IEEE micro
The magazine for chip and silicon systems designers

VOLUME 45, NUMBER 6 NOVEMBER/DECEMBER 2025

Cache Coherent Interconnects and Resource Disaggregation Techniques

IEEE
IEEE COMPUTER SOCIETY
www.computer.org/micro

IEEE

IEEE COMPUTER SOCIETY

IEEE Computer Society

Grants for EMERGING TECHNOLOGY ACTIVITIES

MAKE AN IMPACT | CREATE SOLUTIONS

Are you connecting the computing community with emerging technologies? Help advance emerging tech to create solutions for the betterment of humanity.

Every year, we give up to **$50,000** in funding per project for these efforts.

Learn more at
computer.org/communities/emerging-technology-fund

bitly

IEEE micro

www.computer.org/micro

COMPUTER SOCIETY OFFICE
IEEE MICRO
c/o IEEE Computer Society
10662 Los Vaqueros Circle, Los Alamitos, CA 90720 USA
Phone +1 714 821 8380; Fax +1 714 821 4010
Website: www.computer.org/micro

IEEE micro
The magazine for chip and silicon systems designers

6 GUEST EDITORS' INTRODUCTION
Special Issue on Cache Coherent Interconnects and Resource Disaggregation Techniques
Wonil Choi and Jie Zhang

NOVEMBER/DECEMBER 2025

Theme Articles

8 Efficient Disaggregated Cloud Storage for Cold Videos With Neural Enhancement
Jinyeong Lim, Juncheol Ye, Jaehong Kim, Hwijoon Lim, Hyunho Yeo, Junhyeok Jang, Myoungsoo Jung, and Dongsu Han

16 Improving Remote File Access in Distributed Object Stores by Decoupling Metadata and Data Paths Using NVMe-oF
Daegyu Han, Sungho Moon, Kyeungpyo Kim, Sung-Soon Park, and Beomseok Nam

24 Containerized In-Storage Processing and Computing-Enabled Solid-State Drive Disaggregation
Miryeong Kwon, Donghyun Gouk, Eunjee Na, Jiseon Kim, Junhee Kim, Hyein Woo, Eojin Ryu, Hyunkyu Choi, Jinwoo Baek, Hanyeoreum Bae, Mahmut Kandemir, and Myoungsoo Jung

Theme Articles Continued

36 Compute Express Link Topology-Aware and Expander-Driven Prefetching: Unlocking Solid-State Drive Performance
Dongsuk Oh, Miryeong Kwon, Jiseon Kim, Eunjee Na, Junseok Moon, Hyunkyu Choi, Seonghyeon Jang, Hanjin Choi, Hongjoo Jung, Sangwon Lee, and Myoungsoo Jung

46 From Block to Byte: Transforming PCIe Solid-State Devices With Compute Express Link Memory Protocol and Instruction Annotation
Miryeong Kwon, Donghyun Gouk, Junhyeok Jang, Jinwoo Baek, Hyunwoo You, Sangyoon Ji, Hongjoo Jung, Junseok Moon, Seungkwan Kang, Seungjun Lee, and Myoungsoo Jung

56 Ginkgo: A Learned-Index Enhanced Tiered Memory System
Xiran Yang, Yifei Yu, Chuandong Li, Jianqiang Zeng, Ke Zhou, Diyu Zhou, Xiaolin Wang, Zhenlin Wang, and Yingwei Luo

65 Tiered Cache-Sharing Service for Virtual Machine Images Based on Memory Pool
Zhihao Zhang, Weinan Liu, Zhenlong Song, Xinbiao Gan, Yue Yu, and Yiming Zhang

73 Improving SQL Join Algorithms for Distributed Systems: A Case Study of Compute Express Link-Based Multihost Shared Memory
JaeYung Jun, HyunWoong Ahn, Joohee Lee, Jungmin Choi, Byungil Koh, Donguk Moon, and Hoshik Kim

82 Maximizing Interconnect Bandwidth and Efficiency in Nonvolatile Memory, Express-Based Key-Value Solid-State Devices With Fine-Grained Value Transfer
Junhyeok Park, Chang-Gyu Lee, Soon Hwang, Seung-Jun Cha, Woosuk Chung, and Youngjae Kim

91 FS^2: A Fast, Scalable, and Flexible Switching System for Emerging Interconnects
Heetaek Jeong, Kanghyun Choi, Hamin Jang, Dongup Kwon, Eunjin Baek, Pyeongsu Park, and Jangwoo Kim

100 Compute-Enabled CXL Memory Expansion for Efficient Retrieval Augmented Generation
Derrick Quinn, Neel Patel, and Mohammad Alian

108 CXL–GPU: Pushing GPU Memory Boundaries with the Integration of CXL Technologies
Donghyun Gouk, Seungkwan Kang, Seungjun Lee, Jiseon Kim, Kyungkuk Nam, Eojin Ryu, Sangwon Lee, Dongpyung Kim, Junhyeok Jang, Hanyeoreum Bae, and Myoungsoo Jung

Cover image credit: ©shutterstock/Kseniavasil

Columns and Departments

4 **From the Editor-in-Chief**
A 33-GW Pilgrimage to the Promised Land of Agent Workforce
Hsien-Hsin S. Lee

119 **Micro Law**
A Review of *Wisconsin Alumni Research Foundation v. Apple*—Part VII
Joshua J. Yi

124 **Micro Economics**
Private Returns on Technology Adoption
Shane Greenstein

Also in This Issue

1 Masthead
118 IEEE Computer Society Information

www.computer.org/micro

ISSN: 0272-1732

FROM THE EDITOR-IN-CHIEF

A 33-GW Pilgrimage to the Promised Land of Agent Workforce

Hsien-Hsin S. Lee, *Intel Corporation, Boxborough, MA, 01719, USA*

Imagine a near-future workplace where employers no longer need to hire humans for day-to-day business operations. Instead, artificial intelligence (AI) agents—an entirely new species in the modern (or virtual) office—carry out most of the tasks that real people were once paid to perform. These agents never tire or sleep, work around the clock without weekends or holidays, and operate with almost no mistakes. They do not negotiate for better benefits, organize unions to demand pay raises, or go on strike to disrupt productivity. What sounds like the opening chapter of a science fiction novel is now inching closer to reality. This new working norm, I believe, reflects the workplace of a not-so-distant future envisioned by many frontier thought leaders in AI.

Over the past couple of months, Sam Altman, CEO of OpenAI, has been making news by steadily forging partnerships, one after another, across the leading AI infrastructure supply chain, including vendors such as Oracle, Nvidia, AMD, Broadcom, and others. Thus far, OpenAI has announced future access to at least 33 GW of GPU capacity, an astonishing amount of computing power secured to support the AI services that OpenAI believes it will need in the coming years. Critics remain skeptical of these widely publicized deals. OpenAI is projected to generate roughly $20 billion in revenue this year with rosy forecasts that will reach hundreds of billions by 2030.[1] However, how, then, can the company afford to commit to deals whose cumulative value is well more than $1 trillion at today's prices?

Several of these agreements appear to involve circular revenue flows. Nvidia, for instance, has pledged $100 billion to fund AI infrastructure for OpenAI—capital that OpenAI is expected to spend on purchasing Nvidia's GPUs in return. As such, this circular capital flow artificially inflates Nvidia's revenue numbers, raising big questions of its legitimacy. AMD also announced a similar partnership shortly thereafter. Instead of a direct investment, AMD granted OpenAI stock warrants of up to 160 million shares, which will vest over time contingent on specific milestones to be met by OpenAI, including purchasing and deploying 6 GW of AMD GPU capacity in OpenAI's data center fleet in coming years. Yet, unlike the alleged circular economics of the Nvidia arrangement, this deal, I think, provides AMD with a strategic advantage: it gives AMD a great opportunity to have OpenAI, the most powerful AI private company today, validate and certify its GPU technologies. This is particularly meaningful for its ROCm (Radeon Open Compute platform) software framework, which has been facing an uphill battle against the formidable dominance of Nvidia's CUDA. Should OpenAI fulfill its terms, AMD will gain credibility via serving OpenAI's customers and become a viable alternative to those companies that are seeking a second GPU supplier to run AI on-premises or provide AI services. It could rapidly catalyze AMD's expansion of its GPU market share in the AI data center segment.

But, returning to the 33-GW GPU capacity; if Sam Altman is thinking about what I suspect he is, then such aggressive stockpiling of computing power may not seem completely unrealistic. A new wave of compute-intensive digital behavior is already emerging. For instance, Meta recently launched Meta Vibes through the Meta AI app, a platform where any user can create and directly share entirely AI-generated short videos. This new social network has gone viral, functioning as an AI-native counterpart to Instagram Reels and TikTok. Anyone with a mobile device can easily consume significant GPU cycles to generate media content using AI. More consequentially, just 10 months after my discussion on the rise of AI agents,[2] autonomous agentic AI systems have gained more traction in practical, commercial applications. In the coming era, companies are likely to hire armies of AI agents from providers like OpenAI to replace large portions of their human workforce. These AI agents will be hired and deployed on demand, work 24/7, and require none of the fringe costs for perks, health insurance, and retirement plans. In other words, the

Digital Object Identifier 10.1109/MM.2025.3637948
Date of current version 24 December 2025.

APPENDIX: RELATED ARTICLES

A1. W. Choi and J. Zhang, "Special Issue on Cache Coherent Interconnects and Resource Disaggregation Techniques," *IEEE Micro*, vol. 45, no. 6, pp. 6–7, Nov./Dec. 2025, doi: 10.1109/MM.2025.3627696.

A2. J. J. Yi, "A review of *Wisconsin Alumni Research Foundation v. Apple*—Part VII," *IEEE Micro*, vol. 45, no. 6, pp. 119–123, Nov./Dec. 2025, doi: 10.1109/MM.2025.3638484.

A3. S. Greenstein, "Private returns on technology adoption," *IEEE Micro*, vol. 45, no. 6, pp. 124–126, Nov./Dec. 2025, doi: 10.1109/MM.2025.3615287.

agent service companies could effectively become the "boss" of most workers at companies across the globe. The computational demand required to power such AI workforces, token throughput, task processing, problem solving, continuous workloads, and more would be unimaginably enormous. In that light, even the 33 GW that Sam Altman has secured may one day seem small.

The initial discussion of the topic of this special issue began in early 2024 from my conversation with Prof. Myoungsoo Jung from Korea Advanced Institute of Science and Technology (KAIST). I knew Prof. Jung back in 2009 when he was working at Samsung Electronics while pursuing a graduate degree jointly bestowed by Georgia Tech and Korea University. He has been incredibly successful in multiple computer architecture areas including storage, nonvolatile memory, interconnect technologies, and processing-in-memory during his tenure as a faculty member at the University of Texas at Dallas, Yonsei University, and now an Endowed Chair Professor at KAIST. He is also a key contributing member to the Compute Express Link (CXL) and Ultra Accelerator Link Consortiums. Our conversation back then was centered around the CXL Intellectual Property (IP) and solutions under development by Panmnesia, a start-up company that he started and serves as the CEO. We envisioned that CXL would be a critical enabling standard to enable composable data center architecture and address the huge memory capacity demand by modern, fast-evolving AI applications. Therefore, we reached the decision to solicit recent works for a Special Issue on Cache-Coherent Interconnects and Resource Disaggregation Techniques and appointed Prof. Wonil Choi from Hanyang University and Prof. Jie Zhang from Peking University to serve as guest co-editors. I truly appreciate Prof. Choi and Prof. Zhang for their dedication and diligence to select the 12 outstanding articles featured in this issue. These works were contributed by authors from both academia and industry. For more details, please read the guest editors' introductory message,[A1] which categorizes these works into three main areas: disaggregated storage, disaggregated memory, and interconnect.

As usual, in Part VII of the *Wisconsin Alumni Research Foundation (WARF) v. Apple* series for the Micro Law column,[A2] Dr. Joshua Yi focuses on two critical discovery disputes in the litigation. These disputes illustrate the broader challenges that courts face when balancing meaningful discovery with highly complex and rapidly evolving CPU products. In the Micro Economics column,[A3] Prof. Shane Greenstein reviews the lessons learned from previous successful adoptions of consumer computer technologies (CCTs), including mobile devices and the Internet and the related incremental and novel co-invention to utilize CCTs in business and create values. The article speculates that the same learning will be repeated in the new AI era.

I hope that you enjoy the final issue of 2025. Thank you again for your continued support of *IEEE Micro*. We wish you all a very Merry Christmas, and see you in 2026.

REFERENCES

1. A. Capoot, "Sam Altman says OpenAI will top $20 billion in annualized revenue this year, hundreds of billions by 2030," *CNBC*, Nov. 6, 2025. [Online]. Available: https://www.cnbc.com/2025/11/06/sam-altman-says-openai-will-top-20-billion-annual-revenue-this-year.html
2. H.-H. S. Lee, "Rise of the agentic AI workforce," *IEEE Micro*, vol. 45, no. 1, pp. 4–5, Jan./Feb. 2025, doi: 10.1109/MM.2025.3535912.

HSIEN-HSIN S. LEE is an Intel Fellow at Intel Corporation, Boxborough, MA, 01719, USA. Contact him at lee.sean@gmail.com.

Special Issue on Cache Coherent Interconnects and Resource Disaggregation Techniques

Wonil Choi, *Hanyang University, Ansan, 15588, Republic of Korea*

Jie Zhang, *Peking University, Beijing, 100871, China*

In an era of explosive data growth fueled by workloads, such as machine learning training and large-scale data analytics, the scaling of modern data centers and high-performance computing environments is severely constrained by the traditional monolithic architectures that tightly couple compute, memory, and storage resources. Such tightly integrated systems continue to struggle with issues, including resource overprovisioning, underutilization, and the memory capacity wall, highlighting the urgent need for more flexible and scalable architectural designs.

Resource disaggregation emerges as a compelling paradigm, promising to break down such monolithic system architectures into pools of shared, distributed resources. However, the transition to disaggregated resources introduces its own set of challenges, including the need for significant code refactoring, potential performance penalties, substantial new hardware investments, increased complexity in system maintenance, and security concerns.

These challenges have driven researchers to explore innovative solutions for advancing resource disaggregation. A prominent direction is the development of cache-coherent interconnect technologies, such as Intel's Ultra Path Interconnect and QuickPath Interconnect, AMD's Infinity Fabric, and Compute Express Link (CXL), which provide a strong foundation for disaggregated architectures. By enabling efficient access to remote memory through cache coherence with minimal latency and overhead, these interconnects significantly enhance the feasibility and performance potential of resource disaggregation.

Amid this evolving landscape, this special issue on cache-coherent interconnects and resource disaggregation techniques highlights recent advances and emerging developments in the field. It features 12 articles covering a wide range of topics, from distributed memory and storage to interconnect technologies.

Digital Object Identifier 10.1109/MM.2025.3627696
Date of current version 24 December 2025.

DISAGGREGATED STORAGE

The first four articles explore disaggregated storage in various contexts. Lim et al.[A1] propose a solution for managing cold videos in disaggregated cloud storage, while Han et al.[A2] introduce a novel storage management framework that optimizes remote access in disaggregated storage nodes. Kwon et al.[A3] present an approach to disaggregate solid-state drive (SSD) resources to support multiple workloads simultaneously. Finally, Oh et al.[A4] enhance the performance of CXL storage through prefetching strategies.

DISAGGREGATED MEMORY

The next four articles address the challenges and offer solutions associated with integrating separate memory devices. Focusing on CXL memory, Kwon et al.[A5] develop and use a prototype to evaluate the feasibility of employing a block storage device as a CXL device with byte addressability. Yang et al.[A6] propose an approach to efficiently manage tiered memory systems using a learned index. Meanwhile, Zhang et al.[A7] focus on a cloud memory pool with disaggregated memory devices and present a method for preventing page duplication. Finally, Jun et al.[A8] explore disaggregated memory architectures optimized for executing structured query language (SQL) join operations.

INTERCONNECT

The last four articles target the interconnects of disaggregated devices. Park et al.[A9] propose an approach to improve interconnect bandwidth utilization in the context of disaggregated key-value SSDs. In contrast, Jeong et al.[A10] introduce a switching system designed to overcome topological limitations when integrating a large number of devices in a scalable manner. Quinn et al.[A11]

APPENDIX: RELATED ARTICLES

A1. J. Lim et al., "Efficient disaggregated cloud storage for cold videos with neural enhancement," *IEEE Micro*, vol. 45, no. 6, pp. 8–15, Nov./Dec. 2025, doi: 10.1109/MM.2025.3562625.

A2. D. Han, S. Moon, K. Kim, S.-S. Park, and B. Nam, "Improving remote file access in distributed object stores by decoupling metadata and data paths using NVMe-oF," *IEEE Micro*, vol. 45, no. 6, pp. 16–23, Nov./Dec. 2025, doi: 10.1109/MM.2025.3564477.

A3. M. Kwon et al., "Containerized in-storage processing and computing-enabled solid-state drive disaggregation," *IEEE Micro*, vol. 45, no. 6, pp. 24–35, Nov./Dec. 2025, doi: 10.1109/MM.2025.3574261.

A4. D. Oh et al., "Compute express link topology-aware and expander-driven prefetching: Unlocking solid-state drive performance," *IEEE Micro*, vol. 45, no. 6, pp. 36–45, Nov./Dec. 2025, doi: 10.1109/MM.2025.3574293.

A5. M. Kwon et al., "From block to byte: Transforming PCIe solid-state devices with compute express link memory protocol and instruction annotation," *IEEE Micro*, vol. 45, no. 6, pp. 46–55, Nov./Dec. 2025, doi: 10.1109/MM.2025.3581448.

A6. X. Yang et al., "Ginkgo: A learned-index enhanced tiered memory system," *IEEE Micro*, vol. 45, no. 6, pp. 56–64, Nov./Dec. 2025, doi: 10.1109/MM.2025.3564395.

A7. Z. Zhang, W. Liu, Z. Song, X. Gan, Y. Yu, and Y. Zhang, "Tiered cache-sharing service for virtual machine images based on memory pool," *IEEE Micro*, vol. 45, no. 6, pp. 65–72, Nov./Dec. 2025, doi: 10.1109/MM.2025.3574139.

A8. J. Jun, H. Ahn, J. Lee, J. Choi, B. Koh, and D. Moon, "Improving SQL join algorithms for distributed systems: A case study of compute express link-based multihost shared memory," *IEEE Micro*, vol. 45, no. 6, pp. 73–81, Nov./Dec. 2025, doi: 10.1109/MM.2025.3574357.

A9. J. Park, C.-G. Lee, S. Hwang, S.-J. Cha, W. Chung, and Y. Kim, "Maximizing interconnect bandwidth and efficiency in nonvolatile memory, express-based key-value solid-state devices with fine-grained value transfer," *IEEE Micro*, vol. 45, no. 6, pp. 82–90, Nov./Dec. 2025, doi: 10.1109/MM.2025.3572475.

A10. H. Jeong et al., "FS^2: A fast, scalable, and flexible switching system for emerging interconnects," *IEEE Micro*, vol. 45, no. 6, pp. 91–99, Nov./Dec. 2025, doi: 10.1109/MM.2025.3574732.

A11. D. Quinn, N. Patel, and M. Alian, "Compute-enabled CXL memory expansion for efficient retrieval augmented generation," *IEEE Micro*, vol. 45, no. 6, pp. 100–107, Nov./Dec. 2025, doi: 10.1109/MM.2025.3575280.

A12. D. Gouk et al., "CXL-GPU: Pushing GPU memory boundaries with the integration of CXL technologies," *IEEE Micro*, vol. 45, no. 6, pp. 108–117, Nov./Dec. 2025, doi: 10.1109/MM.2025.3582433.

present a CXL-based device prototype tailored for executing retrieval-augmented generation workloads. Finally, Gouk et al.[A12] enable CXL interconnects for GPU devices, which can expand GPU memory capacity.

