannot differentiate between pages accessed exactly once and those accessed more than once. Nimble selects a fixed number of the top pages in the lower tier’s active list to promote to the bottom of the higher tier’s active list as shown in Figure 3(11). The number of pages that get selected by MULTI-CLOCK is not fixed as it qualitatively chooses pages from the lower tier’s active list based on recent re-accesses to the pages.

One of the key challenges in designing MULTI-CLOCK is keeping track of accesses and updating the reference status of pages in a timely matter. This is addressed differently depending on the type of page access used by applications. Applications can access memory pages in two ways: *supervised*, using the operating system’s (OS) file system call interface, and *unsupervised*, by memory mapping pages into their address-space.

1) *Supervised Access*: This type of access is typically used for file-backed pages, and it gives the OS control at the moment of the access to perform the necessary book-keeping. When applications use supervised access to memory pages, the operating system is able to mark these pages referenced (for e.g., in Linux, via `mark_page_accessed()`) and, if necessary, to move between lists (activate or re-activate) before even processing the requested data access.

2) *Unsupervised Access*: Accesses to anonymous or file-backed memory that is directly mapped into the application’s virtual address space via `mmap` are more difficult to monitor.

This type of access is entirely unsupervised, and the OS is not able to mark such pages as referenced. To handle unsupervised access, MULTI-CLOCK relies on the page reference bit set by the CPU in the process’ page table entry. During each scan, as described earlier, before making any decision regarding a specific page, MULTI-CLOCK checks within every process’ page table that maps it for a set referenced bit. If a referenced bit is found set, MULTI-CLOCK updates the page status and takes care of the necessary movement between lists (i.e., mark as referenced, activate, or re-activate the page).

B. Promotion Mechanism

We design a new system daemon, `kpromoted`, that is woken up periodically to scan the lists, update them, and migrate any pages from the promote list to a higher tier due to recent unsupervised accesses. Every time `kpromoted` runs, it first selects the candidate pages for promotion and promotes all the pages it selected. Thus, once a page is selected for promotion, the page gets promoted to the DRAM in the same `kpromoted` run. As `kpromoted` promotes all the pages it selects, the number of promotions depends on the running application. If the application frequently accesses a large number of pages from the PM tier, the number of promotions will increase. On the other hand, if the application does not frequently access pages from the PM tier, `kpromoted` will promote fewer pages or no pages at all.

Implicitly, MULTI-CLOCK relies on the periodicity of `kpromoted` waking up to ensure that hot pages in lower tiers are migrated to higher tiers in a timely manner. The frequency of `kpromoted`’s execution defines the capacity of the system to react quickly to workload changes. If scheduled too frequently, excessive context switches to accommodate its execution could also affect application performance. Careful tuning of `kpromoted`’s execution schedule is necessary to ensure that applications benefit from the promotion mechanism in MULTI-CLOCK. In the prototype system we built, we chose the `kpromoted` execution schedule to be every 1 second as discussed in Section V-E and this worked fairly well for the workloads we evaluated the system with. It resulted in sufficient responsiveness in promoting hot pages without imposing high CPU overheads due to unnecessary scanning of every page in the LRU lists.

C. Demotion Mechanism

Demotion allows moving cold pages from a higher-performing tier to a lower-performing tier when these pages are no longer sufficiently important. MULTI-CLOCK’s design of this mechanism is based on the page eviction design in today’s virtual memory systems. To avoid running out of memory on a given tier, a tier is marked under memory pressure proactively when it reaches specific watermark levels. These levels are calculated by the system according to the amount of memory in the tier vs. the total amount of memory in the system.

If any tier is marked as being under memory pressure, each list is scanned with the objective of freeing up memory. Any page in the promote list is first attempted to be migrated

to a higher-performing tier, and if that is not possible — for instance, the page is locked — then it is moved to the active list. If the higher-performing tier is also under memory pressure, promotions from the lower tier result in immediate page demotions from the higher tier. Next, if the ratio of pages in the active list with respect to the inactive list exceeds a tunable threshold (inherited from PFRA and typically $\sqrt{10 * n} : 1$, where n is the amount of memory in GB available in the tier), pages not marked as referenced in the active list are moved to the inactive list. Finally, the inactive list is scanned in search of pages not marked as referenced to be migrated to a lower tier. Migration may not be possible, specifically because the candidate pages belong to the lowest tier in the system. In this case, these pages are written back to block storage (i.e., file-backed pages to file system and anonymous pages to the swap area if available) before triggering the out-of-memory (OOM) killer as the last option.

IV. IMPLEMENTATION

The existing Linux mechanism to describe physical memory relies on the definition of nodes. In NUMA architectures, each bank of memory is represented by a single NUMA node. On the other hand, for UMA architectures, Linux uses a single NUMA node to represent all physical memory in the system. The data structure used to represent nodes is called `pglist_data`. Each node is then divided into memory ranges called *memory zones*, and Linux uses the data structure `zone` to represent them in memory. Zones are of different types, and each type is suitable for a different usage (i.e., `ZONE_DMA` gathers physical addresses that can be accessed by legacy hardware through DMA). We implemented a prototype of MULTI-CLOCK for NUMA architectures for Linux kernel v5.3.1. Our prototype evaluates a hybrid two-tiered memory system: one tier of DRAM and another of persistent memory. In comparison with Nimble, which requires an additional launcher to run any workload on the kernel, our implemented prototype of MULTI-CLOCK can directly run any workload without any additional configuration setup or prior knowledge. We used the Intel Optane DC Persistent Memory on a real platform as the persistent memory tier (discussed in Section V-A). Upon creating a new namespace in `devdax` mode using the `ndctl` tool [23], we can hot-plug the namespace as system memory with the `DAX-KMEM` driver. `DAX-KMEM` driver is available in the kernel from v5.1 and onwards. The `DAX-KMEM` driver separates newly added PM from the DRAM by hot-plugging PM as a new node. We modified the `DAX-KMEM` driver to tag the newly hot-plugged node as a PM node, so that MULTI-CLOCK can recognize it by adding a new flag in the `pglist_data` structure. Although PM is hot-plugged as a new node, this node id is different from the physical node of the PM, i.e., the socket where it is physically installed. We define all the DRAM nodes as the DRAM tier and all the PM nodes in the system as the PM tier.

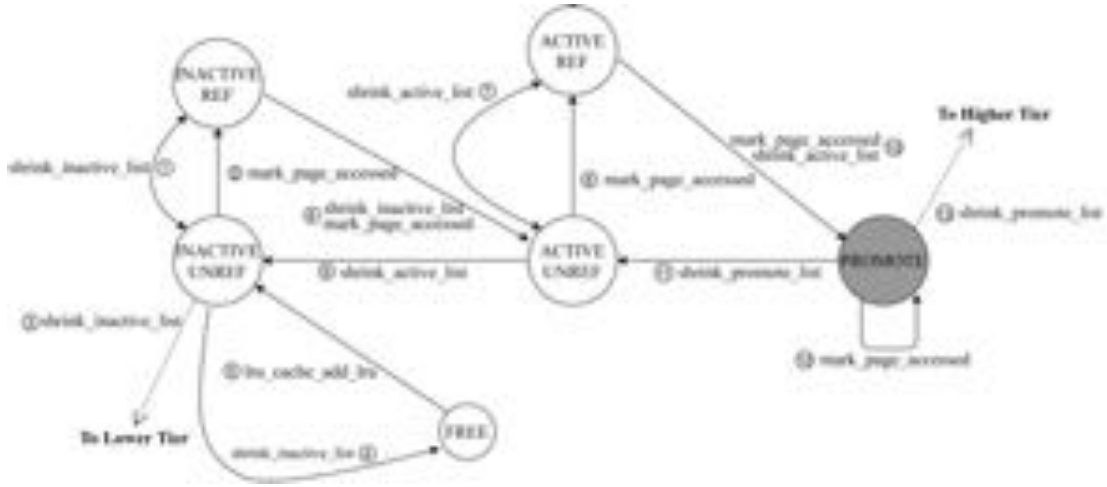


Fig. 4. **Page state diagram depicting the Linux implementation of MULTI-CLOCK** Each vertex represents a page state; white vertices are original PFRA page states while the gray vertex is a new page state introduced by MULTI-CLOCK. Solid edges represent Linux procedures that change page state; dashed edges represent page migration to a different tier. Counterparts to shrink_list methods are implicit on page allocations that cause lists to expand. The numeric edge numbers do not represent any particular order of operations.

The main design principle of MULTI-CLOCK is to migrate cold data from the DRAM tier to the PM tier and move hot data from the PM tier to the DRAM tier. We rely on the existing Linux migration mechanisms already in place for the hot-plug/hot-remove of memory. Linux’s page migration mechanism (`migrate_pages()`) is in charge of allocating new memory pages given an allocation routine, copying the memory contents from origin pages to the newly allocated destination pages, and fixing any memory mappings that refer to the migrated pages.

Originally, each memory node maintains its own set of LRU lists: anonymous inactive, anonymous active, file inactive, file active, and unevictable. We added two lists: *anonymous promote* and *file promote*. Unevictable pages belong to the unevictable list and are pages in the system that are locked into memory (typically via `mlock()`) and cannot be evicted nor migrated. Every evictable page in the system, depending on being file-backed or anonymous, will belong to one set of LRU lists (anonymous lists or file lists), and it will traverse these by transitioning through different states as depicted in Figure 4. We also extended the `struct page` flags which maintain the status of a page during its existence to add a new flag: `PagePromote`. This new flag is used by the OS to mark that the page in question, which is to be added to the LRU lists, belongs to the promote list. The memory overhead of these modifications is negligible since we reused the list pointer on the `struct page` to index the pages in the promote lists; we also reused the space allocated for the page flags to maintain the newly defined flag.

We implemented the system daemon discussed in Section III-B as a new kernel thread, `kpromoted`, which is woken up periodically to execute the migration of any pages sitting in the promote list to a higher tier. This thread’s design follows those of PFRA for the `kswaped` eviction daemon: one kernel thread per NUMA node. This design aims to avoid lock

Source File	New Lines	Modified Lines
drivers/base/node.c	4	1
drivers/dax/kmem.c	10	0
include/linux/gfp.h	7	1
include/linux/mm.h	6	0
include/linux/mm_inline.h	8	1
include/linux/mmzone.h	52	1
include/linux/nodemask.h	6	1
include/linux/page-flags.h	19	6
include/trace/events/mmflags.h	7	1
mm/Kconfig	3	0
mm/memcontrol.c	8	2
mm/migrate.c	1	0
mm/page_alloc.c	43	2
mm/swap.c	59	6
mm/vmscan.c	364	7
mm/vmstat.c	16	0

TABLE II
Linux source code modifications measured as number of lines modified.

contention on critical per-node data structures.

Our implementation of the MULTI-CLOCK algorithm is encapsulated mostly within `mm/vmscan.c` and `mm/swap.c`. Table II presents how much new code was added for MULTI-CLOCK and which files were modified in the Linux’s source code. In total, MULTI-CLOCK inserted 673 new lines and modified 30 existing lines of code. We extended `mark_page_accessed()` to check for pages that are already referenced and marked as active and are being referenced again to mark such pages with the `PagePromote` flag and to move them from their corresponding active list to the promote list (see transition 10 in Figure 4). We created a new `shrink_promote_list()` method that complements the existing `shrink_active_list()` and `shrink_inactive_list()` methods to handle movements of pages out of the promote list. Migrations to the upper tier are handled via `shrink_promote_list()` and

migration to the lower tier (or evictions) are handled via `shrink_inactive_list()`. Both methods result in a physical frame in the tier being freed after the successful migration of its contents.

Figure 4 depicts all the states and transitions of the pages. New pages start with the *inactive unreferenced* state. Depending on whether the page was accessed since the last scan or not, it can move to the *inactive referenced* state via transition (1), and (2), can get demoted to the lower tier via (3), or can be freed via (4). Pages not in LRU can also get added to *inactive unreferenced* state via (5). Pages in the *inactive referenced* state can either move to *inactive unreferenced* via (1) or move to the active list via (6). Pages in the active list with the *active unreferenced* state move to the *active referenced* state using (7) or (8) if they get accessed. Furthermore, if the page is not accessed for a long time, the page state changes to *inactive unreferenced* via (9). From *active referenced* state, a page moves to the promote list if it gets accessed via transition (10). If pages in the promotion list do not get accessed, they move to the *active unreferenced* state again via (11). If they get accessed in this state, then pages remain in the same state, as shown by (12). Lastly, `kpromoted` uses (13) to promote all the pages found in this state.

V. EVALUATION

In this section, we evaluate the performance of our MULTI-CLOCK implementation. The goal of our evaluation is to determine if, when, and how the MULTI-CLOCK is able to improve the performance of application workloads. We evaluate using diverse workloads such as high memory-consuming graph applications and key-value stores. We compare MULTI-CLOCK performance with static tiering, Nimble, AutoTiering, and Memory-mode. As Nimble uses Linux’s CLOCK (an approximation of LRU) based default page profiling mechanism, we do not compare MULTI-CLOCK again with CLOCK or LRU. We also avoid comparing MULTI-CLOCK with the *Least Frequently Used* (LFU) policy as it requires tracking every memory access, which is impractical. Additional reasons for not comparing MULTI-CLOCK with other memory tiering techniques such as AMP [22], Thermostat [19], and AutoNUMA-Tiering [20], are discussed in Section II-D. Finally, we conduct an in-depth sensitivity analysis to better understand the impact of each component of MULTI-CLOCK.

A. Experimental Platform

We used a dual-socket Intel Xeon Gold 5218 Processor with 16 cores per socket for evaluating and comparing the performance of static tiering, Nimble, AutoTiering-CPM (AT-CPM), and AutoTiering-OPM (AT-OPM) with MULTI-CLOCK. This machine has 12 DDR4 (2666 MT/s) DIMMs of 16GB in capacity each and 4 Intel Optane DC Persistent Memory (DCPM) of 128GB in capacity each. In total, the available memory space is 192GB DRAM and 512GB persistent memory. We used another platform to compare the performance of static tiering, Memory-mode, and MULTI-CLOCK. This machine runs a dual-socket Intel Xeon Processor with 24 cores

per socket. The system is equipped with 12 DDR4s (2666 MT/s), each 32GB in capacity and another 12 Intel Optane DCPM with 128GB capacity per DIMM. Hence, the total DRAM capacity is 376GB, and PM capacity is 1.5TB. The only reason for using two separate machines is to expedite the evaluation process.

B. Workloads

We evaluate MULTI-CLOCK using diverse workloads. Here, we discuss our results using six different workloads from Yahoo! Cloud Serving Benchmark (YCSB) [13] and six workloads from the GAP Benchmark Suite (GAPBS) [14]. YCSB workloads are divided into two phases: a *load* phase and an *execution* phase. The load phase is in charge of populating the back-end key-value store with the required number of records. On the other hand, the execution phase carries out diverse types of operations over the back-end. These workloads are named Workload A, B, C, D, E, and F. Workload A is a mix of 50% reads, and 50% writes. Workload B is 95% reads, and only 5% writes. Workload C is 100% read. None of these workloads inserts new records except workload D, where new items are added and read. Workload E issues short ranges queries on the records. And in workload F, a record is read, modified, and then written back. We also created a new workload W, which issues 100% writes. For our evaluation, we used Memcached [3], an in-memory cache service that uses a large amount of main memory to maintain its data, as the key-value store back-end of YCSB. One thing to note is that YCSB’s workload E makes use of SCAN operations that may or may not be implemented by the different back-end key-value stores. Memcached does not implement SCAN operations, making workload E non-operational. Further, the load phase is the same for all workloads, and most workloads (all but D and E) do not change the number of records in the back-end. For all our experiments, we follow the prescribed execution sequence [24] for the YCSB workloads. Since workload D changes the number of records in the back-end, the order of execution is arranged in the following manner: Load Phase, Workload A, Workload B, Workload C, Workload F, Workload W, and Workload D. We report the performance of the six Workloads, excluding the data load phase.

GAPBS is a framework for graph analytics capable of running a wide variety of graph processing algorithms. It has six workloads: Breadth-First Search (BFS), Single-Source Shortest Paths (SSSP), PageRank (PR), Connected Components (CC), Betweenness Centrality (BC), and Triangle Counting (TC). For each of the six workloads, GAP first loads the graph in memory and then executes multiple trials of the workload. We report the average execution time taken per trial for the workloads. During the execution phase, the actual algorithm is executed over the already memory-resident graph representation of the data.

C. Evaluation Result

To evaluate the overall performance of MULTI-CLOCK, we first compare MULTI-CLOCK against systems using PM in

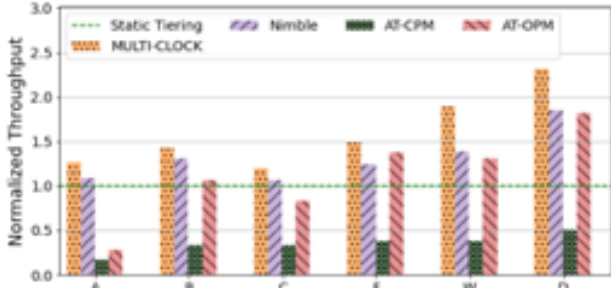


Fig. 5. MULTI-CLOCK, Nimble, AutoTiering-CPM(AT-CPM), and AutoTiering-CPM(AT-OPM) throughput comparison for YCSB workloads. Y axis presents the throughput normalized to static tiering (higher is better).

static tiering, Nimble, AutoTiering-CPM, and AutoTiering-OPM. Then we compare the performance of MULTI-CLOCK with Memory-mode. We configure workloads for all the systems such that their memory footprints are larger than the DRAM size and consume enough persistent memory. For both MULTI-CLOCK and Nimble, we set the number of page scan to 1024. The scanning interval of MULTI-CLOCK and Nimble is set to one second as discussed in Section V-E.