This special issue aims to present recent advances and emerging developments in the paradigm of resource disaggregation. As this paradigm continues to evolve, encompassing a wide range of target contexts, applications, hardware, and system architectures, we hope that these articles will inspire further research and innovation in this rapidly advancing field. We express our sincere gratitude to all the authors who contributed to this special issue, to the reviewers for their valuable insights and constructive feedback, and to the editor-in-chief, Hsien-Hsin (Sean) Lee, as well as the *IEEE Micro* staff, for their support in bringing this issue to fruition.

WONIL CHOI is with Hanyang University, Ansan, 15588, Republic of Korea. Contact him at wonilchoi@hanyang.ac.kr.

JIE ZHANG is with Peking University, Beijing, 100871, China. Contact him at jiez@pku.edu.cn.

Efficient Disaggregated Cloud Storage for Cold Videos With Neural Enhancement

Jinyeong Lim, Juncheol Ye, Jaehong Kim, Hwijoon Lim, Hyunho Yeo, and Junhyeok Jang, *KAIST, Daejeon, 34141, South Korea*

Myoungsoo Jung, *KAIST and Panmnesia, Inc., Daejeon, 34141, South Korea*

Dongsu Han, *KAIST, Daejeon, 34141, South Korea*

The rapid growth of video-sharing platforms has driven immense storage demands, with disaggregated cloud storage emerging as a scalable and reliable solution. However, the proportional cost of cloud storage relative to capacity and duration limits the cost-efficiency for managing large-scale video data. This is particularly critical for cold videos, which constitute the majority of video data but are accessed infrequently. To address this challenge, this article proposes neural cloud storage (NCS), leveraging content-aware super-resolution powered by deep neural networks. By reducing the resolution of cold videos, NCS decreases file sizes while preserving perceptual quality. optimizing the cost tradeoffs in multitiered disaggregated storage. This approach extends the cost-efficiency benefits to a greater range of cold videos and achieves up to a 21.2% reduction in total cost of ownership, providing a scalable, cost-effective solution for video storage.

Video sharing services are playing a pivotal role in today's entertainment, social networks, and business. While videos contain diverse information such as visual and audial contents, it requires significantly large storage capacity to accommodate such information. For instance, YouTube needs 2.64 TB of additional storage capacity every minute, handling 500 hours of video uploads.[1] Similarly, Netflix manages 2 PB of data for its video service.[2]

In response to such increasing storage capacity demand, cloud storage has emerged as a scalable and reliable solution. Based on disaggregated storage systems,[3] it offers storage capacity as much as users need by connecting multiple storage resources through high-bandwidth networks such as RoCE or Infiniband. In addition, it offers reliable services by redundantly storing the videos across geographically distributed locations.

Even though the disaggregated cloud storage can scale to accommodate petabytes of videos, cost-efficient management of such data poses another challenge to video sharing services. Since cloud service providers often charge per-gigabyte fees for user's data, the cost of storing the petabyte-scale videos is significant and continues to grow as new videos are uploaded. To reduce the storage cost, one might think of leveraging a highly skewed access frequency (popularity) of videos to establish a multitiered storage system. Specifically, a small percentage of videos account for the vast majority of views, while cold videos with fewer views occupy most of the storage capacity. For example, the most popular 3.67% of videos on YouTube account for 93.61% of total views while 65.44% of videos are considered cold, having 100 or fewer views.[4] By managing such cold videos in cheaper storage tiers, we can cut off the storage cost without significantly harming the overall performance.

However, existing multitiered cloud storage services can reduce storage costs only for a limited number of

Digital Object Identifier 10.1109/MM.2025.3562625
Date of publication 21 April 2025; date of current version 24 December 2025.

low-access videos. For example, AWS S3's "Infrequent" tier cuts storage costs by nearly half (US$0.021 → US$0.0125/GB*month) compared to the "Frequent" tier but imposes a significant retrieval cost of US$0.01/GB per access. As a result, only videos with an average of 0.85 views or fewer views per month can benefit from such services. Based on our analysis of historical YouTube view data, as shown in Figure 1, only half of all videos could achieve storage cost savings.

To address this issue, this article proposes a solution by leveraging neural enhancement, specifically content-aware super-resolution (SR), to improve the cost efficiency of frozen video cloud storage without compromising video quality. Although additional computational cost is incurred during inference, this approach is suitable for frozen videos with low access frequency, typically accessed once per month. Based on this, we propose a prototype system called neural cloud storage (NCS), offering the following benefits: First, it reduces storage costs by lowering the resolution of frozen videos and optimizes the tradeoff between storage and retrieval costs, thereby reducing the total cost of ownership (TCO). Second, by reducing TCO, it extends the range of frozen videos benefiting from multitier services from 50% to 78%, supporting videos accessed up to 91 times annually, as illustrated in Figure 1. Finally, NCS has the potential to reduce TCO by 21.2% compared to storing cold videos on the cheapest AWS storage.

BACKGROUND

Disaggregated Cloud Storage

Figure 2 shows the typical architecture of disaggregated storage in modern cloud environments. At a high level, disaggregated storage comprises multiple storage nodes connected to compute nodes via a high-bandwidth network fabric. The storage node employs an array of storage devices [e.g., solid-state drives (SSDs), hard disks] which can be accessed through either a storage protocol such as NVMe-over-Fabric or a distributed file system such as Lustre. Compute nodes can secure the storage capacity as much as users need (e.g., for handling video uploads) by connecting the storage node as needed.

In such a system, there are two types of costs that contribute to the TCO for storage: 1) storage cost and 2) retrieval cost. Storage cost refers to the cost of storing video data, while retrieval cost refers to the cost of retrieving the video. The costs are calculated by considering that storing and retrieving consume power. In addition, it also considers that storing and retrieving consume the lifetime of the storage device. For example, in an SSD, storing data directly consumes the device's lifetime because the underlying media have a limited number of write/erase cycles they can endure. It also requires internal tasks to maintain the data without bit errors, consuming both the lifetime and power.

To this end, the TCO of the cloud video storage system can be formulated as follows:

$$\mathrm{TCO}(V_i) = [\text{Storage cost}(V_i) \times time_i + \text{Retrieval cost}(V_i) \times access_i]$$

$$\text{TCO of video cloud storage} = \sum_{i=1}^{N} \mathrm{TCO}(V_i)$$

where the unit of Storage cost(·) is $/time; while the unit of Retrieval cost(·) is $/#access. V_i is the ith video stored in the cloud; $time_i$ is the stored time of V_i, and $access_i$ is the number of accesses during $time_i$.

According to the formula, the TCO per video can be minimized by reducing storage or retrieval costs for cold videos, leading to a lower overall TCO for video cloud storage. Multitier storage in the cloud offers tradeoffs between storage and retrieval costs based on access frequency. By selecting an appropriate storage tier based on video access frequency, TCO can be minimized. For instance, calculations indicate that if a video is accessed fewer than 11 times per year, AWS's Infrequent tier is more cost-effective than other storage options.

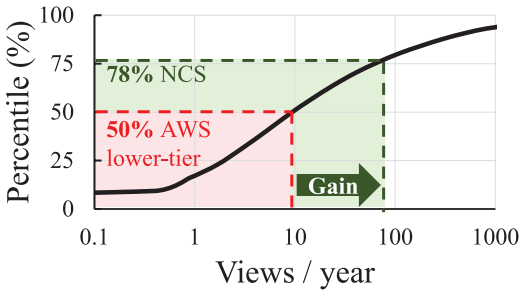


FIGURE 1. NCS expands the cold video coverage that can benefit from multitiered service provided by AWS.

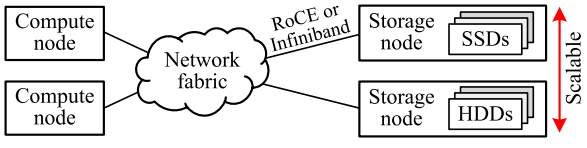


FIGURE 2. Disaggregated storage architecture in modern cloud.

Content-Aware SR

SR is a technique that transforms low-resolution (LR) images into high-resolution (HR) counterparts. With the advent of deep neural networks (DNNs), SR has achieved impressive results in reconstructing LR images. Recent research has highlighted that SR, when applied with a content-aware approach, can substantially reduce Internet bandwidth consumption in video streaming.[5] This method involves customizing SR DNN models for individual video content—referred to as content-aware SR, which offers superior enhancement compared to DNNs trained on generic datasets. In this study, we explore the application of content-aware SR in cloud storage systems, demonstrating its potential to reduce storage costs, especially for cold video content.

MOTIVATION

Why Is NCS Promising?

Neural-enhancement, particularly content-aware SR, shows great potential for cold video cloud storage for three main reasons. First, content-aware SR reduces storage costs while preserving video quality. For example, downscaling a 20-min 4K video to full high definition decreases file size from 1.17 GB to 0.77 GB, cutting storage costs by 34%. While this process incurs inference costs, they are minimal for long-term stored videos like cold videos. Second, cloud storage prices have stabilized, while cloud computing costs continue to decline. As shown in Figure 3(a), storage prices remain steady, whereas computing instance costs are dropping. This trend makes NCS an increasingly cost-effective solution. Finally, content-aware SR typically requires training for each video, leading to high computational costs. However, clustering similar videos and training a single DNN per group efficiently distributes these costs. Most videos can be processed using one DNN trained on a representative video, reducing training costs as the cluster grows. This approach significantly lowers expenses for large-scale video-sharing platforms.

More Flexible Storage Options for Cold Videos

Efficient storage management is essential for both cost reduction and improving user experience. Beyond cost, one of the key distinctions between storage tiers is their availability. Storage tiers with lower availability often result in long tail latency and reduced reliability, leading to degraded service quality and higher data recovery costs in certain scenarios. To address this, NCS provides storage tier upgrades to improve availability without additional cost. This ensures better accessibility and user experience, even for data such as surveillance footage that may occasionally require rapid access.

How to Maximize the Benefits of NCS?

To maximize the advantages of NCS, there are opportunities for further optimization, particularly in two key areas. First, while content-aware SR methods significantly reduce storage costs by storing downscaled video content, further improvements are possible. Videos uploaded to the cloud are typically stored in a compressed format using conventional codecs that are not tailored for SR applications. Developing an SR-optimized encoding approach for these codecs could further enhance storage efficiency. Second, while infrequent access to cold videos reduces retrieval frequency, the computational cost of SR inference remains significant. By leveraging the distinct properties of video data, it is feasible to reduce overall inference costs without compromising video quality. This study explores these two areas of improvement and their practical impact on reducing TCO.

DESIGN OF NCS

Overview

Figure 4 presents an overview of NCS architecture. NCS minimizes storage costs by dynamically adjusting video resolutions and storage tiers based on popularity. Cold videos—those with an annual average view count of 91 or fewer—are stored at lower resolutions in the cheapest storage tier. When requested, NCS restores these videos to their original resolution using content-aware SR DNNs and metadata. To further reduce TCO, NCS optimizes storage by profiling videos and embedding metadata alongside compressed files.

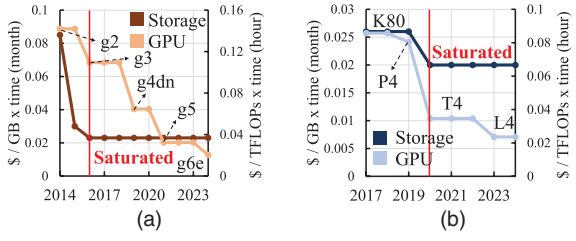


FIGURE 3. The price trend of storage and GPU instance of cloud providers. (a) Amazon Web Services. (b) Google Cloud Platform.

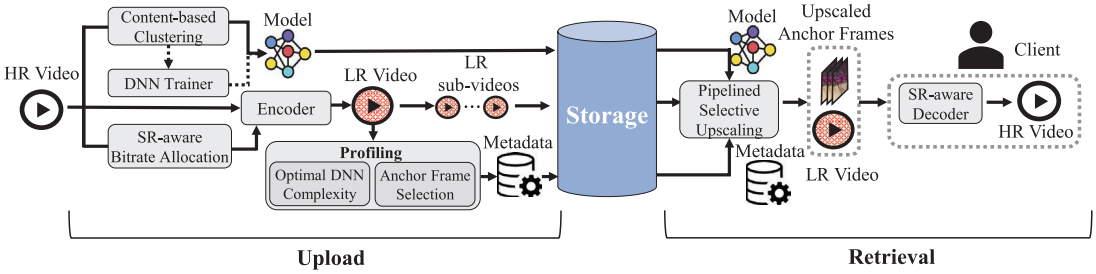


FIGURE 4. High-level overview of NCS.

Producing Content-Aware SR DNN

Training a content-aware SR DNN for each video is computationally expensive, making large-scale deployment impractical. To mitigate this, NCS clusters videos based on content similarity, allowing SR DNNs to be shared within groups. This reduces training costs while preserving video-specific enhancements. Unlike methods that rely solely on metadata classification or generic SR models, NCS employs a two-stage clustering process. It first classifies videos using metadata from uploaders or platforms. Since metadata alone lacks precision, NCS refines clustering with a vision encoder[6] that extracts feature vectors from key frames, ensuring more accurate grouping. NCS also dynamically adjusts clusters. If a video does not fit existing clusters, a new one is created with an initial SR DNN. As more videos join, the model undergoes iterative fine-tuning, continuously improving restoration quality. This adaptive approach optimizes computational efficiency while maintaining high-quality results, outperforming static clustering and generic SR models.

Profiling Video for TCO Optimization

NCS reduces storage costs through LR compression and further lowers TCO with additional optimizations. These include refining video encoding and minimizing SR DNN computing costs, enabled by profiling processes performed prior to storage.

SR-Aware Bitrate Allocation

Conventional video encoders prioritize visual quality but frequently neglect SR quality. NCS's analysis reveals that the relationship between allocated bitrate and SR quality varies across frames based on unique content characteristics. Building on this, NCS proposes a novel SR-aware video encoding method that dynamically allocates bitrate based on each frame's SR quality potential. To achieve this, the video is first encoded at multiple target bitrates, and SR quality is evaluated for key frames within each group of pictures (GOP) instead of all frames to reduce computational complexity. These data are used to determine the optimal bitrate allocation, minimizing overall video size while preserving target SR quality. A greedy algorithm is utilized to efficiently profile SR quality. Initially, each GOP is assigned the minimum bitrate. The bitrate is iteratively increased for the GOP offering the highest SR quality gain per unit of additional bitrate until the desired SR quality is achieved. This approach strikes an effective balance between computational efficiency and quality optimization.

THIS ADAPTIVE APPROACH OPTIMIZES COMPUTATIONAL EFFICIENCY WHILE MAINTAINING HIGH-QUALITY RESULTS, OUTPERFORMING STATIC CLUSTERING AND GENERIC SR MODELS.

Optimizing DNN Complexity

Similar to the relationship between bitrate and SR quality, the sensitivity of SR quality to DNN complexity also varies across frames. Utilizing this, NCS dynamically adjusts DNN complexity for each frame to minimize SR overhead while maintaining overall SR quality. Similar to SR-aware bitrate allocation, NCS profiles SR quality across various DNN configurations. NCS assesses the SR quality of key frames within GOPs and adjusts DNN complexity based on floating-point operations per second. Subsequently, a greedy algorithm is used to determine the optimal DNN complexity for each frame that satisfies the target quality. This adaptive strategy ensures efficient utilization of computational resources by focusing on frames where

increased complexity results in the most significant quality improvement.

Anchor Frame Selection

To reduce SR computing costs, NCS adopts an anchor frame selection strategy.[7] This approach uses specific anchor frames to effectively support the upscaling of other frames. Information from these anchor frames, upscaled using DNNs, is leveraged to upscale the remaining frames through an SR-integrated decoder.[7] This method drastically reduces the frames requiring direct DNN inference, cutting computational costs while preserving acceptable quality.

THIS METHOD DRASTICALLY REDUCES THE FRAMES REQUIRING DIRECT DNN INFERENCE, CUTTING COMPUTATIONAL COSTS WHILE PRESERVING ACCEPTABLE QUALITY.

Once the appropriate SR DNN is selected by clustering, the video is encoded with an optimized bitrate allocation. The encoded video is then analyzed to determine the optimal DNN complexity, which is used to assign a DNN to each chunk and select the corresponding anchor frames. Due to the dependencies between each optimization step, the process follows a fixed order. This information is stored as metadata with the video. The metadata, being minimal (< 100 KB), has a negligible impact on storage requirements.

Video Restoration

NCS encounters two challenges in SR-based video restoration: delays during the SR process, which degrade user experience, and reencoding of restored frames, which introduces significant computational overhead and increases processing time. To address these challenges, NCS leverages pipelining and hybrid video encoding.

Pipelining With Subvideos

NCS employs a pipelining mechanism that strategically overlaps SR processing with video restoration to minimize additional delays. By segmenting videos into subvideos along the temporal axis, NCS enables concurrent data retrieval and SR restoration. To maximize pipelining efficiency, NCS follows two key principles when dividing videos. First, to eliminate reencoding and preserve video quality during merging, segmentation is performed strictly at the GOP level. Second, while shorter subvideos enhance pipeline throughput, excessively small segments introduce inefficiencies due to the inherent latency threshold in storage retrieval. Balancing these factors, NCS optimally determines subvideo sizes to ensure efficient storage and processing. During restoration, subvideos are processed in parallel through the pipeline and seamlessly reassembled into a single video before transmission. This design not only minimizes restoration latency but also maintains high video quality, achieving an optimal tradeoff between speed and fidelity.

Hybrid Video Encoding

To reduce reencoding overhead, NCS adopts a hybrid video encoding approach.[7] This method efficiently minimizes computational overhead by restoring only anchor frames to their original resolution using SR DNN. The restored anchor frames are losslessly compressed with an image codec and packaged with the stored LR video into a single file using a hybrid encoder, then transmitted to the client. During playback, the anchor frames and SR-aware integrated decoder restore and decode the remaining nonanchor frames to their original resolution. This approach is hundreds of times more efficient than reencoding the entire video with conventional encoders.

EVALUATION

Experimental Setup

Setup

We use a modified version of enhanced deep super-resolution network[8] model for content-aware SR. To reduce costs, we employ a lightweight (~100 KB) DNN, pretrained on general image datasets and fine-tuned per video. TensorRT enables fast inference. Videos are encoded with VP9 and FFMPEG, using YouTube's encoding settings except for the target bitrate. HR and LR videos share the same encoding configurations, except for resolution. The dataset consists of various 2160p YouTube videos, each one minute long and centered on specific content themes. For SR-aware bitrate allocation and optimal DNN complexity, a greedy search algorithm considers 10 candidate bitrates and DNN complexities. About 10%–15% of the frames are selected as anchor frames. Clustering and profiling are performed once, incurring negligible costs over long-term storage and are excluded from TCO calculations.

Baseline

We use AWS S3 multitier storage as storage nodes and AWS EC2 g6e.xlarge spot instances (US$0.185/hour) as compute nodes. The baseline stores the HR-encoded video in the most cost-effective storage option based on access patterns, referred to as "Baseline Pareto."

Cost-Benefit Analysis

In this section, we perform a cost-benefit analysis of integrating our optimization techniques into the existing cloud storage system. Given practical limitations, we did not implement the full system end to end to directly measure costs. Instead, we estimated and compared the TCO by aggregating the costs of individual components.

Figure 5(a) shows the normalized TCO comparison for storing various video content between the Baseline and NCS over a 45-day access interval. Unlike the Baseline, NCS incurs SR overhead costs; however, these costs become negligible when the access interval is short. The evaluation shows that NCS reduces TCO by up to 40.8% compared to the baseline, depending on the video content. This indicates that while SR performance varies with the content and dynamic characteristics of videos, NCS consistently delivers high efficiency across various scenarios.

Figure 5(b) illustrates the Pareto optimality between NCS and the Baseline, demonstrating that the selected storage tier varies based on the access interval. NCS achieves a lower TCO than the Baseline for all cold videos with an access interval of four days or more. Additionally, as the popularity of the video decreases, the TCO savings ratio increases, highlighting that NCS is particularly optimized for storing cold data. Based on this calculation, Figure 1 shows that NCS can reduce the TCO for 78% of all videos in YouTube and achieve a 21.1% reduction in TCO compared to the Baseline.

Optimizing Storage Costs and Tier Allocation

As shown in Figure 5(b), NCS allows a lower TCO than the baseline or access to a higher storage tier at the same TCO for video storage. Based on this calculation, NCS offers the opportunity to upgrade 16% of YouTube videos to an upper storage tier. This demonstrates that NCS not only reduces TCO but also enhances user experience through upgraded storage tiers.

Component Analysis

Clustering

To validate the proposed clustering method, a vision encoder was used to extract feature vectors from 40 sports-themed videos, resulting in six clusters with four to eight videos each. Figure 6(a) compares SR quality using cluster-specific DNNs, a general DNN, and video-specific DNNs. Cluster-specific DNNs outperformed the general model and achieved similar quality to video-specific DNNs. This demonstrates that content-based clustering reduces the number of DNNs needed while maintaining high performance and computational efficiency.

SR-Aware Encoding

Figure 6(b) compares the rate-distortion performance between naive encoding and SR-aware encoding with SR-aware bitrate allocation. The quality of SR is measured using peak signal-to-noise ratio, while the size of video is represented by the bitrate. The curve clearly highlights the advantages of SR-aware encoding, showing that it can achieve up to a 15% reduction in bitrate while maintaining the same SR quality.

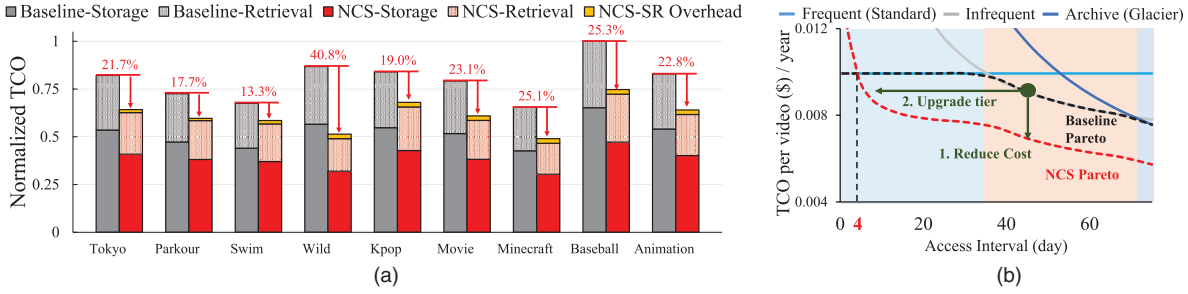


FIGURE 5. NCS provides better TCO for cold video (access interval > four days) compared to Baseline. (a) Cost comparison with different contents (Access interval = 45 days). (b) NCS Pareto versus Baseline Pareto.

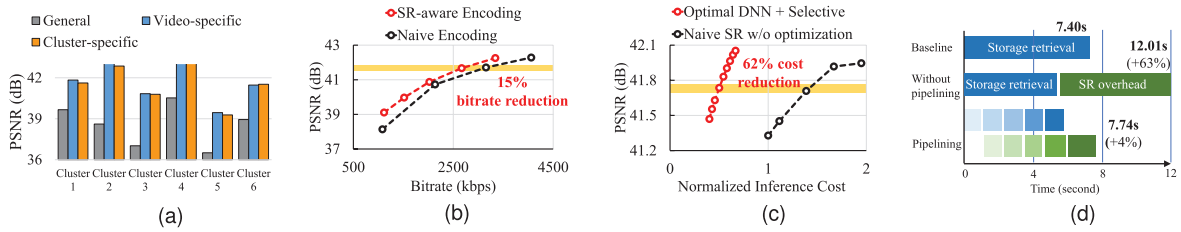


FIGURE 6. Component analysis: (a) Clustering. (b) SR-aware encoding. (c) SR-aware decoding. (d) Clustering.

SR-Aware Decoding

Figure 6(c) compares the SR-aware decoding approach, which leverages optimized DNN complexity and anchor-frame selection, with naive SR methods in terms of quality and computational efficiency. As shown in the figure, SR-aware decoding can achieve the same SR quality while reducing inference costs by an average of 62%.

Restoration

Figure 6(d) illustrates the latency comparison of pipelining in NCS. The experiment utilized a 10-min 4K video file with a size of 703 MB. NCS benefits from reduced storage retrieval latency due to smaller file sizes, but without pipelining, the overall retrieval time increases by 63% compared to the baseline due to the time required for SR processing. By employing pipelining with a minimum subvideo size of 100 MB, the overall latency is reduced by up to 36%, achieving a latency level similar to the baseline.

FUTURE WORK AND DISCUSSION

In this section, we explore the possibilities for further expanding and refining our work.

Temporal Popularity-Aware Storage Strategy

NCS optimizes video storage costs by analyzing the annual average access interval of videos. However, this approach assumes that video popularity remains constant over time, which is not realistic. Most videos receive the majority of their views within days of upload, while only a small fraction maintain long-term viewership.[9] A more effective strategy is to store videos in their original resolution upon upload and transition them to cold storage once their view count drops below a threshold, reducing costs dynamically. Integrating a temporal popularity prediction model with NCS would enable proactive storage management, further optimizing cost savings across a larger proportion of videos.

Adaptive Popularity-Based Storage Strategy

Most videos uploaded to video platforms exhibit a logarithmic viewing pattern over time.[9] However, some videos may unexpectedly surge in views long after upload, leading to frequent lower-tier storage access and raise TCO. To address this, these videos can be moved to upper-tier storage for LR-to-HR conversion, though this may degrade perceptual quality and affect the original viewing experience. A potential solution to this problem is to implement exception handling for videos with highly volatile viewing patterns. Specifically, the HR versions of these videos can be backed up in archive storage (e.g., AWS S3 Deep Archive) and moved to upper-tier storage if the video experiences a sudden surge in views. his approach minimizes TCO spikes for videos with volatile viewing patterns while maintaining high-quality playback and preserving the user experience.

CONCLUSION

We proposed NCS as a cost-effective solution for storing cold videos in disaggregated cloud storage systems. By leveraging neural enhancement, particularly content-aware SR, NCS demonstrates potential benefits such as cost reduction, storage tier upgrades, and expanded applicability to cold videos. With the continued advancement of deep learning and computational resources, we believe neural enhancement will play a pivotal role in revolutionizing disaggregated cloud storage systems, making them more efficient and cost-effective.

ACKNOWLEDGMENTS

This article is a full-length version of an earlier work.[10] We appreciate anonymous reviewers. This work was supported by Samsung Electronics, Co., Ltd. (IO221107-03428-01), the National Research Foundation of Korea (NRF) (RS-2024-00340099), and the Institute for Information and Communications Technology Promotion (IITP) (RS-2023-00221040).

REFERENCES

1. "Google transparency report," 2019. [Online]. Available: https://transparencyreport.google.com/?hl=en
2. J. Summers, T. Brecht, D. Eager, and A. Gutarin, "Characterizing the workload of a Netflix streaming video server," in *Proc. IEEE Int. Symp. Workload Characterization*, Piscataway, NJ, USA: IEEE Press, 2016, pp. 1–12, doi: 10.1109/IISWC.2016.7581265.
3. W. Bai et al., "Empowering azure storage with RDMA," in *Proc. USENIX Symp. Networked Syst. Des. Implementation (NSDI)*, 2023, pp. 49–67.
4. R. McGrady, K. Zheng, R. Curran, J. Baumgartner, and E. Zuckerman, "Dialing for videos: A random sample of YouTube," *J. Quantitative Description: Digit. Media*, vol. 3, pp. 1–85, Dec. 2023, doi: 10.51685/jqd.2023.022.
5. H. Yeo, Y. Jung, J. Kim, J. Shin, and D. Han, "Neural adaptive content-aware internet video delivery," in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 645–661.
6. A. Radford et al., "Learning transferable visual models from natural language supervision," in *Proc. Int. Conf. Mach. Learn. (PMLR)*, 2021, pp. 8748–8763.
7. H. Yeo, H. Lim, J. Kim, Y. Jung, J. Ye, and D. Han, "Neuroscaler: Neural video enhancement at scale," in *Proc. ACM SIGCOMM Conf.*, 2022, pp. 795–811.
8. B. Lim, S. Son, H. Kim, S. Nah, and K. M. Lee, "Enhanced deep residual networks for single image super-resolution," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. Workshops*, 2017, pp. 1132–1140, doi: 10.1109/CVPRW.2017.151.
9. L. C. O. Miranda, R. L. T. Santos, and A. H. F. Laender, "Characterizing video access patterns in mainstream media portals," in *Proc. 22nd Int. Conf. World Wide Web*, 2013, pp. 1085–1092.
10. J. Lim, J. Ye, J. Kim, H. Lim, H. Yeo, and D. Han, "Neural cloud storage: Innovative cloud storage solution for cold video," in *Proc. 15th ACM Workshop Hot Topics Storage File Syst.*, 2023, pp. 1–7.

JINYEONG LIM is a Ph.D. student at KAIST, Daejeon, 34141, South Korea. His research interests include computer systems, video processing, and machine learning optimization. Lim received his master's degree in electrical engineering from KAIST. Contact him at jylim9999@kaist.ac.kr.

JUNCHEOL YE is a Ph.D. student at KAIST, Daejeon, 34141, South Korea. His research interests include machine learning system, video system, and network system. Ye received his M.S. degree from KAIST. Contact him at juncheol@kaist.ac.kr.

JAEHONG KIM is a researcher affiliated with KAIST, Daejeon, 34141, South Korea. His research focuses on immersive multimedia systems, networked computer systems, and AI-driven video streaming. Kim received his Ph.D. degree in electrical engineering from KAIST. Contact him at jaehong950305@gmail.com.

HWIJOON LIM is a researcher affiliated with KAIST, Daejeon, 34141, South Korea. His research focuses on improving machine learning and networking systems by designing efficient algorithms and implementing system-level optimizations in real-world systems. Lim received his Ph.D. degree in electrical engineering from KAIST. Contact him at hwijoon.lim@gmail.com.

HYUNHO YEO is a researcher affiliated with KAIST, Daejeon, 34141, South Korea. His research focuses on optimizing video streaming and networking systems through deep learning. Yeo received his Ph.D. degree in electrical engineering from KAIST. Contact him at chaos5958@gmail.com.

JUNHYEOK JANG is a Ph.D. student at KAIST, Daejeon, 34141, South Korea. His research interests include computer express link, disaggregated systems, and machine learning acceleration. Jang received his master's degree in electrical engineering from KAIST. Contact him at jhjang@camelab.org.

MYOUNGSOO JUNG is a full professor in the School of Electrical Engineering, KAIST, Daejeon, 34141, South Korea, and the CEO of Panmnesia, Inc. His research interests include computer architecture, operating system, storage systems, nonvolatile memory, parallel processing, heterogeneous computing, and computer express link. Jung received his Ph.D. degree in computer science from Pennsylvania State University. Contact him at m.jung@kaist.ac.kr.

DONGSU HAN is a full professor in the School of Electrical Engineering and Graduate School of Artificial Intelligence, KAIST, Daejeon, 34141, South Korea. His research interests include networked/cloud systems design, artificial intelligence (AI) for systems, and systems for AI. Han received his Ph.D. degree in computer science from Carnegie Mellon University. He is a corresponding author of this article. Contact him at dhan.ee@kaist.ac.kr.

Improving Remote File Access in Distributed Object Stores by Decoupling Metadata and Data Paths Using NVMe-oF

Daegyu Han and Sungho Moon, *Sungkyunkwan University, Suwon, 16419, South Korea*

Kyeungpyo Kim, *GlueSys Co., Ltd., Anyang, 14055, South Korea*

Sung-Soon Park, *Anyang University, Anyang, 14028, South Korea, and GlueSys Co., Ltd., Anyang, 14055, South Korea*

Beomseok Nam, *Sungkyunkwan University, Suwon, 16419, South Korea*

Storage network protocols such as NVMe-oF operate below the file system layer. Therefore, even when NVMe-oF allows storage volumes to be shared across the network, compute nodes cannot access remote files managed by another node's file system without a cluster file system. In conventional distributed systems, accessing files owned by a remote node requires communication with the remote node via Remote Procedure Call (RPC). The remote node then retrieves the data from a disaggregated storage node and transfers it to the requesting node. To reduce redundant network traffic, this study proposes remote direct I/O (RDIO), which separates RPC-based remote data access into two distinct planes: a metadata plane for file mapping and a data plane for direct access to remote storage. The data plane ensures data flows only through the storage network. We integrate RDIO into MinIO and show that RDIO significantly improves performance by reducing remote data movement between nodes.

With the increasing demand for scalable storage services and efficient resource utilization, modern data centers are rapidly transitioning from monolithic server architectures where compute and storage resources are tightly coupled to disaggregated storage architectures.[1] State-of-the-art storage network protocols such as NVMe-oF (NVMe over Fabrics) enable physical separation of compute and storage resources, and allow multiple hosts to dynamically share storage resources.[2]

However, the storage network protocol is a block-level protocol that operates below the file system layer. Therefore, compute nodes cannot share access to files owned by remote compute nodes without file system support. This limitation prevents distributed object stores from leveraging the distance connectivity of NVMe-oF. That is, distributed object stores need to use Remote Procedure Call (RPC) communication to access remote files. This RPC communication between compute nodes results in unnecessary data transfer over the network. Specifically, data transmission between the remote compute node and the storage node is followed by the transmission to the local compute node. This sequence of data transfers doubles the input–output (I/O) latency, and we refer to this problem as the *double transfer* problem.

To resolve the double transfer problem, we propose RDIO, a novel remote file access mechanism to enable direct access to storage volumes managed by remote nodes without using conventional shared-disk file systems. RDIO consists of a *metadata plane* and a *data plane*.

Digital Object Identifier 10.1109/MM.2025.3564477
Date of publication 30 April 2025; date of current version 24 December 2025.

The metadata plane retrieves file mapping information from remote nodes and converts file offsets into physical block addresses. Using the physical block addresses, the data plane directly reads and writes the disaggregated storage volume via NVMe-oF. This ensures file blocks are transmitted over the network only once such that both network and I/O stack overhead are reduced.

Key contributions of this study are as follows:

- First, this study introduces a novel approach for accessing files on remote nodes by leveraging the distance connectivity of NVMe-oF. RDIO separates metadata and data planes and minimize RPC communication between compute nodes.
- Second, this study presents *parity-free decoding* and *deferred parity encoding* optimizations, which improve erasure coding workflows by enabling asynchronous parity block encoding while ensuring data availability even when remote nodes are inaccessible.
- Third, we implement RDIO in MinIO, a popular distributed object store, demonstrating its potential to improve performance in distributed object storage systems.

DISTRIBUTED OBJECT STORE AND DISAGGREGATED STORAGE

Distributed Object Store

Distributed object stores, such as MinIO, are widely used in modern data centers to efficiently handle large volumes of unstructured data. To ensure high availability and fault tolerance, these systems typically employ replication or erasure coding.[3]

Designed for a shared-nothing architecture, distributed object stores utilize single-node file systems, such as EXT4 and XFS, instead of shared-disk file systems. Figure 1(a) shows an example of a distributed object store using erasure coding on three nodes with direct-attached storage (DAS), where an object (A) is encoded into two partition blocks (D1 and D2) and one parity block (P). When reading the object, if there is no failure, only the two partition blocks (D1 and D2) are accessed and merged to restore the object. If partition blocks are corrupted or one of the nodes that contain partition blocks does not respond, the parity block is used to decode and restore the object.

Storage Disaggregation

Figure 1(a) shows the architecture of traditional monolithic server with DAS. Since compute and storage resources are tightly coupled, it is difficult to scale each resource individually, which leads to over-provisioning of resources and low resource utilization. To overcome this issue, modern data centers have shifted toward disaggregated storage architectures, where storage resources are physically separated from compute nodes,[1] as shown in Figure 1(b). Storage network protocols, including NVMe-oF and iSCSI, allow disaggregated storage devices to be accessed as if they were local block devices.[2] NVMe-oF is the state-of-the-art storage protocol that transmits block-level I/O requests (NVMe commands) over the network. With remote direct memory access support, NVMe-oF reduces data copy overhead and minimizes CPU usage.[2]

Shared Access to Disaggregated Storage

Storage disaggregation allows multiple compute nodes to access storage resources simultaneously. However, a storage network protocol alone does not resolve conflicts that occur when multiple compute nodes compete for the same block.[4,5] For example, allocating a block that is already in use by another compute node causes conflicts. To resolve such conflicts

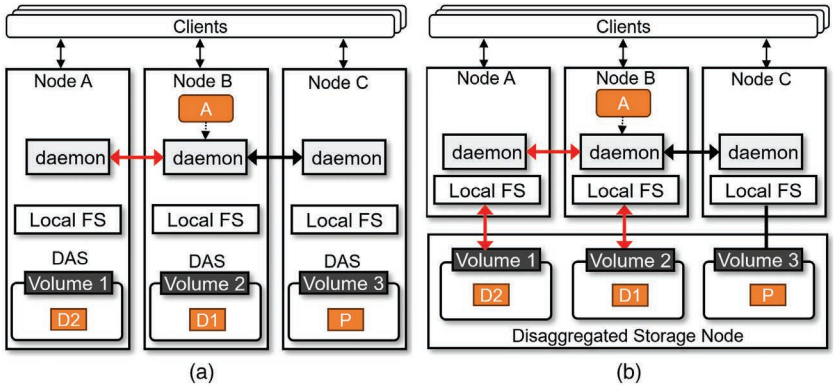


FIGURE 1. Distributed object store on different storage architectures. (a) Direct-attached storage. (b) Disaggregated storage.

and enable shared access to files on remote storage, shared-disk file systems have been used.

General-purpose Portable Operating System Interface shared disk file systems run daemon processes on each compute node to resolve conflicts and prevent inconsistent file accesses. However, due to these consistency checks, shared-disk file systems suffer from nonnegligible overhead and exhibit lower performance compared to single-node file systems.

Double Data Transfer Problem

When deployed on disaggregated storage, accessing a file owned by a remote node in distributed object storage causes the file data to traverse the network twice for a single I/O operation. Specifically, suppose a data node requests file data from a remote data node, which commonly occurs in distributed object stores that use erasure coding. The remote node retrieves the file data from the disaggregated storage node, using the storage network. Then, the remote node sends the file data to the requesting node, using the compute network. This double data transmission consumes significant CPU cycles on both data nodes and increases network traffic.[4,5]

DESIGN AND IMPLEMENTATION OF RDIO

Figure 2 shows the architecture of RDIO. Each node has its local disk volume and accesses it using a local file system. These disk volumes are shared by remote nodes through a storage network protocol. We refer to these volumes as remote volumes. To avoid any potential file system inconsistency, file system-level access to remote volumes is not allowed. In the next subsection, we detail how RDIO supports remote file access.

Remote Direct I/O Path

Rdev Hashmap

RDIO manages metadata about remote volumes in the `rdev hashmap`. Specifically, this hashmap maps remote nodes' file system mount points to their corresponding remote volumes on the local node.