1) *Comparison With Tiered Memory Systems:* We compare the performance of static tiering, MULTI-CLOCK, Nimble, AutoTiering-CPM (AT-CPM), and AutoTiering-OPM (AT-OPM) using YCSB and GAPBS workloads. Figure 5 shows the performance for YCSB workloads. In Figure 5, the Y-axis presents the throughput (operations per second) normalized to static tiering; all the workloads are on the X-axis. MULTI-CLOCK outperforms static tiering, Nimble, AT-CPM, and AT-OPM for all the workloads.

For the YCSB workloads, MULTI-CLOCK outperforms static tiering by 20-132%. MULTI-CLOCK achieves the maximum throughput gain in Workload D as this workload inserts new records and modifies the most recent records multiple times. As MULTI-CLOCK selects the pages that are recently accessed multiple times for promotion, Workload D and other workloads with a similar property would get the most benefit from MULTI-CLOCK. In comparison with Nimble, MULTI-CLOCK achieves 9-36% better performance as MULTI-CLOCK promotes pages more selectively than Nimble. The selective promotion of MULTI-CLOCK helps to reduce the migration overhead incurred for promoting less qualified pages. When compared to AT-CPM, MULTI-CLOCK outperforms by 260-677%. Finally, MULTI-CLOCK achieved 10-352% better performance than AT-OPM. In comparison with MULTI-CLOCK, AT-CPM and AT-OPM perform worse due to costly software page fault-based page access tracking as well as the high overhead of tracking the page history bits for identifying cold pages.

Figure 6 presents the results of executing different GAPBS’s workloads normalized to static tiering. The Y-axis shows the normalized execution time; the X-axis presents all the workloads. As we can see, MULTI-CLOCK outperforms static tiering by 4-68% for the GAPBS workloads. When compared

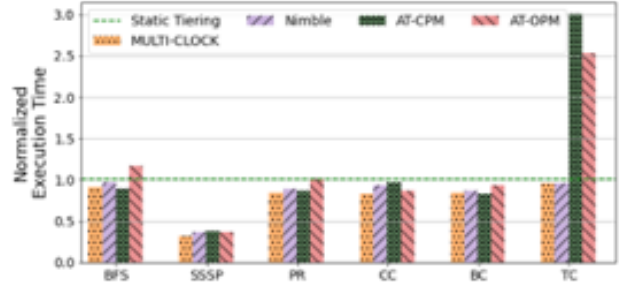


Fig. 6. Performance comparison of GAPBS workloads. Y axis presents the normalized execution time to the static tiering (lower is better).

to Nimble, MULTI-CLOCK improved the execution time by 1-16%. In both comparisons, MULTI-CLOCK reduces the execution time of the SSSP workload the most. Similar to the YCSB workloads, in GAPBS, MULTI-CLOCK benefits from the better page selection mechanism for promotions.

In comparison with AT-CPM, MULTI-CLOCK reduces the execution time by 3-68% for SSP, PR, CC, and TC workloads. However, AT-CPM shows 3% and 1% better performance than MULTI-CLOCK for BFS and BC workloads. As AT-CPM tries to find the best location of the pages, its performance thus highly depends on the initial placement of the pages. If pages are already placed in the best locations, AT-CPM needs to migrate fewer pages. We think the slight performance gain for BFS and BC workloads might be due to this reason. On the other hand, MULTI-CLOCK shows better performance than AT-OPM by 4-62%. Compared to AT-CPM, AT-OPM induces additional overhead of identifying cold pages and page demotions, which is the reason for the observed performance.

From Figure 5 and Figure 6 we observe that the MULTI-CLOCK achieved better performance gain for the YCSB workloads than the GAPBS’s workloads. The performance of the graph processing algorithms can depend on the locality of the data [25]. We assume that the GAPBS workloads first allocate memory that would be accessed the most as graph processing workloads are known to exhibit substantial locality [26]. As static tiering, MULTI-CLOCK, Nimble, AT-CPM, and AT-OPM fill the DRAM first, DRAM contains most of the highly accessed pages. Hence, the performance of the MULTI-CLOCK, Nimble, AT-CPM, and AT-OPM is close to the static tiering for most of the GAPBS workloads. However, by selectively promoting the hot pages from PM to DRAM, MULTI-CLOCK achieves a slightly better performance on average than other tiering mechanisms across different workloads.

In Section II-A, we analyzed the workloads from various benchmarks to show the existence of DRAM-friendly and Tier-friendly pages. The goal of MULTI-CLOCK is to identify these pages and place the frequently accessed pages in the DRAM tier. Workloads with weak locality will not have such a division of pages and would not benefit from MULTI-CLOCK. On the other hand, workloads with strong locality will have many DRAM and Tier friendly pages and can reap benefits from the dynamic tiering capabilities of MULTI-CLOCK. Among the YCSB workloads, workload D inserts new data in PM (as

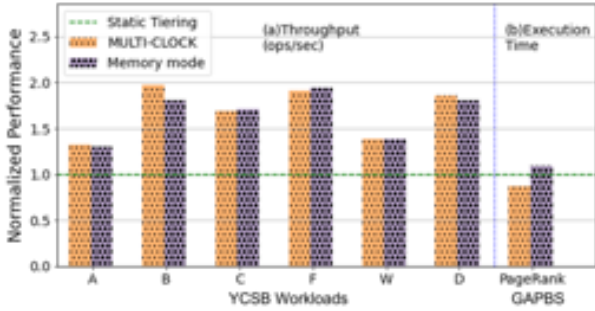


Fig. 7. **Performance comparison of MULTI-CLOCK with Memory-mode.** Y axis presents the normalized performance in (a) throughput (higher is better) and (b) execution time (lower is better).

DRAM is already full) and frequently accesses the recently inserted data, thereby exhibiting a stronger locality than the other workloads. In comparison with static tiering, MULTI-CLOCK obtains the greatest performance gain (132%) for this workload.

2) *Comparison With Memory-mode*:: Finally, we compare the performance of MULTI-CLOCK with Memory-mode. As Memory-mode uses all of the DRAM capacity for caching, to allow for a competitive comparison with MULTI-CLOCK, we set the workload size to be 4x of the available DRAM capacity.

In Figure 7, the Y-axis shows performance normalized to that of static tiering. Figure 7(a) shows the normalized throughput for the YCSB workloads and Figure 7(b) shows the normalized execution time for the GAPBS’s PageRank algorithm. For the YCSB workloads, MULTI-CLOCK outperforms Memory-mode by as much as 9% and operates within 2% of Memory-mode’s performance. For PageRank, MULTI-CLOCK outperforms Memory-mode by 21%. To improve application performance, Memory-mode uses all the available DRAM as cache, thus hiding the available DRAM capacity from the applications; it achieves as much as 2% better performance than MULTI-CLOCK. On the other hand, MULTI-CLOCK exposes all the available DRAM and PM capacity to the application and provides performance that is either better or very similar to Memory-mode.

D. Performance Analysis

To understand the reason behind MULTI-CLOCK’s better performance outcomes, we first analyze the number of pages promoted by MULTI-CLOCK and Nimble. Then we see how many of these promoted pages are getting re-accessed again from the DRAM tier. This discussion helps us understand MULTI-CLOCK in more detail.

1) *Number of page promotions*: In Figure 8, we report the number of pages being promoted across tiers for both MULTI-CLOCK and Nimble. In Figure 8, the Y-axis shows the average number of pages promoted in a time window. We chose the time window as twenty seconds. The X-axis represents the time window ID. As we can see from the figure, the average number of pages Nimble promotes is always 1024. This is because Nimble always selects a fixed number of pages for

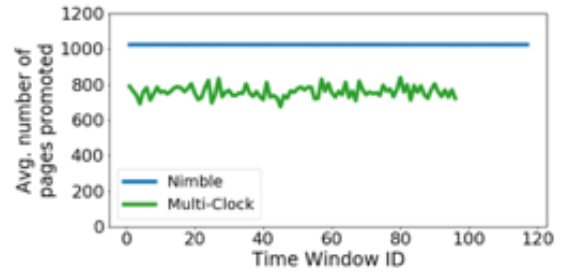


Fig. 8. **The average amount of pages promoted in each scan over time.** Y-axis is the average number of pages that are promoted in 20 seconds window. X-axis presents the time window IDs.

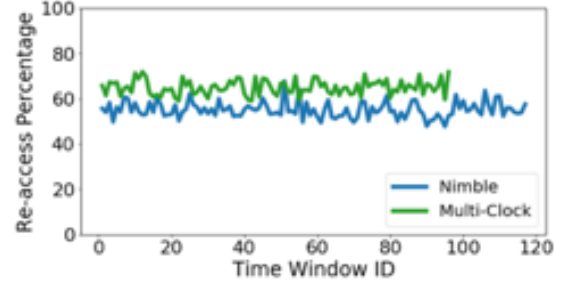


Fig. 9. **The average re-access percentage of the promoted pages in each scan.** Y-axis is the average number of promoted pages that got re-accessed. The average is calculated based on a time window of 20 seconds. X-axis presents the time window IDs.

promotion, and we used 1024 as the fixed value. On the other hand, MULTI-CLOCK promotes 758 pages on average per scan. Similar to Nimble, MULTI-CLOCK scans a maximum of 1024 pages, but unlike Nimble, MULTI-CLOCK selects the pages that have been recently accessed multiple times. If pages that do not get re-accessed again in the future get promoted to DRAM, then the overhead to promote such pages can hurt system performance.

2) *Percentage of Pages Re-accessed*: Now, we analyze the number of pages that have been promoted in the last scan, get re-referenced again from the DRAM. In Figure 9, the Y-axis shows the re-access percentage, which represents the average percentage of the recently promoted pages which have been re-accessed. The average percentage is calculated for 20 second time window. The time window IDs are shown on the X-axis. From Figure 9, we can see that pages promoted by MULTI-CLOCK have 15% higher re-access percentage than Nimble. In combination with Figure 8, we come to an interesting observation. Nimble promotes more pages than MULTI-CLOCK, but a lower percentage of the promoted pages are re-accessed again. This explains the improved performance results that we observed with YCSB and GAPBS workloads.

E. Scanning Interval Sensitivity

As described in Section IV, the $k_{promoted}$ daemon wakes up after a specific time interval. $k_{promoted}$ is responsible for moving pages from the inactive list to the active list, from the active list to the promote list, and from the promote list to the DRAM tier. Varying this time interval in MULTI-CLOCK is expected to affect the performance of the application. We set

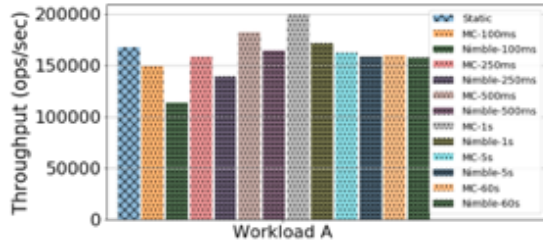


Fig. 10. **Throughput comparison of Static tiering, MULTI-CLOCK, and Nimble with different scan intervals for YCSB Workload A.** Y axis presents throughput (higher is better).

the time interval to 100ms, 250ms, 500ms, 1s, 5s, and 60s and run the workload A from YCSB with each of these MULTI-CLOCK versions. Nimble uses a similar daemon thread to promote pages periodically. Similar to MULTI-CLOCK, we also evaluated Nimble with different time intervals. From Figure 10 we see that overall MULTI-CLOCK performs better when compared to Nimble. For larger scan intervals above 5s, we do not observe much difference due to the lag in the reaction time. The one-second scan interval was found to be the best performing for various workloads, but in Figure 10, we only show the results for YCSB workload A as a representative. Hence, we chose the one-second scanning interval for all the other evaluations for MULTI-CLOCK and Nimble.

F. Overhead

Mainly the overhead of MULTI-CLOCK includes the overhead for promotion and demotion of the pages across different tiers. While running memory access intensive applications, the overhead depends on which tier the pages are being accessed from. First, if DRAM pages are heavily accessed, then there will be no overhead due to no migration being incurred. Second, if pages from the PM tier are heavily accessed, then to reduce access latency, MULTI-CLOCK will identify these pages and promote them to DRAM, incurring promotion overheads as well as demotion overheads if the DRAM is full. However, if the application is memory intensive, then the promoted pages would be accessed repeatedly from the DRAM tier, which can benefit the application due to DRAM’s lower access latency. Thus, for memory-intensive workloads, MULTI-CLOCK’s benefit will surpass the migration overhead.

VI. RELATED WORK

Emerging persistent memory technologies show promise in three distinct areas: non-volatility, very large capacity (as compared to DRAM), and performance suitable for direct load/store access by the CPU. Most studies on persistent memory, far too many to list here, focus on the non-volatility, using it to replace or extend block storage, implement persistent caches, or explore the persistent execution of processes that can survive power failures [27]–[31]. In contrast, our work focuses on the large capacity characteristic of persistent memory and the ability to directly read, write, and execute data residing in persistent memory.

There have been many studies that explore the use of different types of memory for the building of *hybrid memory systems*. Such systems make use of the different characteristics of the available memory types to combine them into a hybrid solution. Most hybrid memory systems do not establish any specific hierarchy between the different memory types as tiered memory systems do. As discussed in Section II-D, Thermostat [19], Nimble [11], AMP [22], AutoNUMA-Tiering [20], and AutoTiering [12] are the recent works on dynamic tiered memory system.

Yang [32] proposes a design to use persistent memory as a NUMA node efficiently. This tiered design is aware of both DRAM and PM nodes and handles promotion/demotion for anonymous pages only via NUMA balancing. In contrast, MULTI-CLOCK selects pages for promotion more carefully by scanning pages periodically and moving them across inactive, active, and newly added promote list depending on page access. Moreover, MULTI-CLOCK is capable of managing all types of pages, anonymous and file-backed pages, making MULTI-CLOCK a complete solution.

Qureshi et al. [33], Dhiman et al. [34], Ramos et al. [35], and Lee et al. [36] propose hybrid memory systems, where DRAM is used as buffer cache, PM is used as the DRAM’s extension, and a hardware-based solution is used to find best page replacement policy. In contrast to these works, we provide a page selection mechanism that can be used to improve the performance of a dynamic tiered memory system without any hardware modification, where DRAM and PM both co-exist as system main memory.

Many replacement algorithms have been studied in the past in the context of caching [37]–[43]. Our solution is orthogonal to these efforts and builds upon existing memory replacement mechanisms, and presents a modified page migration and replacement algorithm for tiered memory.

Liu *et al.* [44] provide object-level memory allocation and migration in hybrid memory systems. Data placement and migration at the object granularity requires modification of the existing application to use the new APIs. In contrast, MULTI-CLOCK operates seamlessly at the kernel level, and existing applications can be run as-is without any modification.

VII. DISCUSSION

MULTI-CLOCK relies on the page reference bit for classifying pages according to their frequency of accesses and characterizing the *importance* of a page. In the current version, MULTI-CLOCK does not differentiate between the data read and write. One possible improvement to this approach is to also include the dirtiness information for memory pages in a weighted formula to compute the *importance* of a page. By including this extra information, we could weigh the different types of accesses for a page (read or write) in the decision of page placement. This additional information becomes particularly relevant when the underlying memory hardware exhibits non-uniform latency for the different types of accesses. For instance, some PM devices, e.g., Intel Optane PM, are known to have asymmetric read and write latencies.

The scanning interval for MULTI-CLOCK is 1s as we discussed in Section V-E. We compared the performance of MULTI-CLOCK across multiple scanning intervals and chose the 1s scan interval. However, it could be valuable to dynamically adjust the scanning interval for `kpromoted` by analyzing the characteristics of the running workload. Additionally, it will also be interesting to see the performance of MULTI-CLOCK with varying DRAM and PM ratios.

VIII. CONCLUSIONS

Byte-addressable, high capacity memory such as PM opens up a new space for optimization of the memory system design and implementation. In this work we design and develop MULTI-CLOCK, a dynamic memory tiering system that is designed to ensure that the right data is in the right tier at the right time. Unlike some other recent approaches for tiered systems, MULTI-CLOCK uses both access recency and frequency to identify potential pages for migration without adding significant system overhead. We deployed MULTI-CLOCK in a real system by developing a prototype that runs CentOS 7 (Linux kernel 5.3.1) and evaluated our prototype using graph processing and key-value store workloads. Our results demonstrate that MULTI-CLOCK is able to significantly improve the performance of these workloads compared to the state-of-the-art techniques without compromising the amount of usable main memory made available to these workloads. MULTI-CLOCK sources can be downloaded at <https://doi.org/10.5281/zenodo.5790897>

ACKNOWLEDGMENTS

We would like to thank the reviewers of this paper for insightful feedback that helped improve the content and presentation of this paper substantially. This work was supported in part by NSF grants CCF-1718335, CNS-1956229, and CNS-2008324.