Write

In the legacy system, in order to write a file to a storage volume managed by a remote node, the write request and data were sent to the remote node and written. In contrast, RDIO uses the metadata plane that sends an RPC request to the remote node (Node B in Figure 2) for file allocation, including metadata such as the file path and data size. The RPC stub on the remote node triggers the extent handler, which uses the `fallocate()` system call to allocate the file. The file mapping information

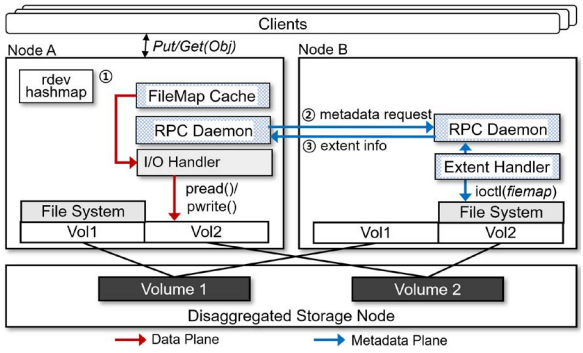


FIGURE 2. RDIO architecture and I/O path.

for the allocated file is retrieved using the `ioctl(fiemap)` and returned to the requester node (Node A). Once the metadata plane for remote write is handled, the requester node performs the data plane by converting the file offset to a logical block address and executing the `pwrite()` system call on the raw block device.

Read

For read operations, the requester node sends an RPC request to retrieve the file mapping information from the remote node, which queries the file mapping information from the file system using the extent handler and returns it to the requester node. Then, similar to the write process, the requester node uses the file mapping information and issues the `pread()` system call to directly read data from the disaggregated raw block device. It is noteworthy that unless there is significant memory pressure, the `ioctl(fiemap)` does not result in disk I/O since the file mapping is likely to have been cached in the remote node's file system cache.

RDIO provides shared access to files on remote storage while still leveraging the existing Linux native file system. The remote data transfer process in RDIO is divided into metadata and data planes, ensuring that data are transmitted only once across the network fabric, unlike the traditional RPC-based access. Specifically, while data are transmitted through the storage network protocol, its metadata (e.g., file mapping information) is retrieved between nodes via RPC communication. Despite RPC communication on the metadata plane, since metadata is much smaller than the actual data, RDIO can reduce CPU usage and network traffic during remote data transfer.

Improving Metadata Plane for File Mapping

Every read or write request requires file mapping information, and each I/O operation must communicate

with the remote node via RPC to access the file system's metadata. Although the size of the metadata is not large, there may be issues with increased I/O latency. To address this problem, RDIO implements *FileMap cache*, a cache for file mapping information. The FileMap cache eliminates one network hop in the metadata plane, allowing only the data plane to be performed through cached entries. As a result, when a cache hit occurs in Node A's FileMap cache, the RPC request for file mapping is skipped, and the data can be directly read from the remote volume.

Entries in the FileMap cache consist of the file path and file mapping information. As a metadata cache that does not store actual data, it has a small memory footprint, making it suitable for environments handling large datasets. Additionally, the FileMap cache takes advantage of temporal locality for frequently reaccessed entries by applying the LRU cache replacement policy.

Cache consistency in distributed systems is crucial to ensure that all nodes have a consistent view of the data. On the other hand, there exist various modern applications that allow for weak cache consistency models. In distributed object stores such as MinIO, which consider the immutability of objects, complex cache consistency mechanisms used in generic shared disk file systems are not necessary. If objects are mutable, their file mapping information stored in the FileMap cache may become stale. This poses a risk of reading incorrect data from the remote storage. Therefore, the cached entries need to be invalidated when the object's file mapping is changed by other nodes as in conventional shared disk file systems.

MINRDIO: INTEGRATING RDIO INTO DISTRIBUTED OBJECT STORAGE

We integrate RDIO into MinIO (RELEASE.2022-05-04T07-45-27Z) to verify the applicability of RDIO and name it MinRDIO. In this section, we first describe how RDIO is specifically applied in MinRDIO, followed by a discussion of the novel resilience mechanism enabled by RDIO.

Putting it All Together

Write

MinIO uses erasure coding to split a new object into encoded blocks (i.e., partition and parity blocks) and then distributes them across multiple nodes along with the metadata file (i.e., `xl.meta`) using consistent hashing. When writing encoded blocks to remote nodes, the remote write path of RDIO is used. For metadata files that are considerably smaller (e.g., a few hundred bytes) than the encoded blocks, the existing RPC-based remote write path is used as is. Additionally, the RPC requests for file mapping that occur in RDIO's remote write path are performed asynchronously during object encoding. This achieves an effect similar to executing only the data plane while hiding the processing in the metadata plane.

MinIO ensures data persistence by flushing encoded blocks and their associated metadata files to disk. MinRDIO aims to achieve the same goal, but since remote volumes are not mounted on the local node, file system operations such as `fdatasync()` cannot be directly invoked for encoded blocks. Note that MinIO's write operation consists of two phases: writing the encoded blocks first, followed by the metadata files. Similarly, MinRDIO uses RDIO to store the encoded block and then writes the metadata file to a remote node. To ensure the persistence and crash consistency, the file path to the encoded block is provided to the remote node such that the encoded block is flushed to disk first using `fdatasync()`, followed by the metadata file.

Read

Since MinIO uses consistent hashing to determine the location of partition blocks, it first reads all metadata files to identify their locations and then accesses the corresponding partition blocks. When reading metadata files and partition blocks from remote nodes, both use the remote read path of RDIO, and the obtained file mappings are cached in the FileMap cache. If the file mapping is cached in the FileMap cache, the partition block or metadata file is read from the remote volume without metadata plane RPC requests.

Direct I/O Versus Buffered I/O

Linux file systems typically use buffered I/O with the page cache to minimize disk I/O. They also support direct I/O, which bypasses the page cache, allowing applications to choose between these I/O options.

When using direct I/O, both MinIO and MinRDIO bypass the page cache of both local and remote compute nodes. As a result, both incur the same storage access cost for each I/O request. However, MinIO consumes additional network bandwidth between compute nodes, whereas MinRDIO avoids this by enabling direct access to the remote shared disk volume.

Although MinRDIO is unable to exploit buffered I/O for write operations due to the constraints of using raw devices, the cost of remote I/O remains comparable to MinIO with buffered I/O, as both need to flush data to

storage for persistence. Since Linux treats block devices as files, raw devices can benefit from buffered I/O, allowing MinRDIO to leverage the local node's page cache for read operations. It reduces unnecessary network round trips compared to MinIO, which relies on the remote node's page cache.

Resilience Mechanism in MinRDIO

In large-scale distributed systems, various failures, e.g., software, hardware, and network partitioning, occur frequently. To mitigate these issues, distributed object stores implement resilience mechanisms such as erasure coding and replication. However, these come with additional computational and space overhead.

Storage nodes contain multiple storage devices, and if a single storage node holds both data and parity blocks, it can become a single point of failure. This study does not address storage node failures, nor does it consider cases where partition blocks are corrupted. These failures need to be addressed by replication or other fault-tolerance mechanisms, which are orthogonal to the parity-based optimization that we describe next.

Parity-Free Decoding

In the event of a compute node failure, vanilla MinIO restores partition blocks owned by the failed node by reading parity blocks.

In contrast, MinRDIO implements parity-free decoding, which reconstructs the object without reading parity blocks. Without reading the parity block, MinRDIO can access lost partitions from other available compute nodes, allowing the original object to be recovered without the need to read or decode parity blocks.

However, although parity blocks are not needed, file mapping information from the failed node is still needed. If the file mapping is available in the FileMap cache in the local node, MinRDIO reads the file mapping from the FileMap cache. If the file mapping is not found in the local cache, MinRDIO mounts the file system of the remote storage volume on the local node. Then, it retrieves the file mapping and read the partition block. Alternatively, MinRDIO can fall back to the legacy method of decoding parity blocks to recover the object.

Deferred Parity Encoding

MinRDIO can directly access partition blocks managed by remote nodes even when the nodes are unavailable. That is, the availability of remote nodes is not relevant to the availability of partition blocks, and thus MinRDIO minimizes the need for parity blocks.

In disaggregated storage systems, multiple block devices are often configured using RAID to create logical disk volumes, such that even when a block device fails, its partition blocks can be restored. In contrast, even if RAID is used in disaggregated storage systems, the vanilla MinIO requires parity blocks because partition blocks can be accessed only through their corresponding nodes.

Since parity blocks at the MinRDIO layer are not essential for data availability, we propose *Deferred Parity Encoding*. In this approach, MinRDIO writes partition blocks synchronously while writing parity blocks asynchronously.

Although parity blocks are not essential in MinRDIO, we let MinRDIO use parity blocks for legacy support. Additionally, having parity blocks at the MinRDIO layer helps recover from various scenarios such as where partition blocks are distributed across multiple disaggregated storage nodes, and some of those nodes fail simultaneously.

EXPERIMENTS

Experiment Setup

We conduct a performance evaluation using a six-node cluster (one client node, four compute nodes, and one storage node). Each node is equipped with two 10-core Xeon Gold 5115 processors (2.4 GHz/14 MB) running on Linux kernel 5.3.0-24-generic with hyperthreading enabled and 64-GB DRAM. The storage node has two eight-core Xeon Silver 4215 processors (2.50 GHz/11 MB), 64-GB DRAM, and eight Samsung PM983 U.2 NVMe SSDs. All nodes are connected via a 56 Gbps Mellanox ConnectX-4 Infiniband interconnect.

Following the *(k,r)* erasure coding construction, we use (6, 2) configuration, consisting of six data partitions and two parity partitions. Each of the four nodes is configured with two NVMe volumes, each mounted with an XFS.

We compare the performance of MinRDIO to MinIO, both configured to use direct I/O. For MinRDIO, we evaluate the performance with and without FileMap cache for read queries. In the former case, the FileMap cache capacity is set to accommodate 4 M objects, enough to cache all file mappings for the dataset and cache warmup is performed before query execution. The latter is named MinRDIO-FMC.

Evaluation of RDIO

First of all, we quantitatively measure and analyze the effects of RDIO using Representational State Transfer-application programming interface-based microbenchmarks. To measure throughput, we use 40 threads

to execute requests on datasets of 40 GB, varying the object size, while latency analysis is measured single-threaded on 32-kB objects. To evaluate CPU consumption and network traffic, we use 1024-kB objects.

Throughput and Latency

Figure 3(a) and (b) show that MinRDIO achieves up to 11% higher write throughput and up to 48% higher read throughput than MinIO by avoiding the double transfer. Compared to MinRDIO-FMC, MinRDIO shows 9%–23% higher throughput because RPC communication for file mapping retrieval is eliminated. As the object size increases, the I/O time increases. Accordingly, the relative overhead of RPC in obtaining file mapping is reduced.

When the object size exceeds 16 MB, MinRDIO exhibits slightly lower throughput than MinIO. Unlike MinIO, which allows remote compute nodes to process I/O for remote blocks, MinRDIO handles all remote I/Os on the local compute node. For small objects, the benefit of reduced network traffic outweighs the impact of I/O contention. However, for large objects, increased I/O contention on the local compute node negates the advantage of RDIO. This limitation could be mitigated by dynamically enabling or disabling RDIO based on the compute node's I/O load. We leave this as future work.

Despite the benefits of RDIO, the improvement in write throughput is not as significant as that in read throughput. To understand the reason for the performance difference, we analyze the latency as shown in Figure 3(c) and (d).

Surprisingly, the actual latency of writing encoded blocks (`WriteBlocks`) in MinRDIO is 85% lower than that of MinIO. However, the performance improvement is hidden because a significant portion of the total work time is spent on writing metadata files, committing (`Commit`) which changes the file path of encoded blocks to the actual bucket name, and encoding objects (`Encode`). MinRDIO shows slightly higher latency than MinIO because it performs `fdatasync()` on encoded blocks during commit. On the other hand, the reason why the encoding latency of MinRDIO is reduced is because other tasks are performed asynchronously, unlike MinIO which synchronously waits for completion during the object encoding.

MinRDIO shows 62% lower read latency than MinIO by affecting metadata (`ReadMeta`) and partition blocks (`ReadParts`). Although the overhead of the object storage interface related to returning objects to the client accounts for a significant portion of the total operation time, the performance improvement effect of RDIO is still remarkable.

CPU Consumption and Network Traffic

Figure 3(e) shows that MinRDIO consumes 12% less CPU when writing objects and 37% less CPU when reading objects compared to MinIO. This is because MinRDIO only transfers data to remote volume via a storage network. Write operations consume more CPU resources than read operations due to the need for object encoding and the writing of both data and parity blocks.

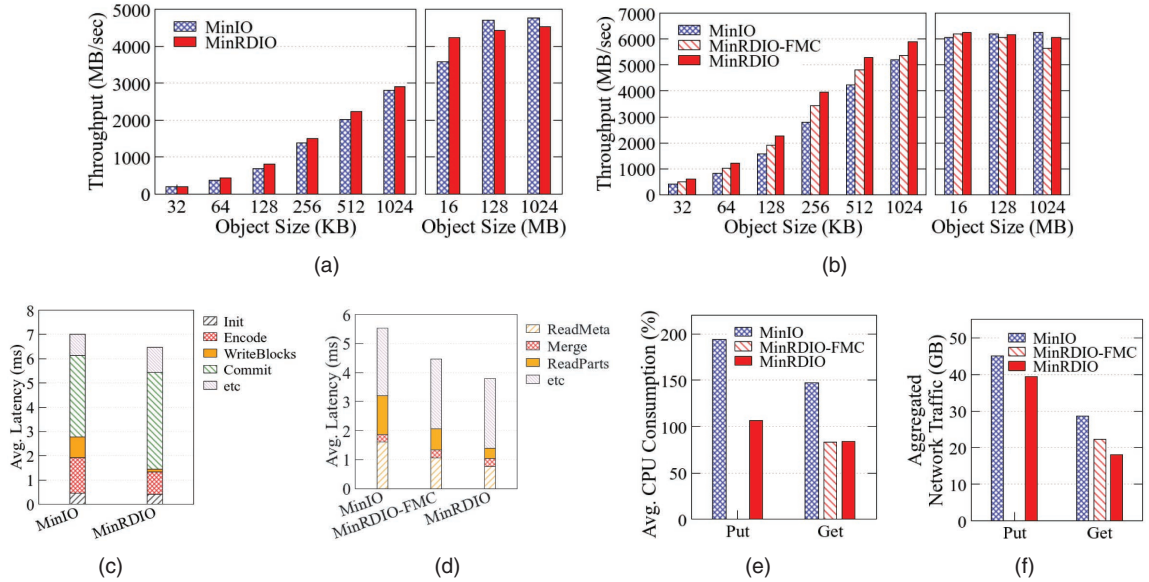


FIGURE 3. Quantification results. (a) Throughput (Put). (b) Throughput (Get). (c) Latency breakdown (Put). (d) Latency breakdown (Get). (e) CPU consumption. (f) Network traffic.

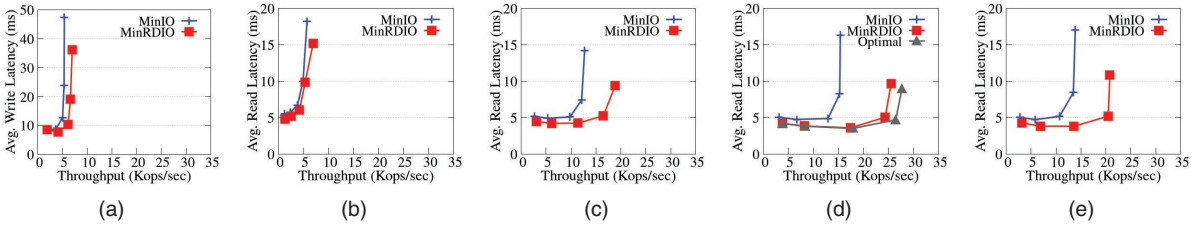


FIGURE 4. Scalability results: latency-throughput analysis. (a) Load. (b) Workload A. (c) Workload B. (d) Workload C. (e) Workload D.

As shown in Figure 3(f), MinIO generates 1.82× and 1.76× more traffic than MinRDIO for write and read operations, respectively. This is because data are transferred redundantly between compute and storage nodes as well as between compute nodes. Considering the network traffic difference between MinRDIO-FMC and MinRDIO, the network traffic used by the metadata plane RPC for file mappings is not that much.

Evaluation of Scalability

The experiments shown in Figure 4 show latency-throughput curves comparing the performance of MinRDIO with MinIO for Yahoo! Cloud Serving Benchmark workloads, varying the number of client threads from 16 to 256. For Load, we measure the average latency for write operations, and for the other workloads, we measure the average latency for read operations. For workload C (100% reads), we compare the performance with the optimal case (`Optimal`), which performs local read operations by routing remote read requests to mounted remote volumes on the local node. Referring to recent studies,[6,7] we set the object size to 32 kB–1 MB, and the key-value size and request distribution follows a Zipfian distribution. The dataset is filled with 800,000 records (about 80 GB) each containing random string data. Then, 800,000 read and write queries are submitted for each workload.

As the number of clients increases, MinRDIO has lower latency and higher throughput overall than MinIO. This is because remote access becomes more frequent and MinRDIO avoids TCP/IP communication in the data path. Due to the expensive write process of MinIO, throughput gradually saturates and latency increases from 64 threads for write-intensive workloads compared to read-intensive workloads. Nevertheless, for write-intensive workloads Load and Workload A (50% write and 50% read), MinRDIO achieves 1.29× and 1.21× higher throughput, respectively, by reducing CPU consumption and network traffic. In read-intensive workloads, significant performance improvements are observed, especially for read-only workload C, MinRDIO shows 1.66× higher throughput and 33% lower latency than MinIO at 256 threads.

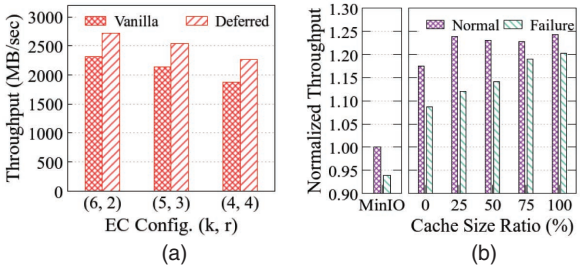


FIGURE 5. Resilience mechanism performance. (a) Encoding (Put). (b) Decoding (Get).

MinRDIO achieves performance comparable to `Optimal` overall. However, at 256 threads, internal operations such as the FileMap cache impact parallelism, resulting in approximately 7% lower throughput.

Evaluation of Resilience Mechanism

Finally, we evaluate the performance of MinRDIO's resilience mechanism. Using microbenchmarks, we query 10,000 4-MB objects (about 40 GB) with 40 threads.

Figure 5(a) shows the impact of deferred parity encoding varying on the erasure coding configuration *(k, r)*, assuming RAID-5 storage. We compare deferred parity encoding (denoted as `Deferred`) with the default encoding scheme of MinRDIO (denoted as `Vanilla`). `Deferred` can take advantage of the ability to read all partition blocks stored in a storage node even if a failure occurs, thereby reducing the object encoding time by asynchronously storing parity blocks. As a result, the throughput of `Deferred` is up to 20% higher than that of `Vanilla`. As *r* increases from 2 to 4, the write throughput slightly decreases, which is due to the larger size of the partitions.

In the experiment in Figure 5(b), we evaluate the impact of parity-free decoding by varying the ratio of cache size to the number of objects in the FileMap cache. After populating the dataset, we induce a failure in one of the nodes except the one that received the client's request. The `Normal` scenario represents no `failure`, while the Failure represents a scenario where the object storage process is terminated due to a system failure and the node is inaccessible. Due to the

consistent hashing that randomly distributes partitions, some objects may not be affected by the object decoding because their parity blocks are stored on the failed node. The results of this experiment are normalized to the throughput in the `Normal` scenario of MinIO.

When a failure occurs, MinIO's `Failure` throughput decreases by 7% compared to the `Normal`. This decrease is because the object reconstruction involves decoding parity blocks to recover missing partition blocks. On the other hand, MinRdio can return the object to the client using only the partition blocks without performing decoding if the file mapping of the partition block is cached, achieving up to 1.27× higher throughput than MinIO. When a cache miss occurs in the FileMap cache for a partition block, we set an option to fall back to the default decoding mechanism. Therefore, as the cache size ratio decreases, the throughput decreases due to expensive decoding and causes a performance gap with `Normal`. However, MinRdio achieves higher throughput than MinIO because it utilizes remote read paths even when reading parity blocks.