REFERENCES

- [1] “Apache Spark,” <http://spark.apache.org>, January 2012.
- [2] “Sap hana,” <http://hana.sap.com>, January 2020.
- [3] B. Fitzpatrick, “Distributed caching with memcached,” in *Linux Journal*, 2004.
- [4] Salvatore Sanfilippo and Pieter Noordhuis, “Redis,” <http://redis.io>, January 2020.
- [5] A. Makarov, V. Sverdlov, and S. Selberherr, “Modeling emerging non-volatile memories: Current trends and challenges,” in *Proceedings of the International Conference on Solid State Devices and Materials Science*, ser. SSDM ’12, April 2012.
- [6] S. Mittal, “Energy saving techniques for phase change memory (PCM),” in *arXiv:1309.3785*, 2013.
- [7] “Intel Optane DC Persistent Memory,” <https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel-Optane-DC-Persistent-Memory-Quick-Start-Guide.pdf>, July 2021.
- [8] I. Peng, M. Gokhale, and E. Green, “System evaluation of the intel optane byte-addressable nvm,” in *arXiv:1908.06503*, 2019.
- [9] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. Soh, Z. Wang, Y. Xu, S. Dullloor, J. Zhao, and S. Swanson, “Basic performance measurements of the intel optane dc persistent memory module,” in *arXiv:1903.05714*, 2019.
- [10] Y. Zhang and S. Swanson, “A study of application performance with non-volatile main memory,” in *Proceedings of the 2015 31st Symposium on Mass Storage Systems and Technologies*, ser. MSST ’15, June 2015.
- [11] Z. Yan, D. Lustig, D. Nellans, and A. Bhattacharjee, “Nimble page management for tiered memory systems,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’19, New York, NY, USA: Association for Computing Machinery, 2019, p. 331–345. [Online]. Available: <https://doi.org/10.1145/3297858.3304024>
- [12] J. Kim, W. Choe, and J. Ahn, “Exploring the design space of page management for multi-tiered memory systems,” in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 715–728. [Online]. Available: <https://www.usenix.org/conference/atc21/presentation/kim-jonghyeon>
- [13] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proceedings of the ACM symposium on Cloud computing*, ser. SoCC ’10, June 2010.
- [14] S. Beamer, K. Asanović, and D. Patterson, “The gap benchmark suite,” in *arXiv:1508.03619*, 2015.
- [15] OW2 Consortium, “RUBiS: Rice University Bidding System,” <http://rubis.ow2.org/>.
- [16] SPECpower_ssj2008, “<https://www.spec.org/power/>”
- [17] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, “The DaCapo benchmarks: Java benchmarking development and analysis,” in *OOPSLA ’06: Proceedings of the 21st annual ACM SIGPLAN conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York, NY, USA: ACM Press, Oct. 2006, pp. 169–190.
- [18] G. Gill, R. Dathathri, L. Hoang, R. Peri, and K. Pingali, “Single machine graph analytics on massive datasets using intel optane dc persistent memory,” in *arXiv:1904.07162*, 2019.
- [19] N. Agarwal and T. F. Wenisch, “Thermostat: Application-transparent page management for two-tiered main memory,” *SIGPLAN Not.*, vol. 52, no. 4, p. 631–644, Apr. 2017. [Online]. Available: <https://doi.org/10.1145/3093336.3037706>
- [20] H. Ying, “tiering-0.6,” <https://git.kernel.org/pub/scm/linux/kernel/git/vishal/tiering.git/commit/?h=tiering-0.6>, 2020.
- [21] R. van Riel and V. Chegu, “Automatic numa balancing, 2014,” https://www.redhat.com/files/summit/2014/summit2014_riel_chegu_w_0340_automatic_numa_balancing.pdf, 2014.
- [22] T. Heo, Y. Wang, W. Cui, J. Huh, and L. Zhang, “Adaptive page migration policy with huge pages in tiered memory systems,” *IEEE Transactions on Computers*, pp. 1–1, 2020.
- [23] , “Utility library for managing the libnvdimm,” <https://github.com/pmem/ndctl>, January 2020.
- [24] B. Cooper and N. Bailey, “Ycsb core workloads,” <https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads>, October 2010.
- [25] Y.-Y. Jo, J. Hong, M.-H. Jang, J.-G. Bang, and S.-W. Kim, “Data locality in graph engines: Implications and preliminary experimental results,” in *Proceedings of the 25th ACM International Conference on Information and Knowledge Management*, ser. CIKM ’16, New York, NY, USA: Association for Computing Machinery, 2016, p. 1885–1888. [Online]. Available: <https://doi.org/10.1145/2983323.2983865>
- [26] S. Beamer, K. Asanovic, and D. Patterson, “Locality exists in graph processing: Workload characterization on an ivy bridge server,” in *2015 IEEE International Symposium on Workload Characterization*, 2015, pp. 56–65.
- [27] J. Coburn, A. Caulfield, A. Akel, L. Grupp, R. Gupta, R. Jhala, and S. Swanson, “NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories,” in *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011.
- [28] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, “Better I/O through byte-addressable, persistent memory,” in *Proceedings of the ACM Symposium on Operating Systems Principles*, ser. SOSP ’09, 2009.
- [29] S. R. Dullloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, “System software for persistent memory,” in *Proceedings of the European Conference on Computer Systems*, ser. EuroSys ’14, 2014.
- [30] H. Kim, S. Seshadri, C. L. Dickey, and L. Chiu, “Evaluating phase change memory for enterprise storage systems: A study of caching and tiering approaches,” in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, ser. FAST ’14, February 2014.

- [31] J. Yang, Q. Wei, C. Chen, C. Wang, and K. L. Yong, "NV-Tree: Reducing consistency cost for NVM-based single level systems," in *Proceedings of the USENIX Conference on File and Storage Technologies*, ser. FAST '15, February 2015.
- [32] Y. Shi, "Another approach to use pmem as numa node," <https://lore.kernel.org/linux-mm/1553316275-21985-1-git-send-email-yang.shi@linux.alibaba.com/>, March 2019.
- [33] M. K. Qureshi, V. Srinivasan, and J. A. Rivers, "Scalable high-performance main memory system using phase-change memory technology," in *Proceedings of the International Symposium on Computer Architecture*, ser. ISCA '09, 2009.
- [34] G. Dhiman, R. Ayoub, and T. Rosing, "PDRAM: a hybrid pram and dram main memory system," in *Proceedings of the Annual Design Automation Conference*, 2009.
- [35] L. Ramos, E. Gorbato, and R. Bianchini, "Page placement in hybrid memory systems," in *Proceedings of the 25th International Conference on Supercomputing*, ser. ICS '11, 2011.
- [36] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian, and O. Mutlu, "Tiered-latency dram: A low latency and low cost dram architecture," in *Proceedings of the 2013 IEEE 19th International Symposium on High Performance Computer Architecture*, ser. HPCA '13, 2013.
- [37] H. Kim and S. Ahn, "BPLRU: A Buffer Management Scheme for Improving Random Writes in Flash Storage," in *USENIX File and Storage Systems (FAST)*, 2008.
- [38] N. Megiddo and D. S. Modha, "Outperforming lru with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, 2004.
- [39] S.-y. Park, D. Jung, J.-u. Kang, J.-s. Kim, and J. Lee, "Cfrru: a replacement algorithm for flash memory," in *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, October 2006.
- [40] T. M. Wong and J. Wilkes, "My cache or yours? making storage more exclusive," in *Proceedings of the USENIX Annual Technical Conference*, ser. USENIX ATC '02, June 2002.
- [41] Y. Zhou, J. F. Philbin, and K. Li, "The multi-queue replacement algorithm for second level buffer caches," in *Proceedings of the 2001 USENIX Annual Technical Conference*, ser. USENIX ATC '01, June 2001.
- [42] S. Jiang, F. Chen, and X. Zhang, "Clock-pro: An effective improvement of the clock replacement," in *Proc. of the USENIX Annual Technical Conference*, 2005.
- [43] S. Bansal and D. Modha, "Car: Clock with adaptive replacement," in *Proc. of the Third USENIX Conference on File and Storage Technologies*, 2004.
- [44] H. Liu, R. Liu, X. Liao, H. Jin, B. He, and Y. Zhang, "Object-level memory allocation and migration in hybrid memory systems," *IEEE Transactions on Computers*, vol. 69, no. 9, pp. 1401–1413, 2020.

Wear Leveling in SSDs Considered Harmful

Ziyang Jiao
Syracuse University
zjiao04@syr.edu

Janki Bhimani
Florida International University
jbbhimani@fiu.edu

Bryan S. Kim
Syracuse University
bkim01@syr.edu

ABSTRACT

We argue that wear leveling in SSDs does more harm than good under modern settings where the endurance limit is in the hundreds. To support this claim, we evaluate existing wear leveling techniques and show that they exhibit anomalous behaviors and produce a high write amplification. These findings are consistent with a recent large-scale field study on the operational characteristics of SSDs. We discuss the option of forgoing wear leveling and instead adopting capacity variance in SSDs, and show that the capacity variance extends the lifetime of the SSD by up to $2.94\times$.

CCS CONCEPTS

• Information systems → Flash memory; Information lifecycle management.

KEYWORDS

SSD, wear leveling, endurance, lifetime, write amplification, capacity variance

ACM Reference Format:

Ziyang Jiao, Janki Bhimani, and Bryan S. Kim. 2022. Wear Leveling in SSDs Considered Harmful. In *14th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '22)*, June 27–28, 2022, Virtual Event, USA. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/3538643.3539750>

1 INTRODUCTION

Wear leveling (WL) in solid-state drives (SSDs) seeks to equalize the amount of wear so that no cells prematurely fail prior to the end of the SSD's lifetime [3, 6–8, 10]. While there are different approaches to implementing WL (from static [3, 6, 10] to dynamic [7, 8]), the underlying goal is to use younger blocks with fewer erases more than the older

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HotStorage '22, June 27–28, 2022, Virtual Event, USA

© 2022 Association for Computing Machinery.

ACM ISBN 978-1-4503-9399-7/22/06...\$15.00

<https://doi.org/10.1145/3538643.3539750>

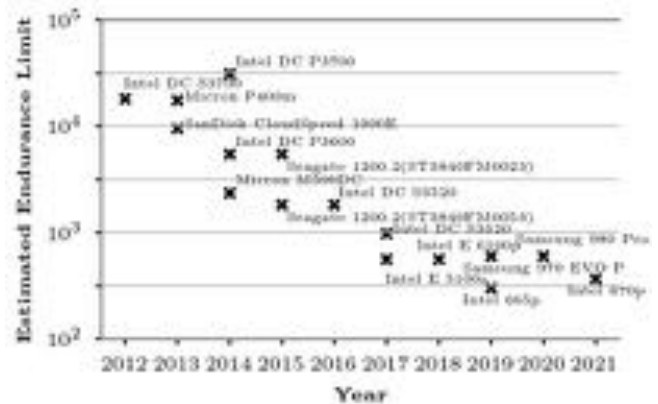


Figure 1: The estimated endurance limit of various SSDs in the past years. We estimate the endurance limit by dividing the SSD's TBW (terabytes written: the total amount of writes the SSD manufacturer guarantees) by the logical capacity. This estimation is consistent with recent works [21, 22].

blocks. Static wear leveling techniques [3, 6, 10], in particular, proactively relocate data within an SSD, thereby incurring additional write amplification for the sake of equalizing the number of erases. Dynamic wear leveling [7, 8], on the other hand, combines WL with other SSD-internal tasks such as garbage collection, reducing the efficiency of victim block selection. In other words, WL techniques incur additional wear-out to increase the overall lifetime of the SSD.

Ideally, the wear leveling algorithm would minimize its overhead while maximizing its effectiveness. However, a recent large-scale field study on millions of SSDs reveals that the WL techniques in modern SSDs present limited effectiveness and are far from perfect [23]. This study shows that some WL algorithms are unable to achieve their intended goal as some of the blocks wear out $6\times$ faster than the average. Furthermore, some SSDs exhibit a median write amplification factor (WAF) of around 100, although the cause of this cannot be definitive. With the endurance limit of flash memory steadily decreasing, as shown in Figure 1, it will become increasingly challenging to design an effective (equal wear) yet efficient (low write amplification) wear leveler.

To understand the underlying reasons for the ineffectiveness of WL in SSDs, we evaluate three representative WL techniques [3, 6, 7] that have been compared against a wide variety of other WLs [5, 10, 15, 24, 28, 30, 33]. Our experiments find that WL algorithms produce a counter-productive

Table 1: Representative wear leveling algorithms.

Name	Type	Parameters	Principle	Comparisons
DP [3]	Static	Fixed, a predefined threshold (TH)	Hot-cold swapping	HC [15], 2L [33], EP [30], OBP [10]
PWL [6]	Static	Adaptive, a initial threshold (THR_{init})	Cold-data migration	BET [5] and Rejuvenator [24]
DAGC [7]	Dynamic	Adaptive, no external parameters	Adjust GC victim	DTGC [28]

result where the erase counts diverge, increasing the spread rather than reducing it. This happens when the WL attempts to move data that it incorrectly perceives to be cold into old blocks. In addition, we observe that WL-induced WAF can reach as high as 11.49 where WL’s attempt to achieve a tight distribution of erase count comes at the cost of a high WAF.

Instead of designing a new wear leveling algorithm that patches these issues, we fundamentally ask if wear leveling is worth the trouble. Wear leveling exists to maintain the fixed capacity abstraction, when in reality, the underlying media for SSDs fail partially [26]. Instead, we explore and quantify the benefits of capacity variance in an SSD that gracefully reduces its capacity as flash memories become bad [18]. Our experimental results show that capacity variance allows up to 84% more writes to the SSD with wear leveling, and up to 2.94× more writes *without* WL.

2 WEAR LEVELING: BOON OR BANE?

Motivated to reproduce the results from a recent large-scale study [23], we examine the behavior of WL algorithms under a synthetic microbenchmark. We evaluate three representative wear leveling (WL) algorithms, **Dual-Pool (DP)** [3], **Progressive Wear Leveling (PWL)** [6], **Dynamic Adjustment Garbage Collection (DAGC)** [7], and Table 1 summarizes their characteristics.

2.1 Experimental Setup

We extend FTLSim [9]¹ for our experiments. This prior work validates the analytical model for SSD performance and thus focuses on accurately modeling SSD-internal statistics, rather than SSD-external performance such as latency and throughput. This allows us to simulate the entire lifetime of an SSD (a few hundreds of TiB written) within a few months and also observe its internal activities. Table 2 summarizes the SSD configuration and policies for our experiments.

To understand the behavior of WL techniques, we synthetically generate workload to control the I/O pattern better. All I/Os are small random writes, but the distribution is controlled by two parameters r and h ($0 < r < 1$ and $0 < h < 1$): r fraction of writes go to the h fraction of the footprint (hot addresses) [29]. We use r/h to indicate that

the r fraction of writes occur on the h fraction of the workload footprint. Unless otherwise noted, we generate the I/O workload for the entire logical address space. Prior to each experiment, we pre-condition the SSD with one sequential full-drive write, followed by three random full-drive writes (256 GiB sequential + 768 GiB random write) to put the drive into a steady-state [31].

2.2 Performance of Wear Leveling

We investigate the performance of wear leveling in the following three aspects: (1) write amplification, (2) effectiveness in equalizing the erase count, and (3) behaviors under different access footprints.

Write amplification. We measure the WL-induced write amplification (WA) by using a synthetic workload of $r/h = 0.9/0.1$ for up to 100 full-drive writes (25 TiB). The WL parameter values we experiment with are similar to those used in the prior work [3, 6, 7]. Figure 2 shows the write amplification, and we make the following four observations. First, the overall write amplification can be as high as 11.49, in which 5.4 is caused by WL. This overhead is as much as the WA caused by garbage collection. This means that for each 256 GiB user data written, wear leveling alone will create an additional 1.35 TiB of data writes internally. Second, the WA is sensitive to the WL threshold parameter, TH . Changing the TH from 10 to 5 for the DP algorithm will amplify the amount of data written to 1.6×. Third, PWL produces a significantly high WA of 11.49 once the SSD ages beyond 80 full-drive writes. PWL is an adaptive WL algorithm, and it becomes overly aggressive at a later stage while being dormant during the early stage. Lastly, WA steadily increases over time as the SSD ages, indicating that SSD aging will accelerate as more data are written.

Table 2: SSD configuration and policies. Only the parameters relevant to understanding the wear leveling behavior are shown.

Parameter	Value	Parameter	Value
Page size	4 KiB	Physical capacity	284 GiB
Pages per block	256	Logical capacity	256 GiB
Block size	1 MiB	Over-provisioning	11%
Block allocation	FIFO	Garbage collection	Greedy

¹Our extension is available at <https://github.com/ZiyangJiao/FTLSim-WL>.

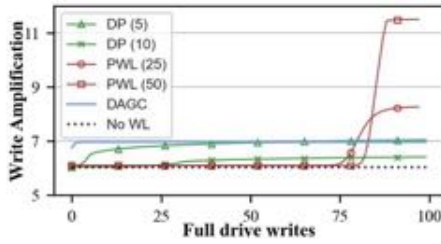


Figure 2: The write amplification caused by wear leveling under a $r/h = 0.9/0.1$ synthetic workload. The parenthetical values in the legends are the WL threshold parameters. $PWL(50)$ that aggressively performs wear leveling at the late stage causes its WAF to be as high as 11.49.

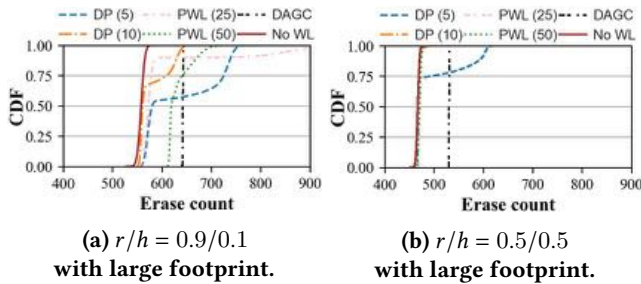
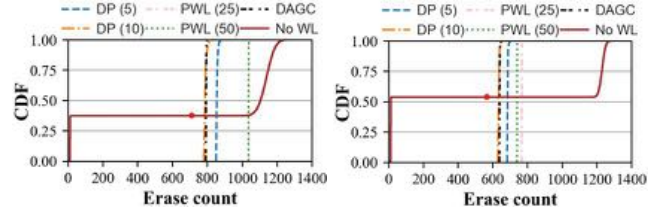


Figure 3: The distribution of erase count when full logical address space is used after writing 25TiB. WL shows the performance anomaly under $r/h = 0.9/0.1$ workload (Figure 3a). On the other hand, the benefit from wear leveling is negligible compared to not running at all under $r/h = 0.5/0.5$ (Figure 3b).

Wear leveling effectiveness. We measure the distribution of erase count under a synthetic workload as shown in Figure 3. We perform 100 full-drive writes (25 TiB) using a workload with $r/h = 0.9/0.1$ (Figure 3a), and with $r/h = 0.5/0.5$ (Figure 3b).

With $r/h = 0.9/0.1$, as shown in Figure 3a, all configurations of DP and PWL show a worse distribution of erase counts than not running WL ($NoWL$). DP and PWL show a concave dip in the CDF curve, indicating a bimodal distribution of erase counts. $NoWL$, on the other hand, shows a nearly vertical line, meaning that the erase counts are more tightly distributed. We consider this to be a *performance anomaly* of wear leveling because it behaves the opposite of what is expected. We examine the bimodal distribution of $DP(5)$ and find that the blocks associated with the cold pool are older than those in the hot pool. The DP algorithm’s underlying assumption is that blocks containing hot data are older than blocks with cold data, and it compares the erase count of the oldest block in the hot pool and the youngest block in the cold pool. If the youngest block in the cold pool



(a) $r/h = 0.9/0.1$ with small footprint. (b) $r/h = 0.5/0.5$ with small footprint.

Figure 4: The distribution of erase count when only 5% of the logical address space is used. The red dot indicates the average erase count for $NoWL$.

happens to be older than the oldest block in the hot pool, however, it will still trigger the swap between the two blocks, causing this inversion. DAGC also achieves good evenness but amplifies data writes by 18% compared to $NoWL$.

On the other hand, with a uniformly random workload (Figure 3b), there is a negligible difference among DP, PWL, and $NoWL$. This is because, with a uniform workload, all blocks are used equally, and there is little room for wear leveling. We do observe, however, that $DP(5)$ still exhibits a performance anomaly though at a smaller degree than under $r/h = 0.9/0.1$ (cf. Figure 3a). As for DAGC, the overall efficiency for garbage collection is reduced as its victim selection considers both valid ratio and erase count, incurring 15% more data writes than $NoWL$.

These experiments show that WL algorithms are a double-edged sword. As shown in Figure 3a, it can make the distribution of wear worse than not running WL at all. On the other hand, it can achieve good wear leveling but at a high cost of accelerated overall wear state.

Small access footprint. Here we explore the performance of wear leveling when the accesses are restricted to a small address space (5% of total) using two synthetic workloads, $r/h = 0.9/0.1$ and $r/h = 0.5/0.5$, as shown in Figure 4.

Overall, we observe that most WL techniques are effective in equalizing the erase count, as shown by the near-vertical CDF curve in both Figure 4a and Figure 4b. $NoWL$, on the other hand, shows a bimodal distribution between used blocks and unused blocks in both workloads. We also observe that when the workload is skewed (Figure 4a), the WL techniques achieve this evenness by amplifying the amount of data writes, as shown by the rightward shift in the CDF curves. For a uniform workload, on the other hand (Figure 4b), the overall write amplification from wear leveling is much lower as data are equally likely to be invalidated.

Unlike the results from Figure 3a and Figure 3b where the entire logical address space is written, WL is effective only when a small fraction of the address space is used, restricting its overall usefulness.

2.3 Summary of Findings

Table 4 summarizes the effectiveness of WL from our experiments using synthetic workloads. Only when the access pattern is uniform and footprint is small, WL is beneficial; otherwise, it is detrimental or has negligible effect.

Instead of proposing a new wear leveling algorithm that solves both the write amplification overhead and performance anomaly, we question the circumstances that require wear leveling and examine its necessity in the next section.

Table 4: Qualitative effectiveness of wear leveling.

	Skewed access	Uniform access
Large footprint	Anomalous (Figure 3a)	Negligible (Figure 3b)
Small footprint	Write amplified (Figure 4a)	Effective (Figure 4b)

3 CASE STUDY ON CAPACITY VARIANCE

If the interface were to allow a reduction in the SSD’s exported capacity, WL becomes unnecessary as it does not need to ensure that all blocks wear out evenly. The idea of capacity variance is not new [18]: the Zoned Namespace (ZNS) specification allows zones to be taken offline [34], effectively shrinking the SSD’s capacity. In this section, we study such a case of capacity reduction and the overall lifetime of the SSD with and without WL.

We implement a capacity-variant SSD on the extended FTLsim [9] from § 2 and use the SSD configuration in Table 2 for our evaluation. However, we set the endurance limit to 500 erases, a typical level for QLC [21, 22], and once a block reaches this, it will be mapped out and no longer used in the SSD, effectively reducing the SSD’s physical capacity. For the fixed capacity SSD, the SSD is considered to reach its end of life once the physical capacity becomes smaller than its logical capacity: the SSD is considered to have failed once

this happens. On the other hand, the capacity-variant SSD gracefully reduce its capacity below the initial logical space, to a user defined threshold (if set) or as low as the access footprint for the workload. For a capacity-variant SSD, if the logical capacity can no longer be reduced without losing user data, the SSD is considered to have failed.

For the workload, we use nine real-world block I/O traces that were collected from running YCSB [36], a virtual desktop infrastructure (VDI) [19], and Microsoft production servers (WBS, DTRS, DAP-PS, LM-TBE, MSN-CFS, MSN-BEFS, RAD-BE) [17]. In particular, the Microsoft production traces are outdated, but we use it to include a wider variety of workloads. The traces are modified into a 256GiB range (the logical capacity of the SSD), and all the requests are aligned to 4KiB boundaries. Similar to the synthetic workload evaluation, the SSD is pre-conditioned with one sequential full-drive write and three random full-drive writes on the entire logical space. The traces run in a loop indefinitely, continuously generating I/O until the SSD becomes unusable. Table 3 summarizes the trace workload characteristics.

We evaluate the following eight designs.

Fix_NoWL runs no WL on a fixed capacity SSD.

Fix_DP runs $DP(5)$ on a fixed capacity SSD.

Fix_PWL runs $PWL(50)$ on a fixed capacity SSD.

Fix_DAGC runs $DAGC$ on a fixed capacity SSD.

Var_NoWL runs no WL on a capacity-variant SSD.

Var_DP runs $DP(5)$ on a capacity-variant SSD.

Var_PWL runs $PWL(50)$ on a capacity-variant SSD.

Var_DAGC runs $DAGC$ on a capacity-variant SSD.

Figure 5 shows the amount of data written to the SSD before failure for the nine I/O traces. The y -axis is in terms of the number of drive writes. For example, for 100 drive writes, 25TiB of data have been written. Overall, we observe that with fixed capacity SSDs, running WL is better than not running WL, but only by a small margin: *Fix_DP* extends the lifetime by only 13% on average compared to *Fix_NoWL*, and with workloads such as VDI and DTRS, *Fix_DP* and *Fix_DAGC* perform worse than *Fix_NoWL*. However, with

Table 3: Trace workload characteristics. YCSB-A is from running YCSB [36], VDI is from a virtual desktop infrastructure [19], and the remaining 7 (from WBS to RAD-BE) are from Microsoft production servers [17].

Workload	Description	Footprint (GiB)	Avg. write size (KiB)	Hotness (r/h)	Sequentiality
YCSB-A	User session recording	89.99	50.48	64.69/35.31	0.49
VDI	Virtual desktop infrastructure	255.99	17.99	64.45/35.55	0.14
WBS	Windows build server	56.05	27.82	60.34/39.66	0.02
DTRS	Developer tools release	150.63	31.85	54.20/45.80	0.12
DAP-PS	Advertisement payload	36.06	97.20	55.02/44.98	0.16
LM-TBE	Map service backend	239.49	61.90	60.29/39.71	0.94
MSN-CFS	Storage metadata	5.58	12.92	69.28/30.72	0.25
MSN-BEFS	Storage backend file	31.42	11.62	70.18/29.82	0.03
RAD-BE	Remote access backend	14.73	13.02	65.51/34.49	0.33

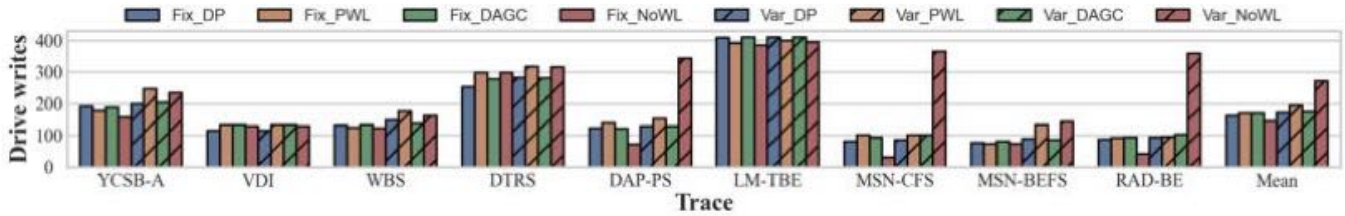


Figure 5: Evaluation of the presence and absence of wear leveling in both a fixed capacity and a capacity-variant SSD. Capacity variance extends the lifetime by 86% on average, and as high as 2.94× in the case of RAD-BE.

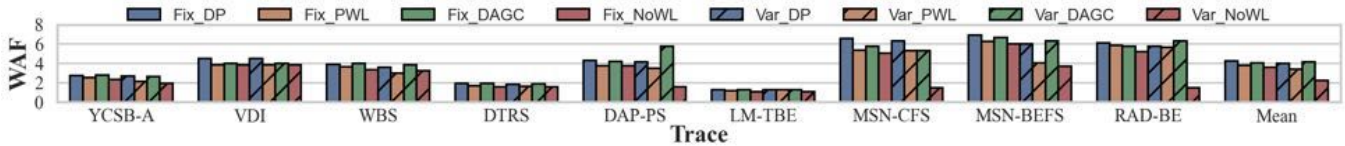


Figure 6: Write amplification caused by WL and GC. While a large sequential workload such as LM-TBE only has a low write amplification overhead of 1.26, most other workloads exhibit high wear leveling overhead for DP and DAGC, reaching as high as 6.9. Without wear leveling, the average write amplification is only 2.89.

capacity variance, not running WL is better than running WL by a large margin: *Var_NoWL* extends the lifetime by 86% on average, and as much as 2.94× for RAD-BE. We explain this result by the measurement of write amplification caused by wear leveling, shown in Figure 6.

Workloads with a relatively small footprint. We observe that capacity variance is most effective on workloads such as DAP-PS, MSN-CFS, and RAD-BE. These workloads are characterized by a small access footprint where gracefully reducing the capacity achieves more lifetime extension than WL. Specifically, capacity variance without wear leveling allows 2.94× more data to be written to the SSD for RAD-BE.

MSN-BEFS also has a small footprint, but we observe a comparatively lower lifetime extension of 0.91×. In fact, the lifetime of *Fix_NoWL* isn’t too far off from that of *Fix_DP*, only 5% less. The reason for this is due to garbage collection: This workload contains a lot of small random writes, causing garbage collection to be active, dwarfing the WL-induced WAF. Because of this, MSN-BEFS only allows 145 full-drive writes (36.27 TiB) even for the capacity-variant SSD.

Workloads with a relatively large footprint. LM-TBE and VDI are two workloads with the largest footprint, and the benefit of capacity variance is diminished in such workloads. However, we find that capacity variance still achieves the similar lifetime extension compared to the best case via WL under this scenario: for VDI, *Fix_PWL* extends the lifetime by only 3.1% compared to *Var_NoWL*, and for LM-TBE, *Fix_DP* extends it by only 3.6% compared to *Var_NoWL*. A large footprint means that there is little to gain from reducing the capacity as data are still in use. For LM-TBE, the large sequential write with relatively high uniformity causes the write amplification for WL to be small, as low as 1.18. This allows wear leveling to squeeze more writes out of the SSD.

DTRS is one of the rare occasions where not running WL is better in a fixed capacity SSD. *Fix_NoWL* allows 18% more writes compared to *Fix_DP*, and 7% more compared to *Fix_DAGC*. This is due to the high write amplification of wear leveling. Although the write access pattern of DTRS is fairly uniform, we suspect that a wear leveling anomaly occurred, causing a subset of blocks to age rapidly. Introducing capacity variance extends the lifetime for all three cases, however, with *Var_NoWL* extending the lifetime by 24% compared to *Fix_DP*. *Var_PWL* outperforms *Var_NoWL*, but the difference is only 3%.

4 DISCUSSION AND RELATED WORK

Wear leveling is a mature and well-understood topic in both academia and industry, but *getting it right* has proven to be difficult as shown by the recent large-scale field study [23]. This study on millions of modern SSDs shows that some blocks wear out 6× faster than the average, revealing the ineffectiveness of wear leveling algorithms. We discuss related work on wear leveling and file system support necessary for capacity variance.

Wear leveling and write amplification. There exists a large body of work on garbage collection and its associated write amplification (WA) for SSDs, from analytical approaches [9, 12, 27, 37] to experimental results [4, 16, 38]. However, there is surprisingly limited work that measures the WA caused by wear leveling (WL), and they often rely on a *back-of-the-envelope calculation* for estimating the overhead and lifetime [39]. Even those that perform a more rigorous study evaluate the efficacy of WL by measuring the amount of writes the SSD can endure [6, 14, 20, 35] or the distribution of erase count [1, 3, 13]; only the Dual-Pool algorithm [3] present the overhead of WL.

File system support. Using a capacity-variant SSD would need support from the file system. Thankfully, the current system design can make this transition less painful for the following reasons. First, the TRIM command, widely supported by interface standards such as NVMe allows the file system to explicitly declare that the data (at the specified addresses) are no longer in use. This allows the SSD to discard the data safely and would help determine if the exported capacity can be gracefully reduced. Second, modern file systems can safely compact their content so that the data in use are contiguous in the logical address space. Log-structure file systems such as F2FS support this more readily, but file system defragmentation can also achieve the same effect in in-place update file systems such as ext4. Lastly, zoned namespace (ZNS), a new abstraction for storage devices that gained significant interest in the research community [2, 11, 32], already supports shrinking the device capacity by taking zones offline [34]. The capacity variance potentially incurs overhead for the file system to relocate data from one logical space to another. Naïvely, the file system would relocate not only the data at the high address space, but also update any metadata for the block allocation and inode. A more advanced command such as SHARE [25] can be used to reduce the relocation overhead.

5 CONCLUSION

From a system design standpoint, it is easier to build the storage stack with a fixed capacity abstraction. However, this abstraction requires the implementation of a wear leveler in SSDs that is surprisingly both ineffective and inefficient. Furthermore, with increasing flash memory block size and decreasing endurance limit for flash, we expect the wear leveling problem to exacerbate in the near future. We believe it is necessary to re-think the benefits and costs of the wear leveler in SSDs and the block interface abstraction.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions that help us to improve the quality of this paper. Peter Desnoyers provided the original version of FTLsim used in this work. This research was supported, in part, by the National Science Foundation awards CNS-2008324 and CNS-2008453.