CONCLUSION

In this study, we explored practical and novel designs for shared storage systems in disaggregated storage. Due to the lack of file-level shared access support in storage network protocols, distributed object stores rely on RPC-based access, even when storage volumes in disaggregated storage are shareable. To address this challenge, we proposed Rdio, which enables shared access to remote data by leveraging file mapping information while making use of existing Linux native file systems. To showcase its applicability, we implement MinRdio and design a new resilience mechanism that takes advantage of Rdio. Our performance evaluation shows that MinRdio improves I/O throughput while reducing CPU consumption and network traffic by eliminating redundant remote data transfer.

ACKNOWLEDGMENTS

This work was supported by an IITP grant funded by the Korea government (MSIT) RS-2021-II210862 (2021-0-00862) and RS-2024-00461572.

REFERENCES

1. P. X. Gao et al., "Network requirements for resource disaggregation," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, 2016, pp. 249–264.
2. "NVM express over fabrics," NVM Express, Beaverton, OR, USA, Oct. 2019. [Online]. Available: https://nvmexpress.org/wp-content/uploads/NVMe-over-Fabrics-1.1-2019.10.22-Ratified.pdf
3. "MinIO quickstart guide." GitHub. Accessed: Dec. 1, 2024. [Online]. Available: https://github.com/minio/minio
4. T. A. Nguyen et al., "NVMe-driven lazy cache coherence for immutable data with NVMe over Fabrics," in *Proc. 16th IEEE Int. Conf. Cloud Comput. (CLOUD)*, 2023, pp. 394–400, doi: 10.1109/CLOUD60044.2023.00053.
5. H. Li et al., "RubbleDB: CPU-efficient replication with NVMe-oF," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2023, pp. 689–703.
6. F. Chowdhury et al., "I/O characterization and performance evaluation of BeeGFS for deep learning," in *Proc. 48th Int. Conf. Parallel Process.*, 2019, pp. 1–10.
7. O. Eytan, D. Harnik, E. Ofer, R. Friedman, and R. Kat, "It's time to revisit LRU vs. FIFO," in *Proc. 12th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2020, pp. 1–12.

DAEGYU HAN is a Ph.D. student at Sungkyunkwan University, Suwon, 16419, South Korea. His research interests include file systems and storage systems. Han received his B.S. degree in computer education from Sungkyunkwan University. Contact him at hdg9400@skku.edu.

SUNGHO MOON is a Ph.D. student in software at Sungkyunkwan University, Suwon, 16419, South Korea. His research interests include database systems, file systems, and RocksDB. Moon received his M.S. degree in software from Sungkyunkwan University. Contact him at sungho960919@gmail.com.

KYEUNGPYO KIM is a CTO at GlueSys Co., Ltd., Anyang, 14055, South Korea. His research interests include storage systems, file systems, and computer systems. Kim received his B.S. degree in information engineering from Sungkyunkwan University. Contact him at kpkim@gluesys.com.

SUNG-SOON PARK is a professor with the Department of Computer Science and Engineering, Anyang University, Anyang, 14028, South Korea, and the founder, president, and the CEO of GlueSys Co., Ltd., Anyang, 14055, South Korea. His research interests include storage systems, file systems, and high-performance computing. Park received his Ph.D. degree from Korea University. Contact him at sspark@gluesys.com.

BEOMSEOK NAM is a professor at Sungkyunkwan University, Suwon, 16419, South Korea. His research interests include data-intensive computing, database systems, and embedded system software. Nam received his Ph.D. degree in computer science from the University of Maryland. Contact him at bnam@skku.edu.

THEME ARTICLE: CACHE COHERENT INTERCONNECTS AND RESOURCE DISAGGREGATION TECHNIQUES

Containerized In-Storage Processing and Computing-Enabled Solid-State Drive Disaggregation

Miryeong Kwon, Donghyun Gouk, Eunjee Na, Jiseon Kim, Junhee Kim, Hyein Woo, Eojin Ryu, Hyunkyu Choi, Jinwoo Baek, and Hanyeoreum Bae, *Panmnesia, Inc., Daejeon, 34136, South Korea*

Mahmut Kandemir, *The Pennsylvania State University, University Park, PA, 16802, USA*

Myoungsoo Jung, *Korea Advanced Institute of Science Technology, Daejeon, 34141, South Korea*

In-storage processing (ISP) minimizes data transfer for analytics but faces challenges in adaptation and disaggregation. We propose DockerSSD, an ISP model that leverages operating system-level virtualization and lightweight firmware to enable containerized data processing directly on solid-state devices. The key features include Ethernet over Non-volatile Memory Express for network-based ISP management, and virtual firmware for secure, efficient container execution. DockerSSD supports disaggregated storage pools, reducing host overhead and enhancing large-scale services like large language model (LLM) inference. It achieves up to 2× better performance for input–output-intensive workloads, and 7.9× improvement in distributed LLM inference.

In-storage processing (ISP) is an emerging storage model widely adopted for data analytics, enabling efficient exploration of large datasets by minimizing data transfer overhead between the host and storage. Recent industry proposals advocate disaggregating ISP resources into storage arrays or nodes, significantly reducing server-side computational demands. To realize this vision, researchers have explored integrating data processing capabilities directly into solid-state drives (SSDs).[1,2,3,4,5,6] However, adapting SSDs to meet various application requirements remains a significant challenge.

The primary challenge in designing a flexible ISP model and storage disaggregation lies in creating practical runtimes and application programming interfaces (APIs), rather than in the computational capabilities of modern SSDs. Protecting vendor-sensitive data and intellectual property restricts the disclosure of internal hardware and firmware, making it essential for SSD vendors to provide ISP runtimes and APIs. However, evolving application requirements complicate the design process as substantial source-level modifications are needed to offload existing applications. New host-side daemons must also be developed to interact with and synchronize the disaggregated storage, further increasing complexity.

Introducing intelligence at the storage level introduces challenges in managing input–output (I/O) requests in block-device format while simultaneously processing data near the flash. This dual functionality can create vulnerabilities and resource protection issues, making SSDs unreliable for ISP tasks and user operations. For instance, SSDs lack awareness of file management and block layout, allowing host-side users to modify in-storage data during processing, potentially leading to unpredictable results. Given these limitations, ISP is currently confined to specific data processing applications, such as key value (KV) or

Digital Object Identifier 10.1109/MM.2025.3574261
Date of publication 30 May 2025; date of current version 24 December 2025.

object SSDs,[7,8] and has not been partially applied to resource disaggregation.

We present *DockerSSD*, an adaptable ISP model capable of diverse data processing near the flash without requiring source-level modifications. DockerSSD implements operating system (OS)-level virtualization within SSDs, enabling ISP to operate in a containerized manner. This design integrates storage intelligence into existing computing environments, accelerating decision making. Users can run algorithms without modifying them for vendor-specific runtimes or using a specialized toolchain. This enables individual computational SSDs to function independently, supporting their disaggregation from the host and formation into a computing-enabled storage pool.

Although OS-level virtualization promotes ISP adoption, two challenges arise when integrating ISP with containers. First, containers, being self-contained and independent, require a specialized interface for managing flash-based data processing, compatible with the storage and network stacks. Second, their encapsulated nature demands a firmware redesign to dynamically construct containers and provide an execution environment. To address these, we propose a communication framework for ISP that uses Ethernet and firmware to autonomously download and execute Docker images. We also redesign firmware to virtualize the ISP environment, enabling data processing without application-level changes. This approach simplifies host management of disaggregated storage and supports large-scale pretrained model services like large language models (LLMs).

A NOVEL COMMUNICATION METHOD FOR ISP

We propose *Ethernet over Non-volatile Memory Express* (*NVMe*) (Ether-oN), a kernel driver that enables network-based ISP management and direct communication between host users and SSDs. Ether-oN introduces asynchronous upcalls and packet overriding within the NVMe protocol to support Ethernet networking. This allows our ISP model to leverage command-line interfaces from Docker. With Ether-oN, users can supply data, monitor ISP statuses, and retrieve results directly from the SSD, enabling real-time, on-demand data analysis. As Ether-oN requires no modifications to existing interfaces, it parallelizes the network and block I/O operations.

CONTAINERIZING ISP WITH FIRMWARE

We present *virtual firmware* (*virtual-FW*), a lightweight firmware stack that integrates minimal OS functionality and a container environment into the SSD. By emulating system calls on bare-metal hardware, virtual-FW maintains ISP system call execution costs comparable to function management costs. It creates *ISP-containers* from existing Docker images and executes them without the overhead of a full OS. Virtual-FW ensures secure and portable processing by employing NVMe namespace and memory isolation to safeguard SSD resources.

DISAGGREGATING COMPUTING-ENABLED STORAGE

Using Ether-oN and virtual-FW, DockerSSD is equipped with its network Internet Protocol (IP), enabling uninterrupted computation and the creation of a computing-enabled storage pool. This disaggregated pool significantly reduces host-side computation overhead and provides distributed data processing capabilities, improving a wide range of large-scale services. We demonstrate its effectiveness through a case study on distributed inference for diverse LLM models. The storage pool achieves substantial performance gain by eliminating data movement and minimizing memory requirements during processing.

We validate DockerSSD using a PCIe SSD prototype with an NVMe controller[9] and a multicore processor on a 16-nm FinFET (field-effect transistor) field-programmable gate array (FPGA). DockerSSD outperforms host system and leading ISP models[5,6] by 1.3 and 1.8×, respectively. In addition, the compute-enabled storage pool enhances LLM performance by an average of 7.9×.

CHALLENGES IN ISP

Various ISP models have been proposed to offload operations to storage and thereby reduce data-movement overhead. For example, Summarizer[3] and KAML,[4] representative examples in this field, handle data filtering and KV database operations, respectively, by offloading specific kernels to the storage side. Willow[5] and Biscuit[6] address broader user requirements, allowing users to compile and offload kernels dynamically via firmware APIs.

Despite these and other efforts, such ISP models have not seen wide adoption. One major reason is that existing models often require static kernels or vendor-specific APIs for offloading, significantly reducing user convenience. Specifically, five key challenges must be overcome to improve these limitations: manual ISP implementation, disregard for file layout, kernel context switching, device reliance, and data vulnerability. Addressing these challenges demands an ISP

model that supports execution and environment independence, and flexible virtualization for diverse applications.

Performance Impact Assessment

We evaluate a revised iteration of the programmable-ISP model (`P.ISP`).[5,6] Detailed workload characteristics and evaluation environments are discussed in the "Evaluation" section. Figure 1(a) illustrates a breakdown of ISP execution times into three components: ISP computation latency (`compute`), storage back-end delay (`storage`), and host-to-ISP communication/synchronization overhead (`communicate`). For comparison, we also evaluate a host-only system (`host`). Our findings reveal that `storage` accounts for 38% of the total execution time, highlighting the potential for reducing application latency. By processing data in storage, `P.ISP` reduces `storage` latency by 50% compared to `host`, effectively mitigating data movement overhead. However, this improvement is offset by a 1.4# increase in overall end-to-end latency, primarily due to `communicate`, which constitutes 43% of `P.ISP` latency.

Addressing these challenges requires making ISP models host independent and enabling autonomous execution. We believe that, by eliminating communication overhead, ISP can achieve better performance. We also believe that virtualizing the ISP model can enable it to function as a secure, portable sandbox, paving the way for broader ISP adoption.

ISP CONTAINERIZATION

Unlike conventional ISPs that rely on static kernels or tasks, DockerSSD establishes virtual ISP environment, enabling the host to process data near flash using containerized applications, called *ISP-containers*. This section explores how DockerSSD achieves ISP containerization through firmware, addressing challenges like manual ISP implementation, context switches, and data vulnerability. By managing back-end media to reduce simultaneous access and leverage firmware-level virtualization, DockerSSD eliminates device reliance and file layout limitations, ensuring that containers operate independently across diverse systems.

High-Level Overview of DockerSSD

To implement containerization in storage, DockerSSD addresses three technical challenges. First, it requires a new communication interface that is compatible with both existing storage and network stacks for effective ISP-container management. Second, the ISP model must ensure consistency across a computing-enabled storage pool, enabling independent operation of all components. Third, the underlying firmware must recognize container structures and execute them with minimal overhead.

Figure 1(b) illustrates the architecture of DockerSSD. To overcome these challenges, it integrates two key components: 1) a system driver introducing

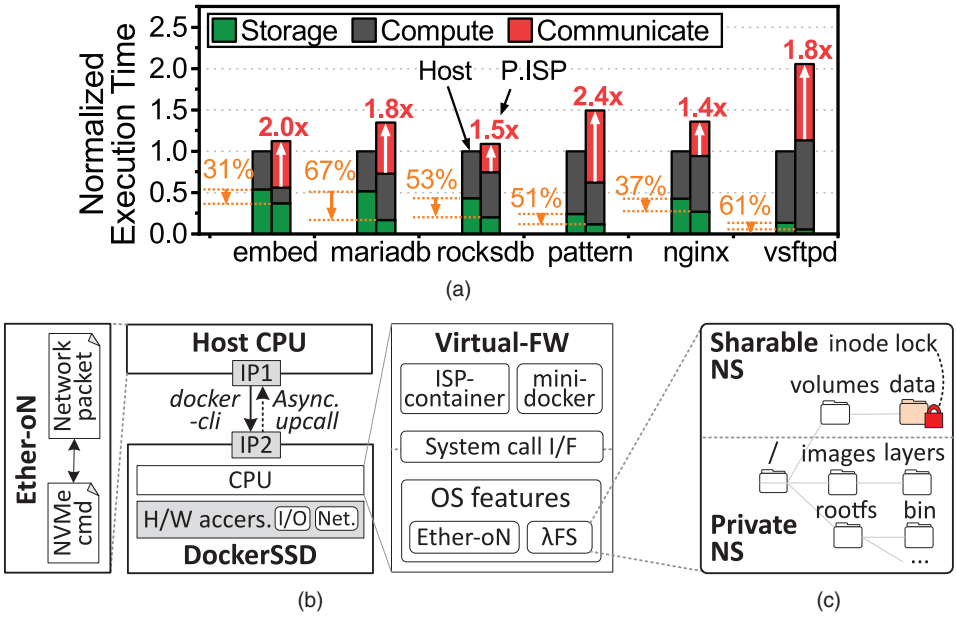


FIGURE 1. Motivation and high-level view of DockerSSD. (a) Performance impact analysis. (b) ISP containerization. (c) Media management. mFS: lambda filesystem; IF: interface; NS: namespace.

a novel ISP management interface, *Ether-oN*, and 2) a firmware environment equipped with Docker stack, *virtual-FW*. *Ether-oN* facilitates socket-based Ethernet communication over PCIe by virtualizing NVMe. It achieves this through a driver that supports network-to-storage packet translation and asynchronous upcalls. By assigning individual IP addresses to each endpoint, Ether-oN enables users to issue ISP-related requests to virtual-FW via `docker-cli`. *Virtual-FW* incorporates minimal OS features into SSDs, emulating system call interfaces on bare-metal hardware. It includes *mini-docker*, which implements core functions of Docker stack, enabling firmware-level ISP containerization.

In this study, we demonstrate how virtual-FW and Ether-oN allow users to run full-scale applications near flash using familiar Docker commands, without relying on vendor-specific interfaces. Furthermore, Ether-oN supports all essential Ethernet functionalities, allowing storage components to operate as independent nodes within a computing-enabled storage pool, communicating seamlessly with the host. Although virtual-FW is optimized to minimize the ISP-container execution cost through system call emulation and firmware-level functionality, ISP-containers themselves can generate I/Os and network requests for data processing and communication as needed.

Back-End Media Management

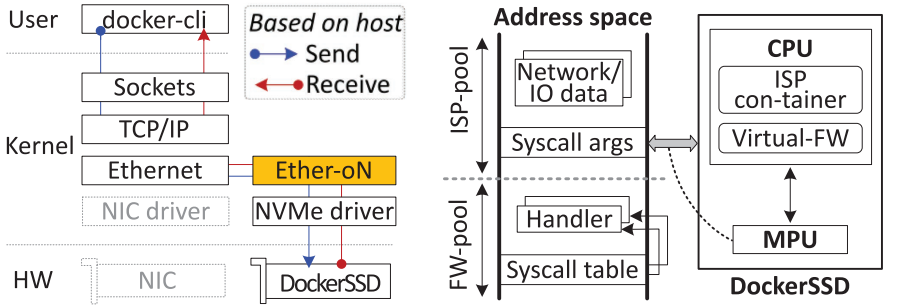
DockerSSD introduces the *lambda filesystem* (λFS) for efficient back-end flash management [Figure 1(c)]. To handle file concurrency and address potential vulnerabilities, λFS partitions the media into two NVMe namespaces, supported by the NVMe subsystem: 1) the private namespace (*private-NS*) and 2) the sharable namespace (*sharable-NS*). The private-NS is isolated from the host, while the sharable-NS is accessible to both the host and ISP-containers. λFS allocates private-NS for container- and OS-level virtualization runtimes. Thus, ISP-related contents that need protection, such as image layers (`/images/`) and container data (`/rootfs/`), are kept invisible to the users. Conversely, the sharable-NS is used for data that the host needs to place/retrieve and ISP-container processes in storage.

To ensure appropriate file lock, λFS employs a new in-memory data structure: the `inode` lock. This lock synchronizes with the host's `inode` cache via Ether-oN when an ISP-container binds a host FS directory or file to λFS. The file is accessible only if the `inode` reference count is zero, resolving concurrent access issues. When the ISP-container gets file access permission, virtual filesystem (VFS) invalidates its `inode` cache, referring to the storage's latest information. Note that when the host and an ISP-container access the same partition concurrently, the host may retain a stale `inode` in the VFS. This can lead to file system corruption or render files/directories updated by the ISP-container inaccessible. To address this, DockerSSD synchronizes updates between the host and the ISP-container, even without host involvement.

Ether-oN

The Docker stack and container interface rely on Ethernet, making it a natural choice for ISP-related service communication. We overlay standard socket-based networking onto NVMe protocol (Ether-oN). Figure 2(a) illustrates Ether-oN's architecture. Host applications communicate with DockerSSD using a conventional network stack, interacting with virtual-FW via Ethernet. Ether-oN establishes an intranet between the host and DockerSSD, translating Ethernet packets into NVMe commands sent to DockerSSD. This introduces two challenges. First, NVMe lacks an in-bound request-handling mechanism. Second, host users need seamless access to DockerSSD via network and storage stacks for ISP and block I/O services. To address these issues, we introduce two NVMe vendor-specific commands: transmit and receive frames. When Ether-oN driver receives an Ethernet request, it extracts the packet buffer, allocates a kernel page, copies the buffer to the kernel page. The Ether-oN driver then creates an NVMe command with the kernel page, sets the operation code to transmit, and submits the command to the driver.

To allow DockerSSD to request host services, we implement an asynchronous upcall mechanism using preallocated NVMe commands. During kernel initialization, Ether-oN pre-submits a set of NVMe commands, each configured as a receive frame, to the NVMe submission queue (SQ). For every preallocated command, Ether-oN assigns a kernel page and inserts a reception code. DockerSSD holds these commands until an ISP-container sends an Ethernet frame to the host. At that point, DockerSSD copies the frame's buffer to the corresponding kernel page and completes the outstanding command. Upon completion, Ether-oN translates the NVMe command into an Ethernet frame and delivers it to the network stack. To maintain communication, Ether-oN immediately submits a new receive frame to DockerSSD.



Handler (# syscall)	Category	Example
Thread Handler (65)	Process management	`fork exit`
	Memory management	`brk mmap`
	Inter-Process comm.	`pipe mq_open`
	Lock & signal mgmt.	`futex`
I/O Handler (43)	File/dir mgmt.	`openat mkdir`
	File I/O & link	`read symlink`
	Permission	`chmod chown`

(c)

Category	CLI command
Image management	`docker rmi`
	`docker pull`
Container life cycle management	`docker create`
	`docker run`
	`docker start`
	`docker stop`
	`docker restart`
	`docker kill`
	`docker rm`

(d)

FIGURE 2. Overview of Ether-oN and virtual-FW. (a) Ether-oN's interface. (b) Virtual-FW. (c) System calls. (d) Docker commands. CLI: command line interface; HW: hardware; mgmt.: management; NIC: network interface card; MPU: microprocessing unit.

Docker-Enabled Firmware

Existing firmware lacks the flexibility required for Ethernet-based ISP computing and resource management. To address this, virtual-FW integrates essential OS features and a container environment into the current I/O service path. Figure 2(b) illustrates how the ISP service path integrates into the existing firmware-level I/O service path. DockerSSD implements three handlers for OS features: thread, I/O, and network management. The thread handler manages its bare-metal dynamic random-access memory (DRAM) in page-granular partitions: FW-pool and ISP-pool. Execution is distinguished by DockerSSD's CPU modes, with privileged mode required for FW-pool access, enforced by the memory protection unit. This safeguards virtual-FW while eliminating the need for data copying between pools as privileged mode allows virtual-FW to access the ISP-pool directly, avoiding mode-switching overhead. The I/O handler processes only I/Os generated by ISP-containers for data stored in DockerSSD, without requiring heavy block management layers. It implements λFS and caches path-walking mappings for faster access. The network handler manages communication through TCP/IP features, including establishing channels and handling network packets.

Virtual-FW emulates system calls to enable seamless container execution within storage, using lightweight function wrappers. Figure 2(c) summarizes the types and quantities of system calls emulated, and the corresponding handler. This minimizes memory overhead and call latency by removing unnecessary system functions and lowering the impact of running containers on SSDs. The container environment within DockerSSD focuses on storing, running, and managing containers near flash, rather than supporting the full Docker functionalities. Thus, virtual-FW's mini-docker supports 11 essential Docker commands (out of 106), as summarized in Figure 2(d).

RESOURCE DISAGGREGATION

With ether-oN and virtual-FW, DockerSSD gains its IP address and can run containerized applications as an independent node. This enables the creation of a computing-enabled storage pool by disaggregating DockerSSDs from their dedicated hosts. As illustrated in Figure 3(a), DockerSSDs can form an array pool that is connected via one or more PCIe switches. Multiple arrays can be integrated into a cluster using a switch tray or chassis unit for larger-scale deployments. Despite the PCIe connectivity, Ether-oN assigns each DockerSSD a unique IP address by converting

Ethernet protocols to NVMe (and vice versa). This allows the host to assign specific applications to any node in the array or cluster and monitor them through mini-docker logs.

Computing-Enabled Storage Pool

There are two primary methods for offloading computing to the computing-enabled storage pool. The first involves offloading independent applications across DockerSSDs in the pool, allowing each node to operate autonomously. The second, preferred in this work, groups multiple DockerSSDs into a distributed computing system. For distributed computing, DockerSSDs leverage frameworks such as `docker-compose` or Kubernetes to orchestrate containers across nodes efficiently.

As a use case for our computing-enabled storage pool, we set up a distributed inference system for a large-scale pretrained model. Specifically, we implement an LLM that is well suited to this scenario. Due to memory and computational limitations, single nodes or devices cannot manage inference tasks for trillion-scale model parameters. As the model size increases, LLMs employ distributed inference techniques, leveraging methods such as pipeline parallelism, data parallelism, and tensor parallelism. In pipeline parallelism, DockerSSDs allocate different layers of the model across multiple ISP devices. For data parallelism and tensor parallelism, the model divides data either based on device awareness within the storage pool or along dimensionality, ensuring efficient utilization of the distributed infrastructure. The TorchServe model-serving platform can help our distributed inference test, which utilizes Kubernetes for container execution.

Distributed Inference With DockerSSDs

As shown in Figure 3(b), LLMs generate the next token based on a given set of tokens. Each iteration reuses the tokens calculated thus far as input for the next step, progressively expanding the sequence as the inference continues. The LLMs consist of multiple stacked encoders, each composed of an attention layer and a

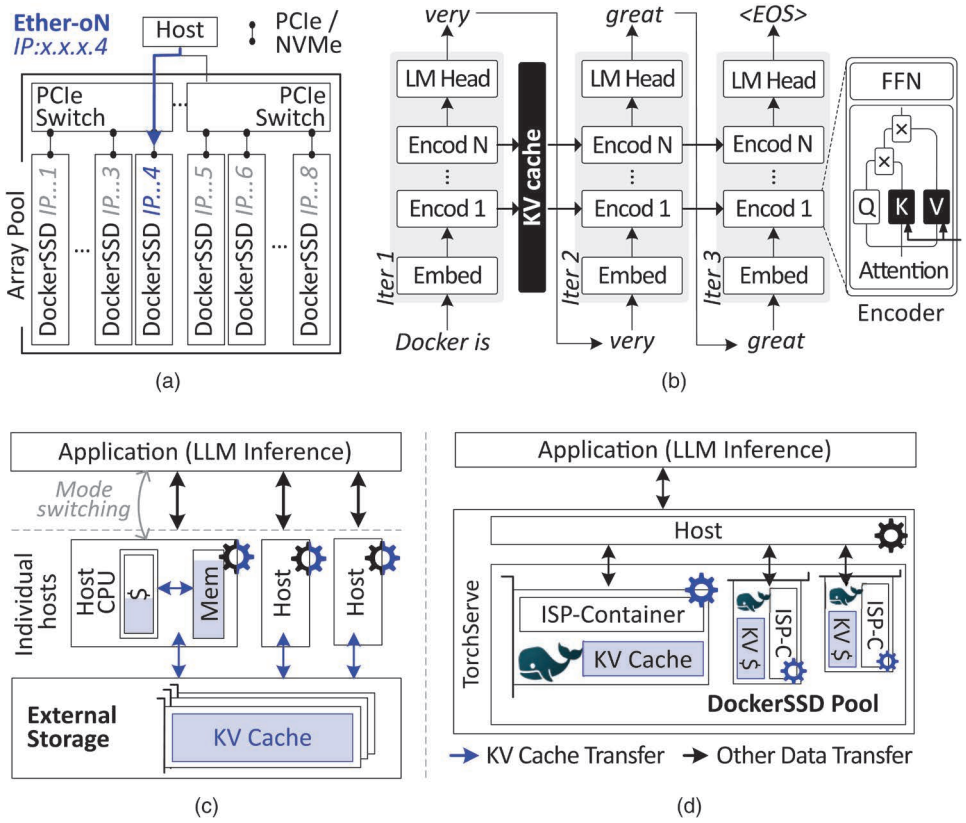


FIGURE 3. DockerSSD disaggregation and LLM distributed inference. (a) Array pool. (b) LLM architecture. (c) Individual hosts. (d) DockerSSD pool. EOs: end-of-sequence; FFN: feed-forward network; LM head: language model head; PCIe: Peripheral Component Interconnect Express.

feedforward network layer. The attention layer calculates the importance of each token relative to others using the key (K) vector, which represents the significance of the token, and the value (V) vector, which encodes the actual semantic information of the token. Because LLMs reuse tokens during each inference iteration, recalculating the K and V vectors for the same tokens in every iteration introduces unnecessary computational overhead. To address this, modern LLMs implement *KV* caching, which stores the computed K and V vectors in memory for reuse in subsequent iterations, significantly reducing computational burden.

Figure 3(c) and (d) compares LLM inference performance when served from individual hosts versus our computing-enabled storage pool in a distributed manner, respectively. KV caching makes LLM inference a memory-intensive application, often requiring additional memory capacity through external storage. In distributed environments, this can lead to cache pollution, frequent mode switches, and unnecessary data copies on the hosts. By contrast, each ISP device in our storage pool manages KV caching directly on its flash storage, avoiding these inefficiencies. In addition, distributed inferences handled by TorchServe utilize the storage pool effectively, enabling the processing of much longer token sequences and providing greater context for each service. This approach enhances the overall capacity and scalability of the LLM system.

EVALUATION

Prototype and Methodology

We built a DockerSSD prototype using a 16-nm FinFET FPGA. We adopted NVMe hardware IPs[9] to include a multicore processor, integrating Register Transfer Language (RTL) into DockerSSD's front end. The prototype includes a queue manager, I/O engine, and PCIe controller for NVMe tasks. Six RISC-V in-order cores run virtual-FW, connected via AXI/TileLink buses. For the back end, two DDR4 controllers emulate flash modifying a multichannel timing model.[10] We modified 3.4 K lines of code (LOC) for Ether-oN/λFS, and 7.5 K LOC for firmware and storage modules (EXT4-based). As a result, virtual-FW reduced the Linux binary size by 83.4×, making it suitable for embedded processors. Because no device can accommodate all hardware/software configurations or ISP communication methods, we evaluated various ISP models using the gem5 full-system simulator.[11] The simulation framework has been cross validated with our hardware RTL and synthesized prototype back end, and integrated into a cycle-accurate SSD simulator.[10] The `host` has a 3.8-GHz CPU and 64-GB DDR4, while storage includes an NVMe SSD with 48 multi-level cell (MLC) flashes across 12 channels. The SSD uses a 2.2-GHz processor and 2 GB of DRAM.

Data Processing Models, Benchmarks, and Workloads

We designed six data processing models. The `host` model represents a baseline non-ISP setup. Two programmable ISP models, `P.ISP-R/V`, were implemented using offloading methods from prior studies.[5,6] `P.ISP-R` uses as its interface,[5] while `P.ISP-V` employs NVMe vendor-specific commands to minimize interface overhead.[6] Both models assume that most ISP functions are offloaded, except for system-specific modules. `P.ISP-R/V` only accesses LBAs when ISP kernels require a new file, reducing overhead. We also evaluated three DockerSSD variations: `D-Naive` uses a separate processor running an ISP-container and controller complexes running firmware with a full Linux. `D-FullOS` integrates DockerSSD's hardware, enabling the ISP-container and firmware to run on the same complex. `D-VirtFW` replaces Linux in `D-FullOS` with lightweight virtual-FW. We selected six benchmarks, encompassing 13 unique workloads, to evaluate performance: Deep Learning Recommendation Model's (DLRM) embedding operations ("embed"), relational database ("mariadb"), KV store ("rocksdb"), text mining ("pattern"), web server ("nginx"), and file server ("vsftpd"). Each workload was executed 10 times. Figure 4(a) provides details of the tested workloads.

Overall Performance Comparisons

Figure 4(b) shows the latency of various ISP models, normalized to `D-VirtFW`. For clarity, the performance is categorized into six: network operation times (*network*), Kernel context switches (*kernel-ctx*), LBA set handshaking (*LBA-set*), SSD access times (*storage*), System call and OS stack latency (*system*), and ISP kernel latency (*compute*).

Programmable-ISP

Although `P.ISP-R/V` eliminates data transfer overhead for processing, its overall performance is worse than `host`'s, primarily due to *kernel-ctx* and *LBA-set* overheads. However, `P.ISP-R/V` outperforms `host` in rocksdb-read and nginx-filedown scenarios. These workloads involve frequent get operations, where system and data movement significantly impact latency. As `P.ISP-R/V` bypasses OS and system call overhead by running ISP kernels on the device's bare-metal system, they achieve better performance. Note that `P.ISP-V` exhibits 13.7% lower latency than `P.ISP-R` by avoiding Remote Procedure Call and eliminating the need for network responses.

ISP-Container With OS

Although D-Naive/D-FullOS removes communication overhead (*kernel-ctx + LBA-set*), they face challenges with computational inefficiencies and data movement due to a full software stack; D-FullOS incurs a 9.3% higher latency than P.ISP-V due to the overhead that handles system calls and the OS with limited computing resources. D-Naive exhibits a 12.8% slowdown compared to D-FullOS, requiring frequent data transfers between the ISP-container processor and controller complexes.

FW-Level Containerization (DockerSSD)

D-VirtFW combines the advantages of full-fledged application execution while avoiding the software stack overhead of D-FullOS and the hardware inefficiencies of D-Naive. It delivers significant performance improvements, outperforming P.ISP-R/V, D-Naive, and D-FullOS by 1.6, 1.8, and 1.6×, respectively. By accessing back-end flash through mFS, D-VirtFW eliminates the need for *LBA-set*, achieving an 8.4% latency reduction compared to P.ISP-R/V. Moreover, execution parameters are prepackaged in rootfs, removing *kernel-ctx* overhead and improving ISP performance by 30.9% over P.ISP-R/V.

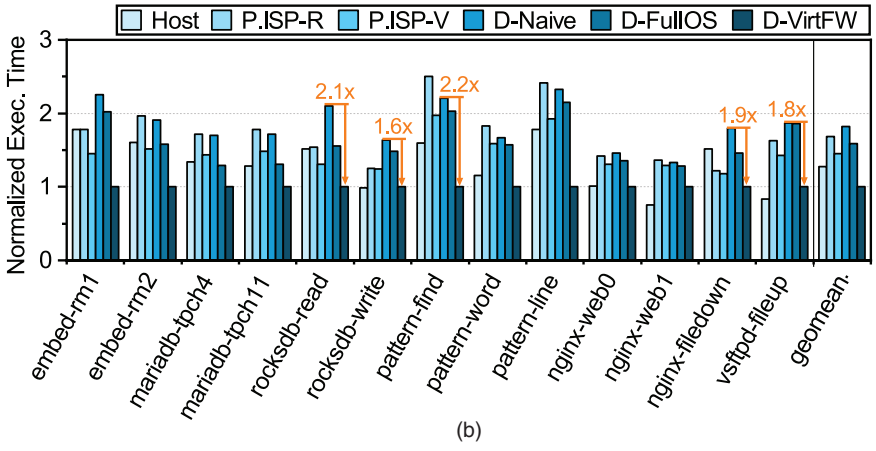
D-VirtFW combines the advantages of full-fledged application execution while avoiding the software stack overhead of D-FullOS and the hardware inefficiencies of D-Naive.

Disaggregated Computing Storage for LLMs

In this extended evaluation, we investigate whether a computing-enabled storage pool powered by DockerSSDs can effectively serve distributed LLM inference. To test this, we select eight LLMs with diverse

Program	Workload	I/O size	I/O count	# sys-calls	# path walk	# files opened	# TCP packet	Exec. Time
embed	rm1	1.3GB	317K	1.3M	9K	260	0	8s
	rm2	5.8GB	1.4M	1.7M	9K	320	0	24s
mariadb	tpch4	17.1GB	1.1M	1.1M	37K	250	160	25s
	tpch11	6.2GB	400K	361K	38K	260	190	8s
rocksdb	read	4.1GB	431K	1.1M	9K	1.2K	0	14s
	write	18.5GB	24K	285K	9K	3.6K	0	24s
pattern	find	2.4GB	381K	1.8M	359K	352K	0	11s
	line	1.7GB	262K	1.7M	476K	235K	0	11s
	word	2.1GB	340K	2.2M	618K	307K	0	10s
nginx	web0	7.5GB	126K	665K	126K	4.4K	543M	9s
	web1	0.9GB	50K	344K	109K	2K	154K	3s
	filedown	13.5GB	109K	30K	1K	40	155K	6s
vsftpd	fileup	12.1GB	93K	5.4M	127K	115K	1.2M	2s

(a)



(b)

FIGURE 4. Overall performance analysis. (a) Workload characteristics. (b) Latency comparisons. exec.: execution.

architectures and varying model sizes: lamda-137B, gpt3-175B, jurassic-178B, pangu-200B, gopher-280B, turing-530B, palm-540B, and megatron-1T. These models, which typically require distributed systems with tens to hundreds of devices, are evaluated using storage pools composed of 16–128 DockerSSDs.

Resource Disaggregation Models

We prepared three disaggregated configurations based on the location that serves distributed inferences (host or DockerSSD) and the presence of a KV cache (NoCache or cache). In `H-NoCache`, distributed inferences are performed across multiple hosts (matching the number of DockerSSDs: 16–128), each with 64 GB of local DRAM. This configuration lacks a KV cache due to insufficient DRAM capacity to store KV vectors, but all the data reside in local DRAM. In `H-Cache`, each host uses external storage (400-GB SSD) combined with DRAM via Linux swap to support the KV cache. In addition to caching KV data, all other data are also maintained in memory. `D-Cache` employs DockerSSDs instead of hosts, with each DockerSSD (400-GB storage capacity).

Methodology and Benchmarks

We used an open source simulator to evaluate distributed inference performance for LLMs by varying parallelization techniques, optimization methods, and system configurations.[12] As the existing simulator supports only basic LLM structures and lacks KV cache functionality, we developed an analytical model for the KV cache and integrated it into the simulator. We also enhanced it to evaluate performance under different degrees of parallelism (data, tensor, and pipeline) based on GPU counts and batch sizes, identifying the optimal configuration by selecting the scenario with the shortest execution time.

Figure 5(a) presents the optimal parallelism values for each disaggregation model with a sequence length of 32 K and a batch size of one per GPU. In `H-NoCache` and `D-NoCache`, where per-layer computation is heavy, pipeline parallelism minimizes inference time. By

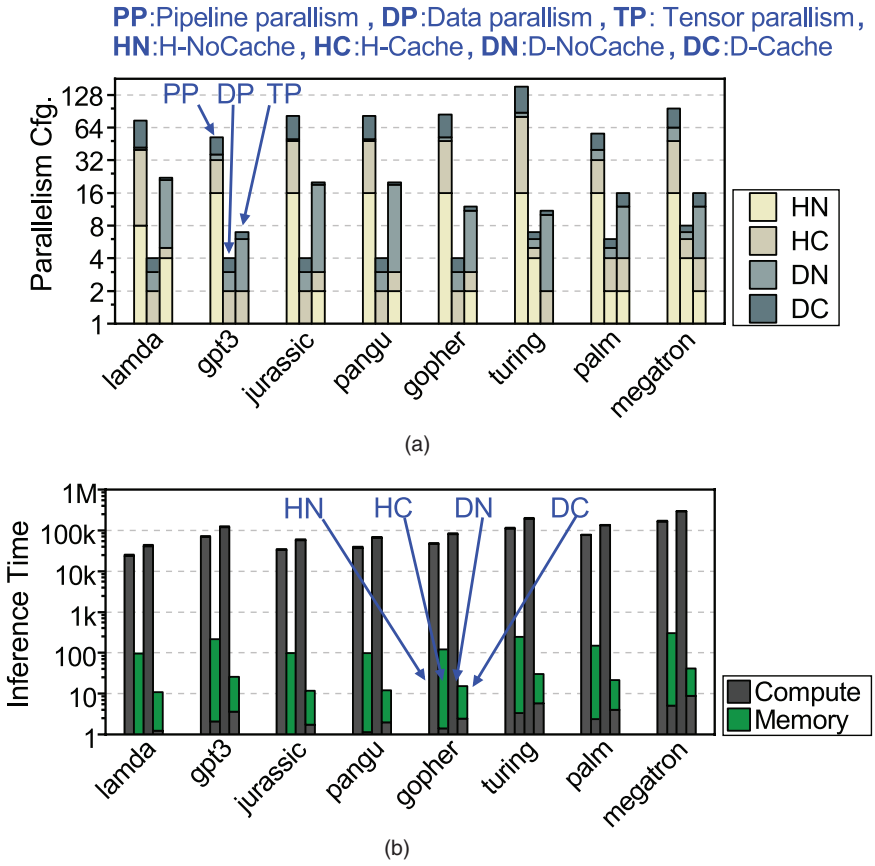


FIGURE 5. Inference performance analysis. (a) Optimal configuration. (b) Inference breakdown. cfg.: configuration. PP: pipeline parallelism; DP: data parallelism; TP: tensor parallelism; HN: H-NoCache; HC: H-cache; DN: D-NoCache; DC:D-cache.