REFERENCES

- [1] Nitin Agrawal, Vijayan Prabhakaran, Ted Wobber, John D. Davis, Mark S. Manasse, and Rina Panigrahy. 2008. Design Tradeoffs for SSD Performance. In *USENIX Annual Technical Conference (ATC)*. 57–70.
- [2] Matias Björling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *USENIX Annual Technical Conference (ATC)*. 689–703.
- [3] Li-Pin Chang. 2007. On efficient wear leveling for large-scale flash-memory storage systems. In *ACM Symposium on Applied Computing (SAC)*.
- [4] Li-Pin Chang, Tei-Wei Kuo, and Shi-Wu Lo. 2004. Real-time garbage collection for flash-memory storage systems of real-time embedded systems. *ACM Trans. Embed. Comput. Syst.* 3, 4 (2004), 837–863.
- [5] Yuan-Hao Chang, Jen-Wei Hsieh, and Tei-Wei Kuo. 2010. Improving Flash Wear-Leveling by Proactively Moving Static Data. *IEEE Trans. Computers* 59, 1 (2010), 53–65.
- [6] Fu-Hsin Chen, Ming-Chang Yang, Yuan-Hao Chang, and Tei-Wei Kuo. 2015. PWL: a progressive wear leveling to minimize data migration overheads for NAND flash devices. In *Design, Automation & Test in Europe Conference & Exhibition, (DATE)*.
- [7] Zhe Chen and Yuelong Zhao. 2020. DA-GC: A Dynamic Adjustment Garbage Collection Method Considering Wear-leveling for SSD. In *Great Lakes Symposium on VLSI (GLSVLSI)*. 475–480.
- [8] Mei-Ling Chiang, Paul CH Lee, and Ruei-Chuan Chang. 1999. Using data clustering to improve cleaning performance for flash memory. *Software: Practice and Experience* 29, 3 (1999), 267–290.
- [9] Peter Desnoyers. 2012. Analytic modeling of SSD write performance. In *International Systems and Storage Conference (SYSTOR)*.
- [10] Thomas Gleixner, Frank Haverkamp, and Artem Bityutskiy. 2006. UBI - Unsorted Block Images. <http://linux-mtd.infradead.org/doc/ubidesign/ubidesign.pdf>.
- [11] Kyuhwa Han, Hyunho Gwak, Dongkun Shin, and Jooyoung Hwang. 2021. ZNS+: Advanced Zoned Namespace Interface for Supporting In-Storage Zone Compaction. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 147–162.
- [12] Xiao-Yu Hu, Evangelos Eleftheriou, Robert Haas, Ilias Iliadis, and Roman A. Pletka. 2009. Write amplification analysis in flash-based solid state drives. In *Israeli Experimental Systems Conference (SYSTOR)*. 10.
- [13] Jian Huang, Anirudh Badam, Laura Caulfield, Suman Nath, Sudipta Sengupta, Bikash Sharma, and Moinuddin K. Qureshi. 2017. FlashBlox: Achieving Both Performance Isolation and Uniform Lifetime for Virtualized SSDs. In *USENIX Conference on File and Storage Technologies (FAST)*. 375–390.
- [14] Xavier Jimenez, David Novo, and Paolo Ienne. 2014. Wear unleveling: improving NAND flash lifetime by balancing page endurance. In *USENIX conference on File and Storage Technologies (FAST)*, Bianca Schroeder and Eno Thereska (Eds.). 47–59.
- [15] Han joon Kim and Sang goo Lee. 2002. An Effective Flash Memory Manager for Reliable Flash Memory Space Management. *IEICE Transactions on Information and Systems* 85 (2002), 950–964.
- [16] Won-Kyung Kang, Dongkun Shin, and Sungjoo Yoo. 2017. Reinforcement Learning-Assisted Garbage Collection to Mitigate Long-Tail Latency in SSD. *ACM Trans. Embed. Comput. Syst.* 16, 5s (2017), 134:1–134:20.
- [17] Swaroop Kavalanekar, Bruce L. Worthington, Qi Zhang, and Vishal Sharda. 2008. Characterization of storage workload traces from production Windows Servers. In *International Symposium on Workload Characterization (IISWC)*.
- [18] Bryan S. Kim, Eunji Lee, Sungjin Lee, and Sang Lyul Min. 2019. CPR for SSDs. In *Workshop on Hot Topics in Operating Systems (HotOS)*.
- [19] Chunghan Lee, Tatsuo Kumano, Tatsuma Matsuki, Hiroshi Endo, Naoto Fukumoto, and Mariko Sugawara. 2017. Understanding storage traffic characteristics on enterprise virtual desktop infrastructure. In *ACM International Systems and Storage Conference (SYSTOR)*.
- [20] Sungjin Lee, Taejin Kim, Kyungho Kim, and Jihong Kim. 2012. Lifetime management of flash-based SSDs using recovery-aware dynamic throttling. In *USENIX conference on File and Storage Technologies (FAST)*.

- [21] Shuwen Liang, Zhi Qiao, Sihai Tang, Jacob Hochstetler, Song Fu, Weisong Shi, and Hsing-Bung Chen. 2019. An Empirical Study of Quad-Level Cell (QLC) NAND Flash SSDs for Big Data Applications. In *IEEE International Conference on Big Data (Big Data)*. 3676–3685.
- [22] Chun-Yi Liu, Yunju Lee, Myoungsoo Jung, Mahmut Taylan Kandemir, and Wonil Choi. 2021. Prolonging 3D NAND SSD Lifetime via Read Latency Relaxation. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 730–742. <https://doi.org/10.1145/3445814.3446733>
- [23] Stathis Maneas, Kaveh Mahdavian, Tim Emami, and Bianca Schroeder. 2022. Operational Characteristics of SSDs in Enterprise Storage Systems: A Large-Scale Field Study. In *USENIX Conference on File and Storage Technologies (FAST)*. 165–180. <https://www.usenix.org/conference/fast22/presentation/maneas>
- [24] Muthukumar Murugan and David Hung-Chang Du. 2011. Rejuvenator: A static wear leveling algorithm for NAND flash memory with minimized overhead. In *IEEE Symposium on Mass Storage Systems and Technologies (MSST)*.
- [25] Gihwan Oh, Chiyoung Seo, Ravi Mayuram, Yang-Suk Kee, and Sang-Won Lee. 2016. SHARE Interface in Flash Storage for Relational and NoSQL Databases. In *International Conference on Management of Data (SIGMOD)*. 343–354.
- [26] Open NAND Flash Interface. 2021. ONFI 5.0 Spec. <http://www.onfi.org/specifications/>.
- [27] Changhyun Park, Seongjin Lee, Youjip Won, and Soohan Ahn. 2017. Practical Implication of Analytical Models for SSD Write Amplification. In *ACM/SPEC on International Conference on Performance Engineering (ICPE)*. 257–262.
- [28] Yi Qin, Dan Feng, Jingning Liu, Wei Tong, and Zhiming Zhu. 2014. DT-GC: Adaptive Garbage Collection with Dynamic Thresholds for SSDs. In *2014 International Conference on Cloud Computing and Big Data*. 182–188. <https://doi.org/10.1109/CCBD.2014.28>
- [29] Mendel Rosenblum and John K. Ousterhout. 1992. The Design and Implementation of a Log-Structured File System. In *PhD thesis, University of California at Berkeley*.
- [30] Sandisk. 2003. Sandisk Flash Memory Cards Wear Leveling. <http://www.sandisk.com/Assets/File/OEM/WhitePapersAndBrochures/RS-MMC/WPaperWearLevelv1.0.pdf>.
- [31] Esther Spanjer and Easen Ho. 2011. The Why and How of SSD Performance Benchmarking - SNIA. https://www.snia.org/sites/default/education/tutorials/2011/fall/SolidState/EstherSpanjer_The_Why_How_SSD_Performance_Benchmarking.pdf.
- [32] Theano Stavrinos, Daniel S. Berger, Ethan Katz-Bassett, and Wyatt Lloyd. 2021. Don't be a blockhead: zoned namespaces make work on conventional SSDs obsolete. In *Workshop on Hot Topics in Operating Systems (HotOS)*. 144–151.
- [33] STMicro. 2006. Wear Leveling in Single Level Cell NAND Flash Memories. STMicroelectronics Application Note (AN1822).
- [34] Western Digital. 2020. Zoned Namespaces (ZNS) SSDs. <https://zonedstorage.io/introduction/zns/>.
- [35] Ellis Herbert Wilson, Myoungsoo Jung, and Mahmut T. Kandemir. 2014. ZombieNAND: Resurrecting Dead NAND Flash for Improved SSD Longevity. In *IEEE International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*. 229–238.
- [36] Gala Yadgar, Moshe Gabel, Shehbaz Jaffer, and Bianca Schroeder. 2021. SSD-based Workload Characteristics and Their Performance Implications. *ACM Trans. Storage* 17, 1 (2021), 8:1–8:26.
- [37] Yudong Yang, Vishal Misra, and Dan Rubenstein. 2015. On the Optimality of Greedy Garbage Collection for SSDs. *SIGMETRICS Perform. Evaluation Rev.* 43, 2 (2015), 63–65.
- [38] Qi Zhang, Xuandong Li, Linzhang Wang, Tian Zhang, Yi Wang, and Zili Shao. 2015. Optimizing deterministic garbage collection in NAND flash storage systems. In *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 14–23.
- [39] Tao Zhang, Aviad Zuck, Donald E. Porter, and Dan Tsafir. 2017. Flash Drive Lifespan *is* a Problem. In *Workshop on Hot Topics in Operating Systems (HotOS)*. 42–49.

Do Temperature and Humidity Exposures Hurt or Benefit Your SSDs?

Adnan Maruf*, Sashri Brahmakshatriya*, Baolin Li[†], Devesh Tiwari[†], Gang Quan*, and Janki Bhimani*

*Florida International University [†]Northeastern University

Abstract—SSDs are becoming mainstream data storage devices, replacing HDDs in most data centers, consumer goods, and IoT gadgets. In this work, we ask an uncharted research question: What is the environmental conditions' impact on SSD performance? To answer it, we systematically measure, quantify, and characterize the impact of various commonly changing environmental conditions such as temperature and humidity on the performance of SSDs. Our experiments and analysis uncover that exposure to changes in temperature and humidity can significantly affect SSD performance.

Index Terms—robust performance, SSDs, design of tests.

I. INTRODUCTION

Mitigating environmental impacts from temperature and humidity has been a challenge for all kinds of computer systems, from high-performance supercomputers [1] to edge computing platforms [2]. Such environmental impacts have caused severe damage to large data centers, leading to long-duration service outages in the past. For example, Amazon's AWS and Microsoft's Azure datacenter failures were caused by unexpected weather in 2021 [3], 2020 [4], and 2018 [5]. Interestingly, a significant number of cases are caused by the storage system's performance degradation and failure under such impacts [6]. Therefore, system-level online testing for appropriate diagnosis is critical towards the robust performance and reliability of storage systems while operating under various environmental conditions [7]. In the storage world, many Hard Disk Drives (HDD) are being replaced by flash-based Solid State Drives (SSD) by most data center providers worldwide for improved performance and reliability of SSDs over HDDs [8], [9]. We find very scarce literature that identifies SSD failures in adverse environments [10], [11], [12], [13]. But none of the studies present the fine-grained runtime and post-performance effects of SSDs exposed to various commonly experienced temperatures and humidity. We believe the root cause of the vulnerability may be essentially different for SSDs compared to HDDs, as SSDs rely on integrated circuit (IC) modules, which may be gradually impacted by temperature and humidity [14], [15], [16] than HDDs' mechanical components. So, it is very important to carefully design controlled accelerated lab tests for high significance diagnoses that can later be used to predict storage failure and prevent data loss.

SSDs consist of the components such as NAND cells, memory controller, SATA/NVMe interface, onboard DRAM, capacitors, and other integrated circuits [17]. How the temperature and humidity impact SSDs would be the reflection of the effects of temperature and humidity on these internal components of SSDs. We found much evidence from the literature [10], [14], [18], [19], [20] on the solid-state in physics that shows that the high-temperature age the NAND cells more quickly than at the normal temperature due to the acceleration of charge leakage

(i.e., retention loss) at a superlinear rate. Moreover, previous researches [15], [16] on electron devices such as capacitors and ICs found that the humidity levels impacted the lifetime of the capacitors and performance of ICs, due to an increase in interconnect capacitance and dielectric loss. *Therefore, our core insight is that the SSD performance is high-likely to be impacted by changes in temperature and humidity, as the literature [10], [14], [16], [18], [19], [20] shows scattered evidence of various components of SSD such as NAND flash cells, ICs, and capacitors are individually impacted.*

In this paper, we design accelerated system-level online tests to understand the runtime and post-performance effects of various commonly experienced temperature and humidity changes on SSDs manufactured by multiple vendors. We provide an in-depth analysis of how the SSD performance is impacted under a range of realistic environment settings. In particular, this is the first work to investigate the following research questions (RQs):

RQ1: What are the runtime impacts of temperature and humidity changes on SSD performance?

RQ2: What are the post-impacts of exposure to temperature and humidity?

RQ3: How do the impacts of the exposure to temperature and humidity on SSD performance vary across different SSD types and I/O operation types (e.g., read and write)?

To ensure the high significance of our diagnoses and reproducibility of our experiments, we repeat each experiment over six new out-of-the-box SSDs. In total, we tested over a hundred SSDs to derive the results presented in this paper. We ensured that the humidity and temperature values were always within the vendor-specified limits for all our experiments. However, by the end of each experiment, many SSDs came out bruised due to post impacts. Some SSDs succumbed to their adverse aging effects due to our accelerated experiments performing a large amount of I/Os. Thus, one of the biggest challenges of this study was not being able to reuse the same SSD for multiple experiments, as that would impact the correctness of our observations. The high cost of data collection and long experimentation time perturb us from exhaustively collecting data at every possible temperature and humidity within vendor-specified limits. We collect and analyze a large amount of sensor and performance data from each experiment using various I/O tools. Here, we share our selected findings and observations that we could conclude with high statistical significance. Anonymized experimental data is being made publicly available at <https://github.com/adnanmaruf/SSD-Temp-Humid> and for the research community to understand better, model, and design alternative solutions to overcome the adverse effects.

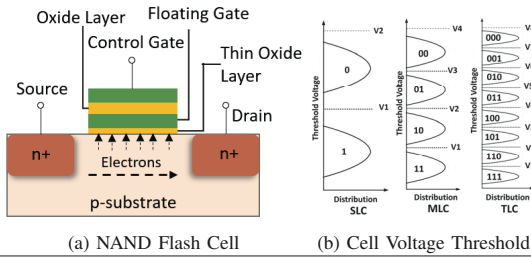


Fig. 1. NAND Flash cell types.

II. BACKGROUND

Among many components of an SSD, NAND flash cells, DRAM, memory controller, capacitors, and ICs are the key ones. In this section, we briefly discuss the details of some of these internal components of the SSD.

[NAND flash cells] are used in SSD devices because of their density, durability, cost, and performance. NAND flash uses Floating-Gate MOSFET (FGMOS) transistors to store data. Fig. 1a shows a simple FGMOS cell. Similar to the MOSFET transistors, the FGMOS acts like an electrical switch where current flows between the source and drain terminal. The MOSFET channel becomes conductive when a voltage greater than the threshold voltage (V_1) is applied to the control gate (CG). Instead of only the CG in MOSFET, in FGMOS, there is another gate called floating gate (FG) to control the flow of current. The FG is separated from the CG and the MOSFET channel by the oxide layer. When a high positive voltage is applied to the CG and a high negative voltage is applied to the source, electrons tunnel through the thin oxide layer and reach the FG. This operation is called tunneling. Electrons trapped inside the FG stay there even after the tunneling operation, making the FGMOS a non-volatile memory cell that can store data. When FG is charged with electrons, the threshold voltage is increased to V_2 ($V_2 > V_1$, e.g., see SLC in Fig. 1b) and the channel will be conductive only when a voltage greater than the V_2 is applied to the CG. Now, depending upon the voltage at which the channel conducts, we can read the bit stored within this FGMOS [21]. This is how data is read from the NAND flash cell. As shown in Fig. 1b, a single-level cell (SLC) stores a single bit, either 0 or 1 differentiated with one threshold voltage, multi-level cell (MLC) stores two-bit data differentiated with three threshold voltages, and triple-level cell (TLC) stores three-bit data differentiated with seven threshold voltages to read the data.

Tunneling is also used to release the electrons from the FG. This time a high negative voltage to the CG and high positive voltage is applied to the source, which is the erase operation of the NAND flash. The tunneling used for both writes and erase operations gradually deteriorates the thin oxide layer, allowing the electrons to get inside and out from the FG more freely. This is known as the retention loss or charge leakage of the NAND cell. Such retention loss leads to an increase in the raw bit error rate (RBER). Thus, different methods like read-retry, error-correcting code (ECC) bits are used to ensure the correctness of the data at the cost of increased I/O latency of the read and write operations.

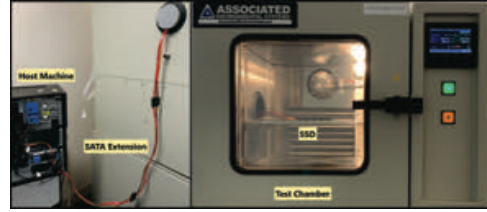


Fig. 2. Testbed for SSD environmental vulnerability tests.

[Integrated Circuits (IC)] are key elements in modern electronics. A set of electronic components, e.g., resistors, transistors, capacitors, etc., are integrated into a small semiconductor material-based chip. Thus, ICs are a magnitude smaller, high-performing, cost-efficient compared to discrete components. The fabrication process of ICs has two main processing, Front-end-of-line (FEOL), when IC components are formed directly on the semiconductor material like silicon, and Back-end-of-line (BEOL), when all components are integrated to interconnect them with metal wiring.

[Chip capacitors] are another key component in SSDs as almost all ICs use capacitors. The volatile memory, i.e., random access memory unit used in the SSD, is mainly based on capacitors. In a capacitor, electrodes are separated by a dielectric medium such as air, vacuum, paper, titanium, etc. The metallic electrodes hold the charge, and the electricity starts to flow once the plates are connected. The capacitance of the capacitor depends on the area of the metal plates, the distance between the plates, and the dielectric material.

III. EXPERIMENTAL METHODOLOGY

This section describes our testbed setup, experiment sequences in different temperature and humidity levels, benchmark implementation, and performance metrics. Our experiments are designed to perform controlled tests on the SSD under different environmental conditions to capture performance impact accurately.

A. Experiment Setup

To ensure that the temperature and humidity impacts are only applied to the SSD and are isolated from the other components of our host machine, we use a specially designed test chamber [22] to conduct the experiments. Our test chamber can maintain a steady-state temperature and humidity without being affected by each other or external environmental conditions. The host machine and the test chamber are placed in an isolated room with HVAC to keep the room temperature and humidity constant during the experiments. We have set up our testbed to ensure that only the SSD is exposed to the temperature and humidity control. At the same time, the other components of the host machine are kept under constant room temperature and humidity. The full setup is illustrated in Fig. 2. The main components of the testbed include a host machine, the SATA extension cables, the test chamber, and the SSD unit under test. The SSD is extended from the host machine connector using SATA extension cables through a sealed portal on the side of the test chamber. The host machine runs the I/O benchmark while the test chamber controls the environmental conditions

TABLE I
TESTBED SPECIFICATIONS.

Component	Specs
Test Chamber	AES LH-1.5 [22]
Host Server	Optiplex9020
Processor	Intel(R) Core(TM) i7-4770 CPU, 3.4GHz
Cores, L3, DRAM	16 Cores, 8192K,16GB
Operating System	Ubuntu 16.04 LTS (4.4.0-137-generic)
SSD Capacity	120 GB
SSD Type	SLC, MLC, and TLC
SATA Version	SATA 3.2, 6.0 Gb/s

TABLE II
EXPERIMENT SEQUENCES.

Exp. ID	Sequence
1	22.5°C, 50RH → (22.5-60)°C, 50RH → 60°C, 50RH → 60°C, (20-80)RH → 60°C, 50RH
2	22.5°C, 50RH → 50°C, 50RH
3	22.5°C, 50RH → 60°C, 50RH
4	60°C, 80RH → 60°C, 50RH
5	60°C, 20RH → 60°C, 50RH
6	70°C, 80RH → 10°C, 80RH
7	60°C, 50RH → 60°C, 80RH → 60°C, 50RH
8	22.5°C, 50RH → 50°C, 50RH → 22.5°C, 50RH

to which SSD is exposed. Temperature and humidity sensors are placed inside the chamber close to the SSD for chamber feedback control.

Table I shows the hardware and software specifications used in our experiments. The test chamber we purchase is from the AES temperature and humidity environmental chamber product series. Industries also use such test chambers to perform product characterization and testing [22]. The test chamber is capable of adjusting the temperature from 5°C to 94°C and relative humidity from 10% to 98%. It is equipped with a programmable automatic controller. After the temperature and humidity have reached the control setpoint during our experiments, we wait ten extra minutes before starting the I/O benchmark to avoid fluctuations. One limitation of this test chamber is that reducing relative humidity at lower temperatures requires an additional air dryer. Our test chamber is not capable of operating below 50 RH relative humidity when the temperature is below 20°C. For this reason, we do not include characterization for both low relative humidity and low temperature in our test sequences.

B. Experiment Sequences

Table II shows our experiment sequences varying the temperature and humidity. In total, we listed eight sequences. Each of the experiments at any particular environmental condition listed in Table II is six hours long. For each run of the experiment sequences in Table II, we used out-of-the-box SSDs to avoid post-impacts of the previous experiments. Before starting each experiment sequence, we preconditioned the SSDs to make them reach a steady state. Preconditioning is a process of applying workload to an SSD to move it from the initial fresh-out-of-the-box state to a state where the steady performance of the device can be reproduced while repeatedly running the same experiments. For the trusted operational range of our SSDs, we conduct all experiments within vendor-specified limits in the datasheet (e.g., 10°C to 70°C). To ensure statistical significance and reproducibility of experiments, we conducted each of the experiment sequences six times. A new SSD is

used from the three types (SLC, MLC, and TLC) for each experiment sequence in Table II repeated six times, and we present the average performance observed across them. We conduct experiments with only one SSD in the test chamber for better accuracy of our results. We also performed experiments over other sequences, but we only report the results that were interesting, and we could conclude with high statistical significance. For example, as we did not observe any impact for the humidity change at the room temperature, so we do not include that in Table II.

One of the biggest challenges of this study was not being able to reuse any SSDs for multiple experiments. Thus, these experiments are expensive and time-consuming, so designing a good test sequence is very important. First, to resemble the natural exposure to various temperatures and humidity while SSDs are deployed within electrical vehicle systems, IoT devices, and distributed data centers, we expose SSDs to continuous change in both the temperature and humidity. Specifically, Experiment ID 1 (Table II) simulates abrupt environmental factor changes over time while SSDs are in use. The temperature is varied within 22.5°C to 60°C and humidity is varied within 20RH to 80RH. 60°C and 50RH is the most common condition we found across multiple systems while processing. In idle state, 22.5°C and 50RH is the most common condition. Only while datacenter water cooling systems are fully active at high temperatures, we see high humidity of 80RH. We found an interesting impact on the performance of the SSDs, and to understand the impacts better, we conducted more controlled experiments changing only one of the environmental factors, keeping the other unchanged.

In the next five experiments, we capture the runtime effects of exposure to change in temperature or humidity. Thus, first, we benchmark SSD for six hours at the initial state of temperature and humidity, then we change the chamber conditions to the final state of temperature and humidity at which we benchmark SSDs again. We compare the performance between the initial and final states. In Experiment IDs 2 and 3 (Table II), we increase the temperature from the room condition (22.5°C) to high (50°C and 60°C) at room humidity (50RH). We observed a high positive impact on the SSD bandwidth at 60°C. We also experimented with varying the humidity at room temperature. However, we did not observe any impact on the humidity change at the room temperature. Then, we pick the best performing temperature (i.e., 60°C) to observe the impacts of decrease (Experiment ID 4 in Table II) and increase (Experiment ID 5 in Table II) in humidity compared to room level humidity. We found that decreasing humidity impacts the SSD performance positively. Next, to observe the effects of temperature decrements, we reduce the temperature within vendor-specified limit, i.e., 70°C to 10°C at 80RH in Experiment ID 6 (Table II). Note that although we observe the best performance at low humidity, we could not experiment with the temperature drop at a lower humidity level due to test chambers' limitations. We also surveyed that most test chambers have a similar limitation. Moreover, due to the inversely proportional relationship between temperature and humidity,

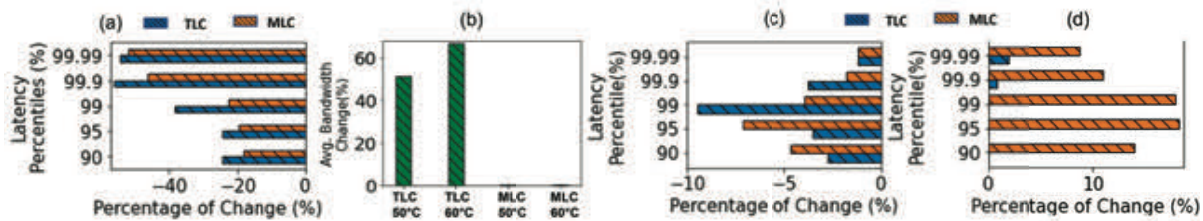


Fig. 3. (a) Tail latencies can improve up to 50% while running experiments on SSDs exposed to varying temperature and humidity (Latency - Lower the better); (b) SSDs can show higher average bandwidth at high temperature (50°C and above); (c) SSDs tail latencies decrease when humidity level decreases; and (d) tail latencies increase when humidity level increases.

this condition is least likely to happen in real environments.

Finally, we observe the post-impacts of the humidity and temperature change in Experiment IDs 7 and 8 (Table II). The initial and the final state of these experiments are the same. We compare the performance of benchmarks before and after being exposed to change in humidity and temperature. Particularly, in Experiment ID 7 (Table II), we run a six-hour experiment at room humidity and the best-performing temperature (i.e., 60°C). Then we expose SSD to high humidity and finally set back to room humidity to run our second set of six-hour experiments. Similarly, in Experiment ID 8 (Table II), we measure the post impacts of the increase in temperature. We conducted Experiment ID 8 (Table II) at room humidity rather than best performing low humidity because of the same above-explained chamber limitation that obstructs attaining low humidity.

C. I/O Benchmark Configuration

In this work, we use the popular open-source tool FIO (Flexible I/O) benchmark [23] to generate I/O workloads for the SSD. The FIO is configured to use the “libaio” I/O engine, with an I/O depth of 16, 50:50 read/write ratio, and I/O sizes of 4KB to 1MB to better mimic scenarios where the I/O queries by a real application stack. The I/O pattern is configured to be random since usually, it is the bottleneck I/O type to meet the service level agreements (SLA) in latency-critical applications. All the analyses presented in section IV, if not specifically mentioned, show the I/O performance for the mixed workloads with both read and write.

D. Performance Metrics

Among other performance metrics, tail latency is a critical metric in many real applications such as cloud computing and autonomous vehicles. Depending on the application, the tail percentile target varies. In this paper, we cover a broad range of tail latency percentiles at 90th, 95th, 99th, 99.9th, and 99.99th to account for different scenarios. The other metric we characterize is the I/O bandwidth as IOPS. IOPS captures the SSD operation throughput. We found the tail latency and the bandwidth are impacted the most compared to the other metrics, e.g., average latency and throughput. The tail latency and the bandwidth are also standard SSD performance metrics evaluated in other works [2], [24]. As this study is conducted using real SSDs and not an emulated SSD, due to the proprietary internal details of SSD such as flash transition

layer, we cannot directly instrument the internal characteristics of SSDs such as bit flip rate and read retries.

IV. RESULTS AND ANALYSIS

In this section, we discuss the results and analysis from the experiments we performed on the SSDs. We observed that among SLC, MLC, and TLC SSD types, SLC has very minimal impacts. Hence, we only discuss the results for the MLC and TLC using the most impacted metrics. We begin by discussing how SSD performance is affected during runtime temperature and humidity change. Then, we discuss the post-effects of temperature and humidity on SSD performance. Finally, we discuss the long-term impact of temperature and humidity.

Counter-intuitively, we observed that the tail latencies improved 50% after SSDs were exposed to abrupt temperature and humidity changes. To the best of our knowledge, we did not find any prior work that studies the performance impact of the SSDs exposed to temperature and humidity. So, in search of finding motivation, we begin our experiments to replicate abrupt environmental changes in temperature and humidity in Experiment ID 1 (Table II). As discussed in section I, we see disaggregated evidence from the literature [10], [14], [16], [18], [19], [20] that individually various components that are used within SSDs such as NAND flash cells, ICs, and capacitors are impacted. Thus, we anticipate the overall performance of SSDs to be impacted as well. To verify the above anticipation, in this experiment, we first measure the performance of the SSDs at the room condition, i.e., 22.5°C, 50RH. Then while running the workload, we increase the temperature and humidity. Finally, we measure the performance at 60°C, 50RH. Experiment ID 1 (Table II) shows the stages of this sequence. Fig. 3a shows the percentage decrease in tail latency for TLC and MLC SSDs. This observation motivated us to further systematically study the impact of both temperature and humidity.

We found that runtime temperature changes mostly affects the TLC SSDs. To find the impact of the temperature changes, we first analyze the performance of the SSDs at high temperatures in room humidity in Experiment IDs 2 and 3 (Table II). From Fig. 3b we observe that TLC SSDs show 51% better average bandwidth than the room condition at 50°C, and 67% better average bandwidth at 60°C. MLC SSDs showed a considerably small amount of improvements in bandwidth at a higher temperature. We have observed the runtime effects of the temperature changes mostly on the TLC NAND flash devices. When temperature increases, electrons can move more freely

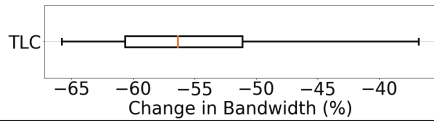


Fig. 4. Temperature decrement shows a lower bandwidth for TLC SSDs.

between the floating gate (FG) and channel [14]. We assume that the high cell electron flow due to temperature increment helps distinguish different threshold voltage levels easily (as we discussed in sec. II). Thus, TLC flash provides faster read/write operations. On the other hand, in MLC flash, there is a higher difference among the three threshold voltage levels. Hence, this small increase in the flow of electrons does not impact the MLC SSDs at all.

Next, we observe the impact of the humidity changes on the SSD performance. **We found that SSD performance deteriorates at high humidity level.** We first analyze the impact of the humidity decrements and increments at room temperature. However, we did not observe any performance changes. Then we pick 60°C for varying the humidity level as we observed high bandwidth gain at this temperature in the previous observation. We perform Experiment IDs 4 and 5 (Table II) to observe the impact of humidity change at a high temperature of 60°C. In Experiment ID 4 (Table II), humidity decreases from 80RH to 50RH, and in Experiment ID 5 (Table II), humidity increases from 20RH to 50RH. In Fig. 3c, when humidity decreases, the tail latency decreases up to 9% for TLC SSDs and 7% for MLC SSDs. In Fig. 3d, the tail latency increases when the humidity increases. Although the TLC SSDs show a very little performance degradation, the 99th tail latency of the MLC SSDs can increase up to 18%. We anticipate this performance drop at high humidity is due to the impact of humidity on the SSD IC capacitance. Ref. [16] discovered that humidity severely hinders capacitor's performance. All capacitors lose capacitance at an increasing slope, and their Equivalent Series Resistant (ESR) values climb at high humidity. Moreover, the capacitance of MLC flash cells is originally higher than TLC flash cells, so the impacts are immediately more evident for MLC SSDs. However, as on-chip DRAM cells also depend upon capacitance, so we anticipate even TLC SSDs may be impacted upon prolonged exposure.

Finally, we simulate the conditions of sudden temperature drop in case of climate control system failure. **Temperature decrement shows a lower bandwidth for TLC SSDs.** Fig. 4 shows the percentage of bandwidth change while temperature changes from 70°C to 10°C maintaining relative humidity at 80RH (Experiment ID 6 in Table II). Similar to the earlier findings, only TLC SSDs are impacted. This time the bandwidth suffers a high degradation ranging from 35% to 65%. When the temperature decreases due to the reduced mobility of electrons, distinguishing seven different threshold voltage levels becomes more challenging for TLC SSDs.

Understanding the post-impact of exposure to any environmental condition is important as this would be the lasting performance of SSD even at normal environment conditions later. **SSD tail latencies showed negative post-impact when exposed to high humidity.** To observe the post-impacts of

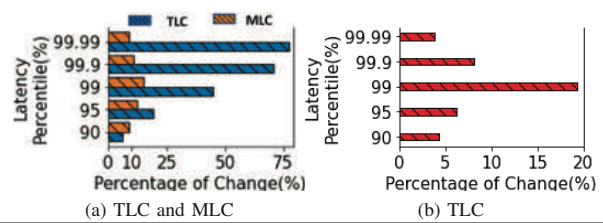


Fig. 5. Post-impacts to SSD tail latency after exposure to (a) high humidity and (b) high temperature.

the environmental changes, we conduct sequences Experiment IDs 7 and 8 (Table II). We expose SSDs to high humidity (80RH) for six hours. Then, to analyze the post-impacts, we compare the performance of SSDs over six hours prior to increasing in humidity and six hours after exposure, at room humidity. As we can see in Fig. 5a, the 99.99th tail latency of TLC and MLC SSDs can degrade up to 75% and 10% respectively. In our previous finding, we observed that high humidity degrades the runtime performance of the SSDs. From Fig. 5a, we find that the performance degradation is not only at runtime, but high humidity leaves post-impact by damaging the IC and capacitors. IC Back-End-of-Line (BEOL) components are prone to permanently suffer from humidity penetration due to their sole protection being a moisture-crack barrier [15].

We observed negative post-impact of high temperature on TLC SSDs. To explore the post-impact of the temperature changes, we perform Experiment ID 8 (Table II), where we compare the performance at normal conditions, i.e., 22.5°C, 50RH before and after exposing the SSD to 50°C, 50RH. As shown in Fig. 5b, we observe negative post-effects on TLC SSDs with an increase in the tail latencies. MLC SSDs did not show any post-impacts of increase in temperature. In the NAND flash cell, the required tunneling for write and erase operations degrades the thin oxide layer over time which causes electrons to leak from the FG. We anticipate this process of retention loss acceleration at high-temperature impacts the TLC flash cell permanently. Increased retention loss may lead to increased read retries and replacement of bad cells with new cells from over-provisioned (OP) cells, causing performance penalties [14]. As MLC flash is less sensitive to retention loss, MLC SSDs do not show post-impact of the high temperature after the short period of exposure. We did not observe any post-impacts of decreasing humidity and temperature. We also analyze the performance difference of read and write I/Os separately. We think because of the same above-mentioned reasons, we observed that **temperature and humidity changes impact the write I/Os more than the read I/Os.** We found that, on average, read I/Os bandwidth can degrade up to 62% while the write I/Os bandwidth can degrade up to 85%.