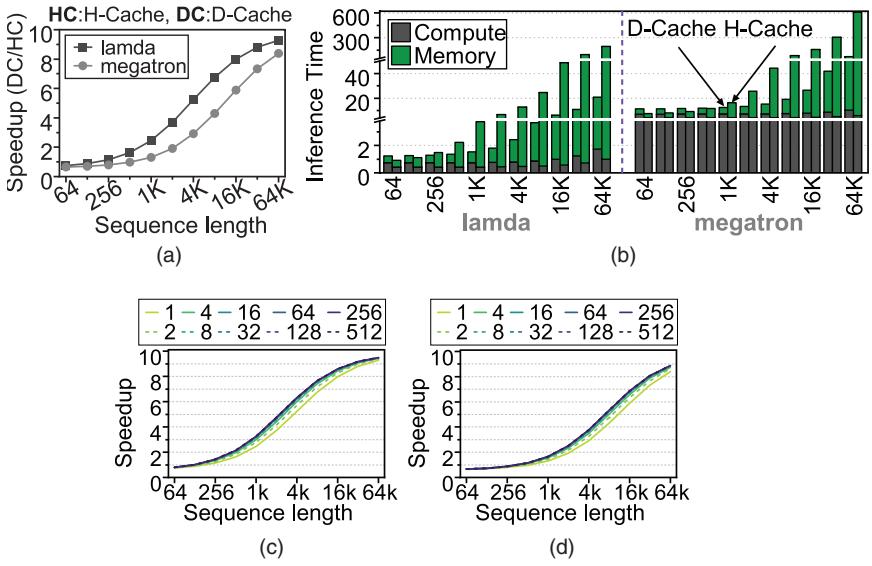


FIGURE 6. Sensitivity test. (a) Model size. (b) Breakdown (sequence length). (c) Lambda (137 B). (d) Megatron (1 T).

contrast, H-Cache and D-Cache leverage KV cache to reduce per-layer computation, making tensor parallelism the most efficient for minimizing total inference time.

Performance Analysis

Figure 5(b) breaks down distributed LLM inference performance into two components: compute, which measures the time spent on core operations like matrix and vector multiplications, and memory, which includes the time required to read input data and write output data during inference. The comparison between H-NoCache and D-NoCache highlights the performance difference based on where distributed inference is performed. DockerSSD achieves distributed inference with only a 1.7× performance degradation compared to the host. This gap is primarily due to the computational capability difference between the host and DockerSSD.

The differences between H-NoCache/H-Cache and D-NoCache/D-Cache demonstrate the significant benefits of KV cache techniques. Expanding memory capacity to enable KV caching greatly boosts performance. Specifically, H-Cache achieves a 421× performance gain over H-NoCache, while D-Cache achieves a remarkable 4.6 K× improvement over D-NoCache. The performance improvement of DockerSSD comes from its ability to access flash memory as local memory, unlike the host, which relies on software-based memory extensions like swap. This advantage makes DockerSSD particularly well suited for KV cache-based methods, which replace computation with memory storage. As a result, D-Cache outperforms H-Cache by 7.9× and H-NoCache by 3.2 K× in distributed inference.

Analysis With Varying Sequence Lengths

The KV cache size varies significantly with sequence length. Without a KV cache, inference for a sequence length n requires $O(n^2)$ operations. Using an $O(n^2)$ -sized KV cache allows reuse of previously computed KV vectors, reducing computational complexity to $O(n)$. This reduction becomes more pronounced as sequence length increases, emphasizing DockerSSD's advantage in efficiently accommodating large KV caches for improved LLM distributed inference. We evaluated DockerSSD's benefits across varying sequence lengths using lambda (smallest model) and megatron (largest model). Figure 6(a) shows that for longer sequences, D-Cache achieves faster inference than H-Cache. For shorter sequences [Figure 6(b)], computation time dominates, and DockerSSD's slower processing speed (2.2 versus 3.8 GHz) results in roughly 60% of host performance. As sequences grow, memory time becomes more significant. Beyond a certain point, DockerSSD's reduced memory overhead outweighs its computational disadvantage. This crossover occurs at a sequence length of 256 for lambda and 1024 for megatron, where DockerSSD begins outperforming the host. With longer sequences, the speedup converges to roughly 9.5×, reflecting the maximum benefit from eliminating the swap overhead in H-Cache. Smaller models exhibit greater speedup

for the same sequence length, as larger models allocate more time to multilayer perceptrons, reducing the impact of KV cache improvements on attention layers.

Sensitivity Test for Batch Sizes

The KV cache size is also influenced by batch size. Figure 6(c) and (d) presents a sensitivity analysis for varying batch sizes, using lambda and megatron as representatives, as in the previous evaluation. Although larger batch sizes can enhance a processing speed that leverages GPU parallelism, the increased memory bandwidth demand from KV caching may create bottlenecks, potentially reducing overall performance. Consequently, smaller batch sizes are often preferable for longer sequence lengths. As the batch size increases from one to 512 for the same sequence length, KV cache memory usage grows linearly with the batch size. However, `D-Cache` shows only modest improvement over `H-Cache`, with a maximum speedup of 1.3× for lambda and megatron.

CONCLUSION

We introduced *DockerSSD*, a novel ISP model that integrates containers and lightweight firmware to enable efficient, containerized data processing directly within SSDs. Leveraging *ether-oN* and *virtual-FW*, DockerSSD addresses challenges in ISP adaptation and disaggregation, enabling high-performance, computing-enabled storage pools. Our evaluations demonstrate significant reductions in host overhead and enhanced distributed inference performance for large-scale services. These results position DockerSSD as a transformative solution for scalable storage systems in disaggregated infrastructures.

ACKNOWLEDGMENTS

The work is supported in part by National Research Foundation of Korea 2021R1A2C4001773, Institute of Information & communications Technology Planning & Evaluation's 2021-0-00524 and 2022-0-00117, RS-2023-00221040, IITP-2024-RS-2023-00256472, RS-2024-00460762, RS-2025-02214652, RS-2025-02214654, RS-2025-02263080, Korea Institute for Advancement of Technology's P0027923 and P0028225, Ministry of SMEs and Startups Technology development Program's RS-2023-00303967, and the Korea Advanced Institute of Technology's IC Design Education Center. Miryeong Kwon and Donghyun Gouk equally contributed equally to this work. This work is protected by one or more patents.

REFERENCES

1. Y. Kang, Y.-s. Kee, E. L. Miller, and C. Park, "Enabling cost-effective data processing with smart SSD," in *Proc. IEEE 29th Symp. Mass Storage Syst. Technol. (MSST)*, Piscataway, NJ, USA: IEEE Press, 2013, pp. 1–12, doi: 10.1109/MSST.2013.6558444.
2. Y.-S. Lee, L. C. Quero, Y. Lee, J.-S. Kim, and S. Maeng, "Accelerating external sorting via on-the-fly data merge in active SSDs," in *Proc. 6th USENIX Workshop Hot Topics Storage File Syst. (HotStorage)*, 2014, p. 14.
3. G. Koo et al., "Summarizer: Trading communication with computing near storage," in *Proc. 50th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2017, pp. 219–231.
4. Y. Jin, H.-W. Tseng, Y. Papakonstantinou, and S. Swanson, "KAML: A flexible, high-performance key-value SSD," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2017, pp. 373–384, doi: 10.1109/HPCA.2017.15.
5. S. Seshadri et al., "Willow: A user-programmable SSD," in *Proc. 11th USENIX Conf. Operating Syst. Des. Implementation (OSDI)*, 2014, pp. 67–80.
6. B. Gu et al., "Biscuit: A framework for near-data processing of big data workloads," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, 2016, pp. 153–165, doi: 10.1109/ISCA.2016.23.
7. R. Suzuki, "Samsung's 12nm-class automotive LPDDR5X: DRAM for safety-critical centralized automotive systems," *Samsung*, May 12, 2025. [Online]. Available: https://semiconductor.samsung.com/news-events/tech-blog/samsungs-12nm-class-automotive-lpddr5x-dram-for-safety-critical-centralized-automotive-systems/
8. "LS2088A intelligent-SSD card." Argonboards. [Online]. February 26, 2025. Available: https://www.argonboards.com/ls2088a-intelligent-ssd-card
9. M. Jung, "OpenExpress: Fully hardware automated open research framework for future fast NVMe devices," in *Proc. 2020 USENIX Conf. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2020, 649–656.
10. M. Jung et al., "SimpleSSD: Modeling solid state drives for holistic system simulation," *IEEE Comput. Archit. Lett.*, vol. 17, no. 1, pp. 37–41, Jan./Jun. 2018, doi: 10.1109/LCA.2017.2750658.
11. N. Binkert et al., "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011, doi: 10.1145/2024716.2024718.
12. M. Isaev, N. McDonald, L. Dennison, and R. Vuduc, "Calculon: A methodology and tool for high-level codesign of systems and large language models," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2023, pp. 1–14, doi: 10.1145/3581784.3607102.

MIRYEONG KWON is the chief strategy officer in the Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea. Her research interests include edge computing, on-device artificial intelligence, and link solutions including CXL, UALink, and Ethernet. Kwon received her Ph.D. degree from the School of Electrical Engineering from the Korea Advanced Institute of Science Technology. She is a Member of IEEE. Contact her at mkwon@panmnesia.com.

DONGHYUN GOUK is the chief architect officer in the Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include computing switches/networks, artificial intelligence, and link solutions including CXL, UALink, and Ethernet. Contact him at donghyun@panmnesia.com.

EUNJEE NA is an assistant director in the Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea. Her research interests include disaggregated systems, and link solutions including CXL, UALink, and Ethernet. Contact her at ejna@panmnesia.com.

JISEON KIM is an assistant director in the Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea. Her research interests include edge computing, on-device artificial intelligence, and link solutions including CXL, UALink, and Ethernet. Contact her at jskim@panmnesia.com.

JUNHEE KIM is a senior professional in the Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea. Her research interests include machine learning acceleration, computing switches/networks, and artificial intelligence infrastructure. Contact her at jhkim@panmnesia.com.

HYEIN WOO is a senior professional in the Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea. Her research interests include machine learning acceleration, computing switches/networks, and link solutions including CXL, UALink, and Ethernet. Contact her at hiwoo@panmnesia.com.

EOJIN RYU is an assistant director in the Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include GPU/solid-state device integration, edge computing, and link solutions including CXL, UALink, and Ethernet. Contact him at neo21top@camelab.org.

HYUNKYU CHOI is an assistant director in the Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include network/storage, edge computing, and link solutions including CXL, UALink, and Ethernet. Contact him at hkchoi@panmnesia.com.

JINWOO BAEK is a senior professional in the Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea. Her research interests include machine learning acceleration, computing switches/networks, and link solutions including CXL, UALink, and Ethernet. Contact her at jwbaek@camelab.org.

HANYEOREUM BAE is a director in the Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include new storage, zoned namespace, and link solutions including CXL, UALink, and Ethernet. Contact him at hyr.bae@camelab.org.

MAHMUT KANDEMIR is a distinguished professor of computer science and engineering at The Pennsylvania State University, University Park, PA, 16802, USA. His research interests include optimizing compilers, runtime systems, mobile systems, embedded systems, input–outputs and high-performance storage, and nonvolatile processors and memory; and the latest trends in public cloud services. Kandemir received his Ph.D. degree from the Department of Computer Science at Syracuse University. Contact him at mtk2@psu.edu.

MYOUNGSOO JUNG is a full professor in the School of Electrical Engineering, Korea Advanced Institute of Science Technology, Daejeon, 34141, South Korea. He is also the CEO at Panmnesia, Inc. His research interests include computer architecture, operating systems, and link solutions including CXL, UALink, and Ethernet. Jung received his Ph.D. degree from the Department of Computer Science and Engineering at Pennsylvania State University. He is the corresponding author of this article. Contact him at mj@panmnesia.com.

Compute Express Link Topology-Aware and Expander-Driven Prefetching: Unlocking Solid-State Drive Performance

Dongsuk Oh, Miryeong Kwon, Jiseon Kim, Eunjee Na, Junseok Moon, Hyunkyu Choi, Seonghyeon Jang, Hanjin Choi, Hongjoo Jung, and Sangwon Lee, *Next-Generation Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea*

Myoungsoo Jung, *Next-Generation Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea; Advanced Product Engineering Division, Panmnesia, Inc., Seoul, 06180, South Korea; and KAIST, Daejeon, 34141, South Korea*

Integrating compute express link (CXL) with solid-state drives (SSDs) allows scalable access to large memory but has slower speeds than dynamic RAMs. We present ExPAND, an expander-driven CXL prefetcher that offloads last-level cache prefetching from host CPU to CXL-SSDs. ExPAND uses a heterogeneous prediction algorithm for prefetching and ensures data consistency with CXL.mem's back-invalidation. We examine prefetch timeliness for accurate latency estimation. ExPAND, being aware of CXL multitiered switching, provides end-to-end latency for each CXL-SSD and precise prefetch timeliness estimations. Our method reduces CXL-SSD reliance and enables direct host cache access for most data. ExPAND enhances graph application performance and SPEC CPU's performance by 9.0× and 14.7×, respectively, surpassing CXL-SSD pools with diverse prefetching strategies.

Compute express link (CXL) is a key interface for enabling memory disaggregation, where memory resources are decoupled from computing servers to provide scalable access to large-capacity memory. This shift is particularly relevant as storage class memory (SCM) technologies—such as programmable RAM (PRAM), Z-NAND, and XL-Flash—offer significant capacity advantages compared to dynamic RAM (DRAM). SCM's ability to store large datasets with byte-addressable access makes it an attractive candidate for new memory systems. As a result, both industry and academia are exploring the potential of byte-addressable solid-state drives (SSDs) that leverage the CXL protocol to combine SCM's memory semantics with scalable interconnects.

Despite the potential of CXL-SSDs for addressing the increasing demand for memory capacity, the underlying SCM technologies remain significantly slower than DRAM. PRAM, for instance, has been shown to exhibit access latencies up to 7× higher than DRAM, while Z-NAND and XL-Flash introduce latencies that are approximately 30× slower. To mitigate these, many designs incorporate SSD-side DRAM buffers as internal caches, mimicking the architecture of high-performance nonvolatile memory express (NVMe) storage equipped with substantial internal DRAM. These are effective in reducing write latency but are insufficient to address the long read latencies caused by SCM backend media.

Addressing read latency in CXL-SSDs requires a departure from traditional SSD design principles. CXL-SSDs directly serve memory requests using load/store operations, bypassing the host-side storage stack. This fundamental shift necessitates understanding the execution patterns of host applications and managing

Digital Object Identifier 10.1109/MM.2025.3574293
Date of publication 30 May 2025; date of current version 24 December 2025.

the CPU cache hierarchy to align with the unique performance characteristics of CXL-SSDs. However, current SSD technologies have largely been designed to handle block-level requests, leaving them ill-equipped to address the challenges associated with high-latency memory requests. As a result, there remains a pressing need for solutions that can bridge this gap, ensuring that CXL-SSDs can fully realize their potential in emerging memory disaggregation architectures.

When CXL-SSDs are integrated into the system memory space as host-managed device memory, existing CPU-side cache prefetching mechanisms can still provide some performance benefits. However, two primary challenges limit the effectiveness of current prefetchers within the cache hierarchy in fully exploiting the advantages of the last-level cache (LLC) for CXL-SSDs: 1) *hardware logic size constraints* that hinder the handling of diverse memory access patterns encountered in the CXL memory pooling space, and 2) *latency variations* caused by the differing physical positions of CXL-SSDs within the CXL switch network.

Rule-based cache prefetchers, such as spatial[1] and temporal algorithms[2] often require tens of megabytes of storage, comparable to the size of a typical CPU LLC. Due to these high storage requirements, modern CPUs employ simpler prefetching algorithms, such as stream cache prefetchers. While these are efficient in hardware implementation, they are insufficient to mask the increased latency introduced by CXL-SSDs.

An additional complication arises from the interconnect topology used in CXL-based memory disaggregation. To enable scalable memory expansion, CXL employs a multilevel switch architecture, where each switch level introduces a processing delay. The cumulative latency depends on the position of the target device within the switch network, with deeper levels resulting in higher delays. This latency variation prevents existing prefetchers from retrieving data from CXL-SSDs distributed across the network.

This article presents an expander-driven CXL prefetcher, *ExPAND*, designed to offload primary LLC prefetching tasks from the host CPU to CXL-SSDs, addressing CPU design area constraints. Implemented within CXL-SSDs, ExPAND employs a heterogeneous machine learning (ML) algorithm for address prediction, enabling data prefetching across multiple expander accesses. The host-side logic in ExPAND ensures that CXL-SSDs remain aware of the execution semantics of host-side CPUs, while the SSD-side logic maintains data consistency between the LLC and CXL-SSDs using CXL.mem's *back-invalidation* (*BI*) mechanism. This bidirectional collaboration allows reducing reliance on CXL-SSDs for frequent memory requests.

Accurate estimation of prefetching latency is essential for optimizing the limited capacity of on-chip caches. To address this, we define the concept of *prefetch timeliness*, representing the latency constraints inherent to CXL-based prefetching. ExPAND incorporates a detailed understanding of prefetch timeliness by identifying the CXL network topology and device latencies during peripheral component interconnect express (PCIe) enumeration and device discovery. Using this topology information, ExPAND calculates precise end-to-end latency values for each CXL-SSD within the network and writes these values into the PCIe configuration space of each device. This information enables the offloaded cache prefetching algorithm to determine the optimal timing for transferring data to the host LLC, effectively mitigating the long read latencies caused by the slower backend media of CXL-SSDs.

Our evaluation results show that ExPAND enhances graph application performance and SPEC CPU's performance by 9× and 14.7×, respectively, surpassing CXL-SSD pools with diverse prefetching strategies.

BACKGROUND

Memory Pooling Using CXL

Protocol Primary

CXL is a cache-coherent interconnect designed for heterogeneous devices, enabling scalable memory expansion. It consists of three subprotocols: *CXL.io*, *CXL.cache*, and *CXL.mem*. Built on the PCIe physical layer, CXL.io functions as a direct counterpart to the PCIe protocol. CXL.cache facilitates efficient access to host memory for accelerators, while CXL.mem enables hosts to access memory attached to devices across the CXL network. Together, CXL.io and CXL.mem support the connection of multiple memory expanders to create large-scale memory pools. Note that memory expanders can connect to the host system memory without requiring CXL.cache, appearing to CPUs as locally attached memory. This is possible because CXL allows *endpoint* (*EP*) devices to be mapped into the cacheable memory space.

Incorporating CXL Into Storage

CXL.mem and CXL.io allow CPU to access memory via load/store instructions, using a CXL message packet called a *flit*. This flit-based communication enables various memory and storage media to be integrated into the CXL pool. SCMs having greater capacity than DRAM is leading to an interest in integrating CXL into block storage, known as *CXL-SSDs*. CXL-SSDs often

use large internal DRAM caches to store data ahead of backend SCMs, achieving performance akin to DRAM-based EP expanders. Samsung's proof of concept employs Z-NAND and a 16-GB DRAM cache, claiming an 18× write latency improvement over NVMe SSDs. Kioxia's proof of concept uses XL-Flash and a sizable DRAM prefetch buffer, asserting DRAM-like speeds by combining prefetching with hardware compression.

Go Beyond Pooling

Enhanced Memory Coherence

CXL's flit-based communication decouples memory resources from processor complexes, enabling efficient memory pooling. However, while CXL.cache provides cache coherence, it imposes considerable overhead on EP devices when managing their internal memory. When a CXL-SSD employs CXL.cache to synchronize host-side cache updates, frequent monitoring and approval for memory accesses targeting internal DRAM or backend SCM are required to maintain coherence. To address this, the BI introduced by CXL 3.0 enables CXL.mem to back-snoop host cache lines. This feature allows EPs, such as CXL-SSDs, to autonomously invalidate host cache lines, reducing dependence on CXL.cache while maintaining coherency.