Finally, we continued our long-term experiments with the intent to let it continue until the SSD wears out by writing more data than what is specified in the warranty sheet. To our surprise, some SSDs running under high humidity observed fail-stop faults much before they surpassed the write endurance limit. We note that these failures were not observed in all SSDs, making it difficult to predict and proactively manage such failures. Also, any SSD which was under normal room

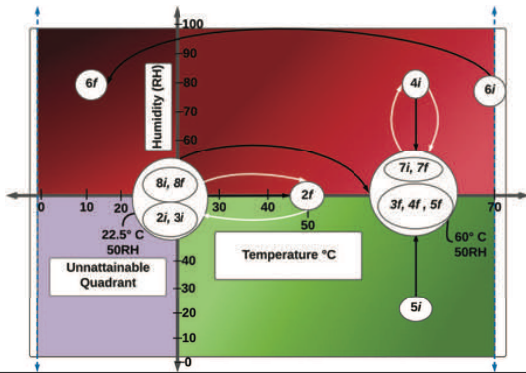


Fig. 6. Summary of the impacts of temperature and humidity changes.

conditions did not show any such behavior. The SSD failures resulted in the total loss of all the data present on the SSDs. Upon further inspection of the logs collected until SSD was operational, we observe that the “media wear-out” increased at a 2x higher rate towards the end, despite the same workload. This may be because the damaged NAND cells are rapidly replaced by spare NAND cells of the over-provision (OP) region until there are no new NAND cells to replace, and at that stage, SSD fails. For temperature changes at room and lower humidity, we do not observe any SSD failure.

We summarize our observations in Fig. 6, with temperature on the x-axis and relative humidity on the y-axis. The intersection of axes in the plot is set at the room condition, i.e., 22.5°C and 50RH. Due to the test chamber’s operating constraint, we could not run any experiment in the lower-left quadrant, i.e., low temperature-low humidity zone. All the experiment sequences from Table II are represented using white circles. The arrows indicate the sequence state of the experiments. For example, 2i denotes the initial state of Experiment ID 2 (Table II), i.e., 22.5°C, 50RH and 2f denotes the final state of Experiment ID 2 (Table II), i.e., 50°C, 50RH. The black and white-colored arrows are for the runtime and post-impact experiments, respectively. Based on the observed performance of the SSDs in different temperature and humidity levels, we colored the quadrants where the worst performance is represented by red and the best performance by green. From Fig. 6, we can see that it is safe to operate at high temperature (60°C) and low humidity (20RH) levels as it gives higher runtime performance with minimal post-impacts. In comparison, high humidity at low as well as high temperature can extremely deteriorate SSD performance and reliability.

V. CONCLUSION

This paper begins by posing a simple question for investigation: do temperature and humidity exposures hurt or benefit your SSDs? We conclude by observing that humidity has a severe post-impact on the tail latency of the SSD, even when SSDs are operating under room conditions. This can have profound implications for data center SLAs and usage of SSDs in autonomous vehicles. The extent of the impact is dependent on the NAND flash types of the SSDs. Additionally, our finding shows that a small increase in temperature may be

instantaneously beneficial to the performance of SSDs, but it may also have some minimal post impacts. In the future, we plan to further alleviate our understanding by designing models to capture the observed trends to simulate the performance behaviors, exploring different real workloads, NVMe interface, long-term impacts, and characterizing with other environmental factors such as electromagnetic waves.

ACKNOWLEDGMENTS

This work was partially supported by Cyber Florida Collaborative Seed Award, Northeastern University, and National Science Foundation Award CNS-2008324 and 1910601.

REFERENCES

- [1] UptimeInstitute. (2021) Extreme weather affects nearly half of data centers. [Online]. Available: <https://journal.uptimeinstitute.com/extreme-weather-affects-nearly-half-of-data-centers/>
- [2] S.-C. Lin, Y. Zhang, C.-H. Hsu, M. Skach, M. E. Haque, L. Tang, and J. Mars, “The architectural implications of autonomous driving: Constraints and acceleration,” in *ASPLOS*, 2018.
- [3] M. Otey. (2021) AWS Details Frankfurt Data Center Outage Cause. [Online]. Available: <https://petri.com/?p=658442>
- [4] M. Foley. (2020) Microsoft’s March 3 Azure East US outage: What went wrong (or right)? [Online]. Available: <https://www.zdnet.com/article/microsofts-march-3-azure-east-us-outage-what-went-wrong-or-right/>
- [5] TheRegister. (2018) Azure North Europe downed by the curse of the Irish – sunshine. [Online]. Available: <https://reg.cx/2K4X>
- [6] E. Xu, M. Zheng, F. Qin, Y. Xu, and J. Wu, “Lessons and actions: What we learned from 10k ssd-related storage system failures,” in *ATC*, 2019.
- [7] S. Halder and G. Sivakumar, “Embedded based remote monitoring station for live streaming of temperature and humidity,” in *ICEECCOT*, 2017.
- [8] C. Metz. (2012) Flash drives replace disks at amazon, facebook, dropbox. [Online]. Available: <https://www.wired.com/2012/06/flash-data-centers>
- [9] N. Agrawal, V. Prabhakaran, T. Wobber, J. D. Davis, M. S. Manasse, and R. Panigrahy, “Design tradeoffs for ssd performance,” in *ATC*, 2008.
- [10] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, “A large-scale study of flash memory failures in the field,” in *SIGMETRICS*, 2015.
- [11] H. Zhang, E. Thompson, N. Ye, D. Nissim, S. Chi, and H. Takiar, “Ssd thermal throttling prediction using improved fast prediction model,” in *ITherm*, 2019.
- [12] Y. Wang, X. Dong, X. Zhang, and L. Wang, “Measurement and analysis of ssd reliability data based on accelerated endurance test,” *Electronics*, 2019.
- [13] B. Schroeder, A. Merchant, and R. Lagisetty, “Reliability of nand-based ssds: What field studies tell us,” *Proceedings of the IEEE*, 2017.
- [14] Y. Luo, S. Ghose, Y. Cai, E. F. Haratsch, and O. Mutlu, “Heatwatch: Improving 3d nand flash memory device reliability by exploiting self-recovery and temperature awareness,” in *HPCA*, 2018.
- [15] F. Stellari, C. Cabral, P. Song, and R. Laibowitz, “Humidity penetration impact on integrated circuit performance and reliability,” in *IEDM*, 2019.
- [16] H. Wang, P. D. Reigosa, and F. Blaabjerg, “A humidity-dependent lifetime derating factor for dc film capacitors,” in *ECCE*, 2015.
- [17] J. Bhimani, T. Patel, N. Mi, and D. Tiwari, “What does vibration do to your ssd?” in *DAC*, 2019.
- [18] N. Mielke, T. Marquart, N. Wu, J. Kessenich, H. Belgal, E. Schares, F. Trivedi, E. Goodness, and L. Nevill, “Bit error rate in nand flash memories,” in *IEEE IRPS*, 2008.
- [19] W. Choi, M. Arjomand, M. Jung, and M. Kandemir, “Exploiting data longevity for enhancing the lifetime of flash-based storage class memory,” *SIGMETRICS*, 2017.
- [20] M. Xu, C. Tan, and M. Li, “Extended arrhenius law of time-to-breakdown of ultrathin gate oxides,” 2003.
- [21] S. Aritome, *NAND flash memory technologies*. Wiley, 2015.
- [22] A. E. Systems. (2021) Environmental Test Chamber LH Series. [Online]. Available: <https://www.associatedenvironmentalsystems.com/products/lh-1-5>
- [23] J. Axboe. (2021) Fio. [Online]. Available: https://fio.readthedocs.io/en/latest/fio_doc.html
- [24] J. Bhimani, A. Maruf, N. Mi, R. Pandurangan, and V. Balakrishnan, “Auto-tuning parameters for emerging multi-stream flash-based storage drives through new i/o pattern generations,” *IEEE TOC*, 2020.

MoKE: Modular Key-value Emulator for Realistic Studies on Emerging Storage Devices

Manoj Pravakar Saha
Florida International University
Miami, USA
msaha002@fiu.edu

Danlin Jia
Northeastern University
Boston, Massachusetts
jia.da@northeastern.edu

Janki Bhimani
Florida International University
Miami, Florida
jbhimani@fiu.edu

Ningfang Mi
Northeastern University
Boston, Massachusetts
ningfang@ece.neu.edu

Abstract—Key-value stores are widely used as building blocks in today’s IT infrastructure for managing and storing large amounts of data. Storage technologies are undergoing continuous innovations to accelerate KV workloads. However, designing high-performance KV or object storage devices is challenging and still needs more research to address the performance bottlenecks of the existing designs. There is a void for an inexpensive and extendable research platform that enables in-depth exploration of the index management components within the KV devices. To fill this void, we design Modular Key-value Emulator (MoKE). MoKE is a software emulator for fostering future full-stack software/hardware KV and object storage device research. MoKE is cheap (software-based emulator), usable with SNIA KV API (supports popular host-device interfaces), extendable (supports internal KV device research), and adaptable (QEMU-based).

Index Terms—Solid State Drive, Key-Value SSD, emulator

I. INTRODUCTION

The explosion of unstructured data has given rise to the Key-Value (KV) and object interfaces and unstructured data stores. However, conventional software KV/object databases confront computational and memory overheads on the operating system (OS) I/O stack. I/O requests need to go through multiple layers, and each layer adds syntax translation complexity and performance cost. Therefore, computational and memory overheads of software KV/object stores [6], [11] have intrigued researchers in industry and academia to explore KV/Object Drives over the years. The proposed designs include Kinetic Drive [8], KAML [4], KV-SSD [2], [6], [9], and KV CSD [12]. While each design is innovative, we have not seen wide-scale adoption of such devices. Despite the low adoption of such devices, KV- and Object-SSD retains their intrigue because of the popularity of the KV and object interfaces, which needs constant efforts from both academia and industry.

Research quests to design efficient KV and Object SSDs have been challenging and expensive. Existing research works have all been implemented either on real devices, accessible only to manufacturers or on costly FPGA-based device prototypes. On top of the direct cost of such devices, the learning curve of working on FPGA-based devices is also very high. It is not easy for most academic research labs to afford such expensive and complex platforms. More importantly, there is a void in terms of open-source software platforms that foster research on the impact of individual components within KV storage devices on the overall I/O stack. We need open-source

emulators/simulators that can enable research on the primary bottlenecks, such as host-device interface, application data index management, and integrated RAM caching, within KV-SSDs and similar devices [11].

To fill this void, we present Modular Key-value Emulator (MoKE), based on FEMU (a QEMU-based Flash Emulator [10]), to foster in-depth research on KV-SSDs. MoKE provides a complete overview of the impact of individual internal components on the overall I/O stack.

II. MOKE: MODULAR KEY-VALUE EMULATOR

MoKE is designed on top of FEMU [10], a popular block-SSD emulator. Like FEMU, MoKE offers a virtualized SSD interface running inside a virtual machine on top of QEMU [1]. In MoKE, we replace FEMU’s block-based host-device interface with NVMe KV interface, with support for the SNIA KV API [15]. We also replace FEMU’s data backend and FTL emulation to enable KV data management and delay emulation.

A. Host-Device Interface for KV Stack

The first step in designing an NVMe KV host-device interface is to implement the NVMe KV command set [3] into device drivers and implement KV request handlers inside the emulator. Since, MoKE is a research platform, we integrate MoKE with the more flexible OpenMPDK NVMe KV command set and device driver [14].

1) *KV Command Transfer*: **Problem**: KV command transfers are inherently different and expensive compared to their block device counterparts and can affect I/O performance of the device [7]. First, keys larger than 16 bytes cannot be passed on as part of a NVMe command and requires a separate DMA transfer using Physical Region Pages (PRP). Secondly, additional packing and unpacking of keys can lead to overhead in command processing. Thirdly, unlike block devices, KV commands cannot be coalesced together based on sequentially or key group. A real Samsung KV-SSD (not the KV emulator) [6] uses multiple request handlers running on different CPUs to offset these overheads. However, implementing emulation of such request-handling architectures would complicate the design and incur significant overhead. It is challenging to correctly emulate the command processing to match real KV-SSD’s command processing throughput.

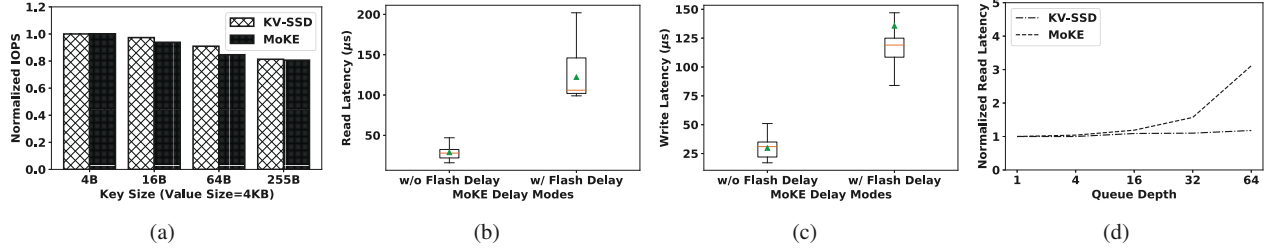


Fig. 1: (a) MoKE IOPS drop with increase in key size. (b) Read I/O latency in different modes. (c) Write I/O latency in different modes. (d) Impact of queue depth on read latency.

Our Solution: To achieve correct and efficient command processing in MoKE, we first implement separate DMA transfers for keys and values based on command parameters. We also introduce new data structures to identify and process KV commands and to manage the KV data inside the emulator efficiently. Approximately of 900 lines of codes (LOC) has been added in `nvme.h`, `nvme-io.c` and `bb.c` to support the feature of NVMe KV command transfer.

Results: We measure the correctness of the host-device interface by comparing MoKE’s throughput trends with Samsung KV-SSD for different key sizes. When the key sizes increase, the I/O throughput of KV-SSD is expected to decrease [13], due to additional DMA transfers. Figure 1a shows that MoKE can correctly emulate this behavior.

2) *KV API support:* **Problem:** Applications interact with KV-SSD using the standardized SNIA KV API. The KV API library transforms the API requests into NVMe KV commands. For SNIA KV API to work properly, a storage device/emulator needs to implement two features - key space management and KV I/O error handling. However, key space is an abstraction recognized by the KV API, not the NVMe layer. In addition, KV-specific I/O errors during a transaction also needs to be handled by the KV API layer. The exact method through which the API-level abstractions are recognized and processed inside the KV-SSD is not trivial. For example, the data structures that list the key spaces, some details such as the actual keys being used or the exact NVMe commands being used for management of key spaces are still implementation dependent. Hence, to enable key space management KV API support, we either need to reverse engineer these implementation details (i.e. keys used to store key space list and commands used to manage key spaces) and modify MoKE, or implement our own solution both in the KV API layer and in the emulator layer.

Our Solution: To enable KV API support in MoKE, we first analyze the chain of KV API requests, their transformation in the ADI layer, and the corresponding NVMe commands issued to the device, starting from the initialization of the device. We notice that the OpenMPDK KV library uses the three basic KV read, write, and delete commands to manage the device and key spaces instead of any admin or custom NVMe commands. The exact KV pairs that hold the key space list are pre-defined inside the KV API. We reverse engineer the implementation

details of these KV pairs from the OpenMPDK KV API library. We create and insert these KV pairs in MoKE during initialization using the reverse-engineered keys and values. We also add support for multiple key spaces, along with key space isolation, in MoKE. As a result of these changes, applications can create, use or delete key spaces using the KV API layer. Next, we add support for KV I/O error handling by adding KV command error codes inside the emulator and transferring the error information from emulator to KV API using KV command context data structure. The above changes enable both regular and benchmark applications to interact with MoKE through the SNIA KV API.

B. Delay Emulation

Problem: The latency of an I/O operation in KV-SSD is an accumulation of two main components - request handling delay and data access delay. FEMU already implements a basic delay model for data access based on single-register and uniform page latency. However, it lacks delay emulation for variable-length KV data and KV request handling overheads (e.g. key hashing delay, K2P table processing delay, DMA transfer delay). To support variable-length KV data, we need to adapt the data backend and the FTL.

On the other hand, if we try to emulate the exact delays caused by KV request handling overheads, it would require frequent calls to the system clock, and may negatively influence delay emulation.

Our Solution: We modified FEMU’s data backend and FTL to support variable-length KV data. Then, to tackle the problem of emulating KV request handling overheads, we consider a hybrid approach. We strive to emulate overheads incurred by different components in KV devices using different methods. First, we experiment with DMA transfer delays for large keys. Our experiments reveal that DMA transfer delays for command transfers are included in the latency inherently, because subsequent I/O commands are inserted into the NVMe submission queue at a slightly slower rate. Directly adding the DMA transfer delay to the I/O request submission time would result in counting the delay twice. Thus, we do not add any additional delay for the DMA transfer of large keys. Next, we measure the processing delay of different operations, such as key hashing and K2P table access. The average cumulative delay of these overheads for both read and write requests is around 4.2 and 4.4 microseconds, respectively, when the

99th percentile tail latency is ignored. When the tail latency is considered, the cumulative delay is about 4.8 and 16.2 microseconds for read and write requests. These measurements indicate that the delays added by different index operations are dependent on the type of requests (e.g., read or write or exist). Hence, we consolidate the delays incurred by different index operations and introduce four configurable variables T_{read} , T_{write} , T_{delete} , and T_{exist} to emulate the cumulative processing delays for read, write, delete and exist requests. These delays can be different for different index structures. Users can set these values through empirical studies on their own FTL implementation(s).

III. ACCURACY EVALUATION

We measure MoKE using the following configuration. We emulate a 64GB SSD (8 channels, 8 LUNs/channel, and 16KB flash page) where latency for program, read and erase operations are $660\mu s$, $45\mu s$, and $3.5ms$, respectively [5]. All experiments were performed on a machine with 2xIntel(R) Xeon(R) Silver 4208 CPU, 192GB DDR4 DRAM, with a 1TB SAS HDD as a backing store. OpenMPDK KVbench [14] is used to measure MoKE's performance (latency and throughput) because the microbenchmark tool is compatible with SNIA KV API and can generate realistic workloads [6], [13]. All experiments in this work use 10GB random read or write workloads (2.6M KV pairs with 16B keys and 4KB values), with queue depth of 1 for latency and 64 for throughput measurements similar to industry practice.

First, we observe the I/O latency with and without flash access delay emulation to verify if MoKE emulates delays (or no delays) correctly (Figure 1b and 1c). Average store and retrieve latency without delay emulation are $28.7\mu s$ and $29.5\mu s$, respectively. This delay is added partly by the host (KV driver, API, and KVbench itself) to process the request and partly by MoKE (as it cannot return immediately upon receiving the request even if the delay is set to 0). The unintentionally added processing delays by MoKE are small enough to be offset during actual I/O operations with (data placement and index) delay emulation. Second, we verify that MoKE's read latency can scale similar to Samsung KV-SSD, as we increase the queue depth [13]. MoKE can sustain read latency up to 16 queue depth, but suffers at very high queue depth due to the lack of large number of concurrent request managers, as seen in Samsung KV-SSD (Figure 1d). In summary, our experimental results show that MoKE is accurate enough to be used as a platform to trigger important emerging KV/object storage device-related research. We plan to continue enhancing MoKE to be more scalable and easy to use.

IV. CONCLUSIONS

MoKE is designed to fill the void of an inexpensive, extendable KV/object-SSD research platform. MoKE will likely foster thorough investigations related to the complex design choices within the algorithms performed and the data structures maintained inside the KV storage devices. Our

evaluation results show that MoKE accurately mimics the performance trends of a real Samsung KV-SSD. We expect that MoKE will speed up future KV-SSD research and also enable researchers to explore other emerging directions such as namespace isolation, KV interface-based Computational Storage Devices (CSD), and many more.

ACKNOWLEDGMENT

This work was partially supported by National Science Foundation Awards CNS-2008324 and CNS-2008072 and KV-SSD equipment grant and Funded Research Collaboration with Samsung MSL GRO (Global Research Outreach).

REFERENCES

- [1] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, page 41, USA, 2005. USENIX Association.
- [2] Tai Chang, Jen-Wei Hsieh, Tai-Chieh Chang, and Liang-Wei Lai. Emt: Elegantly measured tanner for key-value store on ssd. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(1):91–103, 2022.
- [3] NVM Express. NVMe Specification 2.0, 2022. <https://nvmexpress.org/developers/nvme-command-set-specifications/>.
- [4] Yanqin Jin, Hung-Wei Tseng, Yannis Papakonstantinou, and Steven Swanson. Kaml: A flexible, high-performance key-value ssd. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 373–384. IEEE, 2017.
- [5] Dongku Kang, Woopyo Jeong, Chulbum Kim, Doo-Hyun Kim, Yong Sung Cho, Kyung-Tae Kang, Jinho Ryu, Kyung-Min Kang, Sungeon Lee, Wandong Kim, Hanjun Lee, Jaedoeg Yu, Nayoung Choi, Dong-Su Jang, Jeong-Don Ihm, Doogon Kim, Young-Sun Min, Moo-Sung Kim, An-Soo Park, Jae-Ick Son, In-Mo Kim, Pansuk Kwak, Bong-Kil Jung, Doo-Sub Lee, Hyunggon Kim, Hyang-Ja Yang, Dae-Seok Byeon, Ki-Tae Park, Kye-Hyun Kyung, and Jeong-Hyuk Choi. 7.1 256gb 3b/cell v-nand flash memory with 48 stacked wl layers. In *2016 IEEE International Solid-State Circuits Conference (ISSCC)*, pages 130–131, 2016.
- [6] Yangwook Kang, Rekha Pitchumani, Pratik Mishra, Yang-suk Kee, Francisco Londono, Sangyoon Oh, Jongyeol Lee, and Daniel D. G. Lee. Towards building a high-performance, scale-in key-value storage system. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 144–154, New York, NY, USA, 2019. Association for Computing Machinery.
- [7] Sang-Hoon Kim, Jinhong Kim, Kisik Jeong, and Jin-Soo Kim. Transaction support using compound commands in key-value SSDs. In *Proceedings of the USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage)*, 2019.
- [8] KOSP. Kinetic Open Storage Project, 2015. <https://github.com/Kinetic>.
- [9] Changgyu Lee, Hyeongu Kang, Donggyu Park, Sungyong Park, Kim Youngjae, Jungki Noh, Woosuk Chung, and Kyoung Park. ilsm-ssd: An intelligent lsm-tree based key-value ssd for data analytics. pages 384–395, 10 2019.
- [10] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Björling, and Haryadi S Gunawi. The {CASE} of {FEMU}: Cheap, accurate, scalable and extensible flash emulator. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 83–90, 2018.
- [11] Pratik Mishra, Rekha Pitchumani, and Yang Suk Kee. Learnings from an under the hood analysis of an object storage node io stack. 2022.
- [12] Dan Robinson (The Register). SK hynix and Los Alamos Labs to demo key-value store accelerating SSD, 2022. https://www.theregister.com/2022/08/01/sk_hynix_lanl_kv_csd/.
- [13] Manoj P. Saha, Adnan Maruf, Bryan S. Kim, and Janki Bhimani. Kv-ssd: What is it good for? In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1105–1110, 2021.
- [14] Samsung. OpenMPDK KVSSD, 2018. <https://github.com/OpenMPDK/KVSSD/>.
- [15] SNIA. Key value storage API specification, 2020. <https://www.snia.org/keyvalue>.



RHIK: Re-configurable Hash-based Indexing for KVSSD

Manoj P. Saha
Florida International University
Miami, Florida, USA
msaha002@fiu.edu

Haryadi S. Gunawi
University of Chicago
Chicago, Illinois, USA
haryadi@cs.uchicago.edu

Bryan S. Kim
Syracuse University
Syracuse, New York, USA
bkim01@syr.edu

Janki Bhimani
Florida International University
Miami, Florida, USA
jkbhimani@fiu.edu

ABSTRACT

Key-Value Solid State Drive (KVSSD), a key addressable SSD technology, promises to simplify storage management for unstructured data and improve system performance with minimal host-side intervention. However, we find that the current state-of-the-art KVSSD exhibits indexing peculiarities that limit their widespread adoption. Through experiments, we observe that the performance degrades as more data are stored, and the KVSSD can only store a limited number of key-value pairs even though the amount of data stored on the device is significantly lower than its capacity. We introduce RHIK, a reconfigurable hash-based indexing for KVSSD, for high performance and high occupancy. We implement our proposed indexing scheme on the open-source KVSSD emulator that is validated against a real KVSSD, and demonstrate its effectiveness using real workload traces and synthetic microbenchmarks.

CCS CONCEPTS

• Information systems → Flash memory.

KEYWORDS

Key-Value SSD, data storage, KV indexing

ACM Reference Format:

Manoj P. Saha, Bryan S. Kim, Haryadi S. Gunawi, and Janki Bhimani. 2023. RHIK: Re-configurable Hash-based Indexing for KVSSD. In *Proceedings of the 32nd International Symposium on High-Performance Parallel and Distributed Computing (HPDC '23)*, June 16–23, 2023, Orlando, FL, USA. ACM, New York, NY, USA, 2 pages. <https://doi.org/10.1145/3588195.3595945>

1 MOTIVATION

Performance drops as index size increases: The hash-based index in current KVSSD increases the tail latency and hampers performance as the size of the index grows [2] because the index becomes too large to fit in the SSD DRAM and optimizations from block-SSD cannot be applied directly. This behavior is visible in Fig. 1. Starting in Fig. 1a, KVSSD can sustain I/O performance at a smaller index size, but as the index grows from 2 million vs. 116 million vs. 1760 million vs. 3100 million (patterned vertical lines as shown in 1b) KVSSD's performance collapses.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
HPDC '23, June 16–23, 2023, Orlando, FL, USA
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0155-9/23/06.
<https://doi.org/10.1145/3588195.3595945>

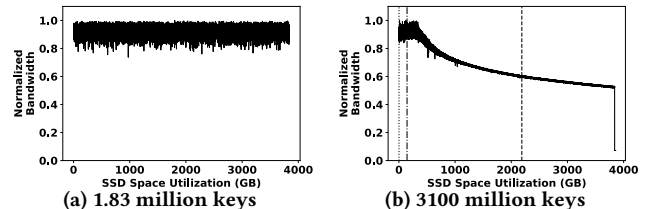


Figure 1: Write bandwidth drops with increasing index size

Index supports only a limited number of keys: We observe that KVSSD also supports only a limited number of keys compared to its capacity. Even though Samsung KVSSD supports an unfathomably large keyspace (consisting of $2^4 + 2^5 + \dots + 2^{254} + 2^{255}$ keys), our experiments reveal that a 3.84TB PM983 KVSSD can store a maximum of approximately 3.1 billion KV pairs (with value lengths of 1KB or lower). Thus, we next analyze the number of keys required by most of the existing KV stores. The index in a 4TB KVSSD would need to store 34 million-2.7 billion keys for a typical Baidu Atlas KV workload [3] and 24-744 billion keys for a typical Facebook Memcached workload [1].

2 RHIK ARCHITECTURE

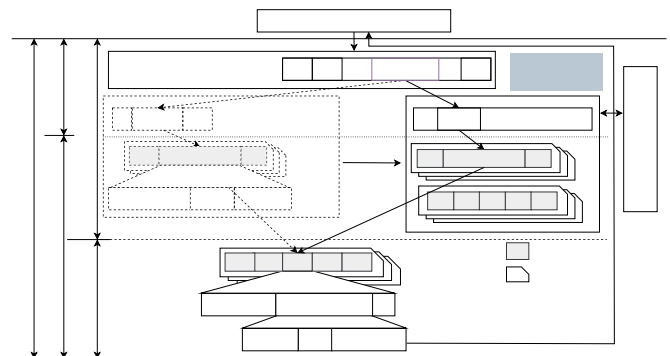


Figure 2: Re-configurable indexing

We introduce RHIK, a reconfigurable hash-based indexing for KVSSD, and to the best of our knowledge the first KVSSD design that guarantees maximum of one flash read for index access and supports high index occupancy. The first layer in RHIK works like a directory, containing D entries, and accessed from SSD DRAM (Fig. 2). At the same time, a periodically updated persistent copy of these D entries resides on flash. The second layer, or record layer, holds fixed-size independent hash tables that store the metadata of corresponding KV pairs and is served from flash unless available in the integrated RAM. With this 2-level design, we are able to

guarantee maximum of one flash read per index access, something that is almost impossible to achieve in the current state-of-the-art multi-level hash index or LSM-tree design. In addition, RHIK has the three other important features - efficient collision management, conservative index initialization and resizing, and low-overhead membership checking.

Efficient Collision Management: RHIK employs Hopscotch hashing to ensure better collision handling and higher translation page (i.e. NAND flash pages storing the index records) occupancy (Fig. 2). While RHIK’s collision management scheme needs marginally more compute than existing collision handling mechanism, it reduces the number of translation pages required for storing the index.

Conservative Index Initialization and Resizing: RHIK is initialized conservatively to reduce space wastage related to provisioning a large under-utilized hash index, and resizing triggered only when total occupancy reaches a pre-defined threshold (Fig. 2).

Low-overhead Membership Checking: RHIK reuses key signatures for membership checking, instead of traditional Bloom filter-based approaches (Fig. 2). This reduces the memory footprint and management complexity of the index.

RHIK on KV Emulator We develop an advanced version of the KV Emulator by extending OpenMDK KV Emulator. The new KV Emulator imitates the fundamental hardware primitives of an SSD, such as flash blocks, pages, etc. We implement RHIK in the new KV Emulator. We implement separate indexing and data layout schemes.

3 EVALUATION

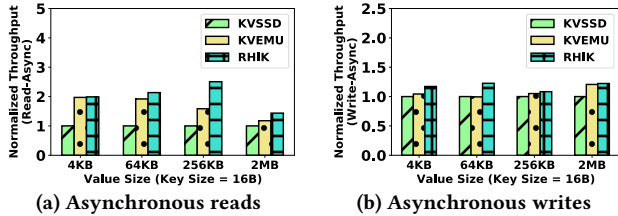


Figure 3: I/O performance comparison

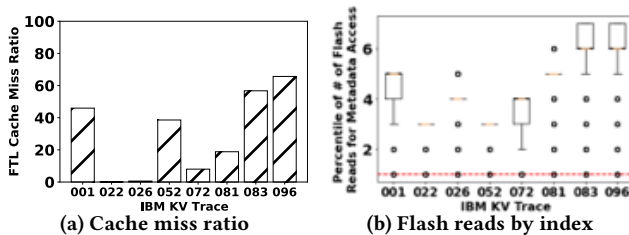


Figure 4: RHIK incurs only one flash read per I/O request

First, we evaluate RHIK’s performance using multiple sequential workloads with variable key-value sizes (Fig. 3). Since the KV Emulator is limited by host DRAM capacity, we evaluate RHIK for 100 million keys on a 64GB emulated drive. When we compare asynchronous write performance, for almost all value sizes, RHIK achieves higher throughput. For asynchronous read requests, we observe that RHIK is able to perform better with large value sizes during read. We observe that the performance trends of the normalized throughput of the OpenMPDK KV Emulator differs from KVSSD. We believe that this difference in the performance trends

may be due to the IOPS model used by the OpenMPDK KV Emulator. Since, our extended KV Emulator also uses the same IOPS model, we measure RHIK’s performance improvements relative to OpenMPDK KV Emulator performance. In addition, KVSSD and RHIK both maintain un-ordered index with KV pairs written in a log-like manner on flash, and KV operations are largely dominated by key handling operations, the performance of sequential workloads are not significantly different from Uniform or Zipfian workloads in KVSSD for most cases.

As we show in the motivation, the KVSSD performance drops when multi-level hash index size increases due to multiple flash page accesses needed to read the portion of the index that doesn’t fit within the SSD DRAM. Fig. 4 shows that the number of flash page accesses using RHIK is significantly less compared to that of the multi-level hash index. Particularly, here we limit the SSD DRAM cache budget to 10MB (i.e. for a 10GB SSD) for both and we replay the real-world IBM Cloud Object Store KV traces. Out of the eight clusters, four clusters (i.e. 022, 026, 052, and 072) index fits within the SSD cache, so no difference between RHIK and KVSSD is observed. However, for the rest of the clusters that have an index size of more than 10MB, RHIK incurs only one flash access but multi-level hash within KVSSD may incur 3 to 7 accesses. Hence, the performance using RHIK will not drop upon increasing index size.

4 CONCLUSION

KVSSDs established the performance benefits of removing the I/O stack redundancies for KV workloads and finally became a commodity product in 2018, but it is considerably more expensive than block SSDs. However, as we show unoptimized index management can make the KVSSD underutilized and lower its performance to be worse than the block SSDs. In this work, we show novel index management along with many other features needed to achieve sustained performance from KVSSDs even when the index size increases.

ACKNOWLEDGMENTS

This work was partially supported by the National Science Foundation (NSF) Awards CNS-2008324 and CNS-2122987, and KV-SSD equipment grant and funded research collaboration with Samsung MS� GRO(Global Research Outreach).

REFERENCES

- [1] Berk Atikoglu, Yuehai Xu, Eitan Frachtenberg, Song Jiang, and Mike Paleczny. 2012. Workload Analysis of a Large-Scale Key-Value Store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems* (London, England, UK) (SIGMETRICS '12). Association for Computing Machinery, New York, NY, USA, 53–64. <https://doi.org/10.1145/2254756.2254766>
- [2] Junsu Im, Jinwook Bae, Chanwoo Chung, Arvind, and Sungjin Lee. 2020. PinK: High-speed In-storage Key-value Store with Bounded Tails. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 173–187. <https://www.usenix.org/conference/atc20/presentation/im>
- [3] C. Lai, S. Jiang, L. Yang, S. Lin, G. Sun, Z. Hou, C. Cui, and J. Cong. 2015. Atlas: Baidu’s key-value storage system for cloud data. In *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*. 1–14.