Multitiered Switching

EP expanders within a pool are interconnected via one or more CXL switches. Each CXL switch consists of *upstream ports* and *downstream ports*, allowing connections between CPUs and EP expanders. A fabric manager configures and manages these ports, enabling each host to access its EP expanders through a dedicated data path known as a *virtual hierarchy* (VH). Previously, CXL architectures were restricted to a single switch layer, which limited the capacity of each VH. With the introduction of *multitiered switching* in CXL 3.0/3.1, switch ports can now connect to additional switches, significantly increasing the capacity of each VH. This enhancement supports up to 4 K devices per VH and accommodates various CXL subprotocols, greatly improving scalability for resource disaggregation.

MOTIVATION AND CHALLENGES

Prefetching Impact

The performance impact and effectiveness of prefetching are influenced by two parameters[1]: *prefetch accuracy* and *prefetch coverage*. Prefetch accuracy is the proportion of prefetched data actually utilized by the target application, while prefetch coverage measures the fraction of total memory requests served by prefetched data.

Figure 1(a) shows the latency speedup achieved by applying prefetching techniques in CXL-SSD systems, normalized to the latency of LocalDRAM. The graph demonstrates the relationship between prefetch effectiveness and latency improvements in graph applications. Both parameters were configured with identical values, varying from 0% to 100%, to evaluate their combined impact on system behavior. This analysis uses four representative large-scale graph workloads sourced from Leskovec et al.,[3] with further details provided in the "Evaluation" section.

The results indicate that CXL-SSD performance is significantly slower than LocalDRAM, with up to 4.5× slower latency when prefetch effectiveness is below 80%. However, performance improves substantially as prefetch effectiveness increases. Once prefetch effectiveness exceeds 90%, the increased cache hit rate reduces the frequency of actual memory accesses, leading to notable latency reductions. While unrealistic, a perfect prefetch allows the CXL-SSD to outperform LocalDRAM, improving latency up to 3.9×.

To better understand the performance improvements by prefetching, we analyzed the LLC *misses per kilo instructions* (*MPKI*) for each workload, as shown in Figure 1(b). MPKI increases in the order of connected components (CC), triangle counting (TC), PageRank (PR), and single-source shortest path (SSSP), and the degree of performance improvement corresponds to this order. This improvement occurs because the

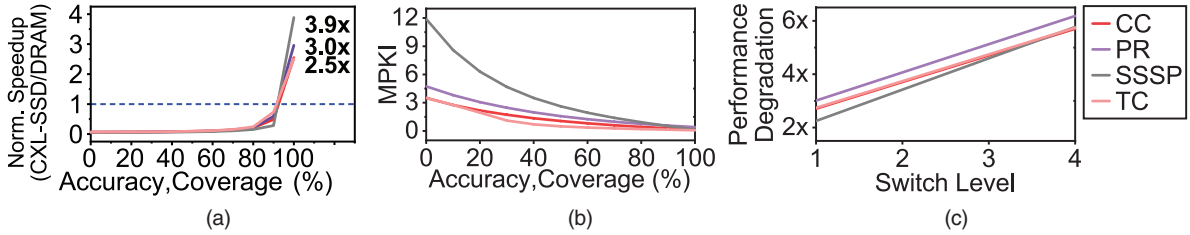


FIGURE 1. CXL-SSD prefetching performance analysis. (a) Speedup, (b) misses per kilo instructions, and (c) unawareness impact of CXL switches. CC: connected components; PR: PageRank; SSSP: single-source shortest path; TC: triangle counting.

prefetcher prepares data both accurately and in a timely manner, allowing the CPU to perform tasks without memory stalls.

Latency Variation With CXL Switch Topology

Unawareness Impact of CXL Switches

Even a high-effective prefetcher cannot fully mitigate performance degradation caused by latency variations due to multitiered switching. Specifically, conventional prefetchers don't account for additional latency from CXL switches, effectively making cache hits to misses. This limitation reduces prefetcher effectiveness in CXL systems, thus degrading performance.

To evaluate how a multitiered CXL switch architecture affects application performance, we increased the switch layers between the host and CXL-SSD from one to four and compared the performance against a baseline system without any switches. For consistent analysis, we utilized 90% prefetch effectiveness, which was the median value in Figure 1(a). Figure 1(c) shows that the four graph workloads (CC, PR, TC, SSSP) experienced a 2.7× performance degradation per additional CXL topology switch layer in average. The graph's slopes reflect degradation rates, where CC, PR, and TC shows 1.3× degradation per switch layer, whereas SSSP shows 1.4× per layer.

EXPANDER-DRIVEN PREFETCHING

CPU-side rule-based spatial[1] and temporal[2] prefetchers have been adopted in various industrial processors. However, their accuracy is limited (9% to 76%) for workloads with large-scale, irregular, or random memory access patterns, insufficient to accelerate CXL-SSD performance to match Local DRAM levels (e.g., 90% accuracy). To address these limitations, more advanced prefetching techniques incorporating ML approaches have been proposed.[4,5] Prefetching involves prediction, making ML-based approaches promising for higher accuracy. While these techniques could achieve the required accuracy threshold, they remain impractical for on-chip CPU implementation due to substantial storage requirements for model computation and metadata management overhead.[6,7,8]

On the other hand, techniques to minimize the memory overhead of ML-based prefetching algorithms have also been proposed.[7,8] Unfortunately, these methods either require profiling-based offline training using workload memory traces, thus limiting their effectiveness for accelerating unseen workloads,[7] or utilize knowledge distillation and product quantization techniques to reduce memory requirements, resulting in low accuracy and high training complexity.[8] In addition, existing prefetching algorithms are unaware of multilevel switch architectures, thus unable to account for latency variations inherent in CXL topology.

Prefetching Delegation and Collaboration

This article introduces ExPAND, which delegates cache prefetching decisions to CXL-SSDs, enabling autonomous CPU cache line updates. By shifting decision making to the EP side, ExPAND leverages the larger form factor and computational capabilities of SSD EPs versus on-chip CPUs. Figure 2(a) presents ExPAND's architecture, which comprises two key components: the *reflector* and the *decider*.

The reflector is implemented on the host-side CXL *root complex* (RC) and LLC controller. Its main role is to provide the decider with essential decision-making inputs like program counter (PC) and switch depth of connected CXL-SSD, while communicating cache prefetching results determined by the decider. Using a small buffer (16 KB), it logs cache line updates prefetched by the decider. For efficiency, each host's LLC controller in the CXL network first checks this

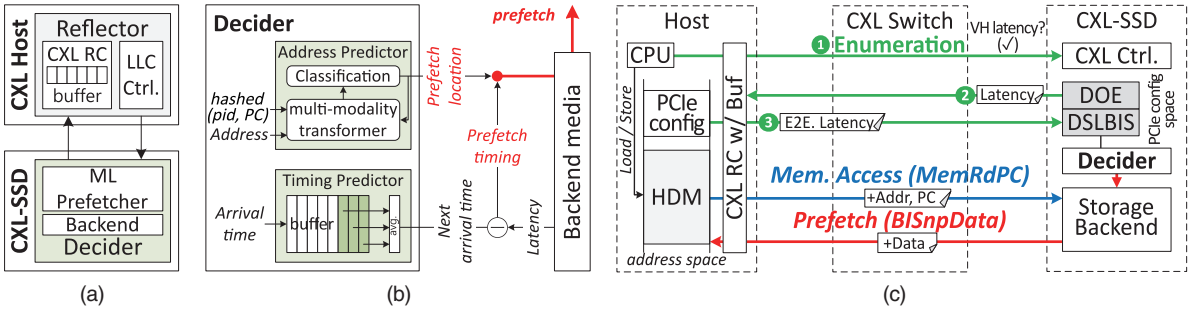


FIGURE 2. Overview of ExPAND. (a) Architecture, (b) expander-driven prefetching, and (c) CXL topology aware prefetch timeliness. DOE: data object exchange; DSLBIS: device scoped latency and bandwidth information structure; RC: root complex.

buffer. If the required data are present in the CXL RC, it's served directly from the buffer, avoiding unnecessary traversal through the CXL-SSD pool.

The decider resides in the EP-side CXL-SSD controller and implements a heterogeneous ML prefetcher optimized for irregular memory access patterns.[5] Using the provided inputs (PC and memory address), the decider identifies and transfers data to the reflector buffer. It records input data for refining prefetching patterns. Details of the prefetcher's operation and its interaction with the reflector will follow.

Prefetch Address and Timing Speculation

As shown in Figure 2(b), ExPAND's decider incorporates two specialized predictors: an address predictor and a timing predictor, responsible for determining the location and timing of cache prefetch operations, respectively. The address predictor is inspired by a multimodality transformer model,[5] while the timing predictor employs a simpler rule-based approach. The address predictor generates a sequence of memory addresses for prefetching by utilizing a transformer as its sequence model. It integrates a multimodality attention network[9] to enhance the analysis of relationships between memory access patterns and the PC. In addition, it monitors changes in application execution behavior using a decision tree classifier[10] to dynamically refine prefetching accuracy.

ExPAND's decision tree classifier is pretrained to categorize memory traces of various applications into 64 categories. For online inference, ExPAND maintains a sliding window containing recent memory addresses and their corresponding PCs, feeding this information to the classifier model. The classifier infers the current window's requests into one of the pretrained 64 categories. If the classifier's inference changes from the previously inferred category, ExPAND records this as a behavior-change event. Such events are then provided as hints, along with memory addresses, to the multimodality transformer model. By recognizing these behavior-change events, the transformer model achieves more accurate predictions of subsequent addresses. This detection-based feedback allows the transformer model to respond promptly to changes in memory access patterns.

The timing predictor, in contrast, maintains request arrival time information in a small-sized buffer (80B) and estimates future memory request times by averaging historical arrival times within its history window. Accurate prediction requires retaining all past arrival times in the history window. However, when memory requests are served directly by the LLC, relevant information may not reach the timing predictor. To address this, the reflector notifies the decider of cache hit events via CXL.io, enabling the timing predictor to account for future request times even when no direct requests are observed.

CXL CROSS-LAYER INTERSECTION

CXL Topology-Aware Prefetch Timeliness

CXL Switch Hierarchy Discovery

The reflector identifies the switch level of CXL-SSDs during PCIe enumeration. The host accesses the configuration space of connected devices (using CXL.io) and organizes system buses within their CXL network. During enumeration, buses are segregated upon identifying new devices, with a unique number assigned to each bus. As CXL switch operates as a PCIe bridge device with a distinct bus number, it allows determining the number of switches between a host CPU and target CXL-SSD. The reflector stores this information on its RC side, which aids in estimating accurate prefetch timeliness, detailed subsequently.

Timeliness Speculation

Prefetching data too early could contaminate the LLC, reducing hit ratio, while prefetching too late may delay execution. Therefore, pinpointing exact prefetch timeliness is essential. Figure 2(c) shows this process. Since CXL EPs must manage *data object exchange* (*DOE*) capability in PCIe configuration space, each CXL-SSD can determine its device latency through DOE. However, this latency cannot directly estimate prefetch timeliness due to variations from multitiered switching. During enumeration, the reflector retrieves each CXL-SSD's latency by extracting *device scoped latency and bandwidth information structure* (*DSLBIS*) from DOE. It then calculates the latency overhead incurred in VH between the RC and target CXL-SSD. The reflector combines this VH latency with the DSLBIS latency and stores the end-to-end latency in the corresponding device's configuration space. Consequently, the decider estimates prefetch timeliness by subtracting the end-to-end latency from the time predicted by its timing predictor.

Bidirectional Communication on CXL

Downward: Piggybacking on CXL.mem

To accurately predict addresses, timely transmission of PCs and corresponding memory requests is vital. CXL.mem's master-to-subordinate (M2S) transactions

include request without data (Req), request with data (RwD), and BI response (BIRsp). Req is primarily for memory read opcode (MemRd) without payload, while RwD carries payload for memory write opcode (MemWr). RwD allows 13 custom opcodes, enabling an opcode for memory reads with PCs (MemRdPC). When a read misses the LLC, the reflector sends an M2S transaction using MemRdPC, including the current PC. Consequently, the target decider can access the memory address and PC in the host's execution environment. Note that BIRsp responds to the CXL-SSD's BI snoop command, discussed subsequently.

Upward: Leveraging BI

When it reaches the time to prefetch (estimated in the "CXL Topology-Aware Prefetch Timeliness" section), the decider must update the reflector buffer with data obtained from its address predictor results. However, the existing CXL.mem lacks the capability to update host-side on-chip storage. To address this limitation, we use the CXL.mem subordinate-to-master (S2M) transaction's BISnp. BISnp, similar to CXL.mem's Req, is a nonpayload message. We introduce a new BI opcode, termed *BISnpData*, to the S2M transaction message, allowing up to 10 custom opcodes. Using BISnpData, the decider generates a payload accompanying its message, containing data for updating the host. When the reflector detects BISnpData, it awaits the corresponding payload and inserts the prefetched data into its buffer, enabling the LLC controller to fetch it for execution.

EVALUATION

- *Methodologies:* Since no CXL-SSDs are currently available, we used the CXL hardware RTL modules in a full system simulation model. We conducted this simulation using gem5[11] and SimpleSSD.[12] These CXL RTL modules have been validated using a real CXL end-to-end system at the cycle-level. Tables 1–4 provide the main parameters of our simulation. We compare ExPAND, our proposed expander-driven prefetcher, with several modern rule-based and ML-based prefetchers. These include a spatial prefetcher[1] (`Rule1`), a temporal prefetcher[2] (`Rule2`), an LSTM-based prefetcher[6] (`ML1`), and a transformer-based prefetcher[4] (`ML2`). We summarize the important characteristics of each prefetching algorithm in Table 4.
- *Workload and benchmarks:* To evaluate the effectiveness of ExPAND in diverse graph application scenarios, we employed four widely used algorithms across five datasets. The algorithms were sourced from established graph processing frameworks: CC, PR, SSSP, and TC. These were applied to five datasets: Amazon's product copurchasing network, Google's web graph, the California road network, the Wikipedia talk network, and the YouTube online social network. We also summarize key characteristics of each workload in Table 3.

Overall Analysis of Prefetching

We evaluated all five prefetching techniques (`Rule1`, `Rule2`, `ML1`, `ML2`, and `ExPAND`) across the graph workloads and SPEC CPU benchmarks. For clarity, we normalized their performance relative to `NoPrefetch`, representing a CXL-SSD without prefetching. Figure 3(a) analyzes the speedup of the five prefetching techniques normalized to `NoPrefetch`.

Rule-Based Prefetcher

In graph applications, `Rule1` achieves a performance improvement over `NoPrefetch` and `Rule2` by 2× and 1.8×, respectively. This enhancement occurs because

TABLE 1. Evaluation setup: Host configurations.

Parameter	Value
CPU	OoO 12 cores@3.6 GHz, 512-entry ROB
L1 I$	32 KB two-way, five-cycle latency
L1 D$	48 KB two-way, five-cycle latency
L2$	1.25 MB 16-way 20-cycle latency
DRAM	tRP = tRCD = tCAS = 22 ns,
	8 rank, 16 bank, 2 channel
PMEM	Intel P5800X
PCIe/CXL	64 GT/s (PCIe 6.0), CXL 3.0

TABLE 2. Evaluation setup: CXL-SSD configurations.

Parameter	Value
NAND Flash	Samsung 983 ZET, 2TB
	tRd: 3 μs, tWr: 100 μs, tEr: 1 ms
Internal DRAM	tRP = tRCD = 9.1 ns,
	tRAS = 19 ns, size = 1.5 GB
Location predictor	Attention dimension: 64
	Modality fusion dimension: 128
	Transformer dimension: 128
Timing	Buffer entries: 10

graph applications typically exhibit significant spatial locality, making data accesses easier for prefetchers to predict compared to temporal patterns. In contrast, performance varies across SPEC CPU benchmarks. Workloads with high MPKI (e.g., mcf) still perform comparably to NoPrefetch. However, workloads exhibiting structured, repetitive memory access patterns achieve an average speedup of 7×.

ML-Based Prefetcher

ML-based prefetchers achieve performance improvements of 1.6× over rule-based prefetchers and 4.4× over NoPrefetch. This gain is attributed to their ability to effectively learn complex spatial and temporal locality patterns. ExPAND, our proposed multimodality transformer-based prefetcher, further outperforms existing ML-based prefetchers by 2.4×, achieving speedups ranging from 4.3× up to 71.8× over NoPrefetch. ExPAND particularly excels in workloads dominated by stencil computations, such as bwaves, leslie3d, and lbm. Stencil computations typically involve referencing neighboring data points across multiple dimensions.

TABLE 3. Evaluation setup: Workloads.

Workload	Working set (GB)	MPKI	Read ratio
PR	82	4.13	0.01
SSSP	428	11.03	0.01
TC	31	3.13	0.01
bwaves	22	0.27	0.84
leslie3d	41	0.45	0.52
lbm	22	0.28	0.03
libquantum	141	1.48	0.63
mcf	215	12.17	0.87

TABLE 4. Evaluation setup: Prefetch algorithms.

Algorithm	Memory overhead	IOPs	Accuracy
Prior work 1[7]	64 KB	56.6 K	86%
Prior work 2[8]	548.8 KB	4.9 M	81%
Rule 1	4 KB	768	82%
Rule 2	8 KB	2304	53%
ML 1	936.8 KB	11.3 M	88%
ML 2	865 KB	26 M	89%
ExPAND	839.2 KB	10.3 M	92%

Performance With Mixed Workloads

Figure 3(b) illustrates the execution time results under mixed workload scenarios, where each core simultaneously runs distinct workloads. The performance of existing prefetching algorithms (Rule1, ML1, and ML2) significantly degrades under mixed workloads due to reduced accuracy in predicting subsequent memory addresses resulting from intertwined memory access patterns. An exception is Rule2, which preprocesses memory accesses by grouping addresses with similar values. Consequently, Rule2 maintains relatively high performance when workloads with strong spatial locality (e.g., CC and TC) are combined.

In contrast, ExPAND employs a multimodality transformer model considering both PC and memory addresses, enabling it to effectively distinguish between memory access patterns even under mixed workloads. Therefore, under mixed workloads, ExPAND outperforms Rule1, Rule2, ML1, and ML2 by averages of 7.0×, 10.2×, 3.7×, and 3.5×, respectively.

Model Optimizations

We further evaluate ExPAND's prefetching optimizations, particularly its timeliness model and online tuning mechanisms. ExPAND's timeliness model considers backend media latency of CXL expansion devices and multilayer switch latency, enabling efficient utilization of limited LLC resources.

Figure 3(c) shows workload performance relative to the accuracy of the timeliness model, evaluated using the TC workload. As shown, improved timeliness

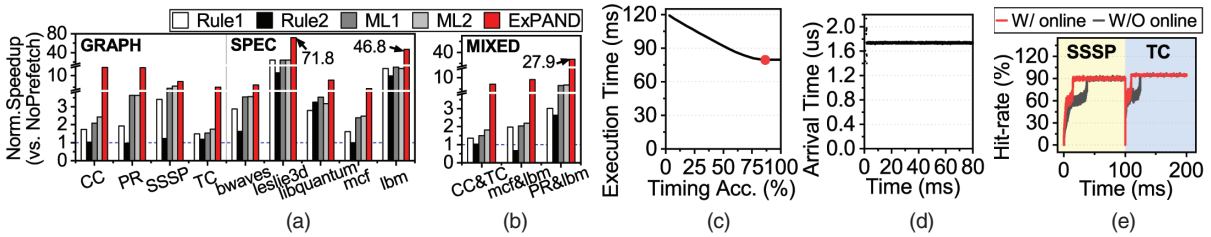


FIGURE 3. Overall performance. (a) Speedup compared to NoPrefetch, (b) mixed workload, (c) sensitivity, (d) arrival time, and (e) online tuning. W/: with; W/O: without.

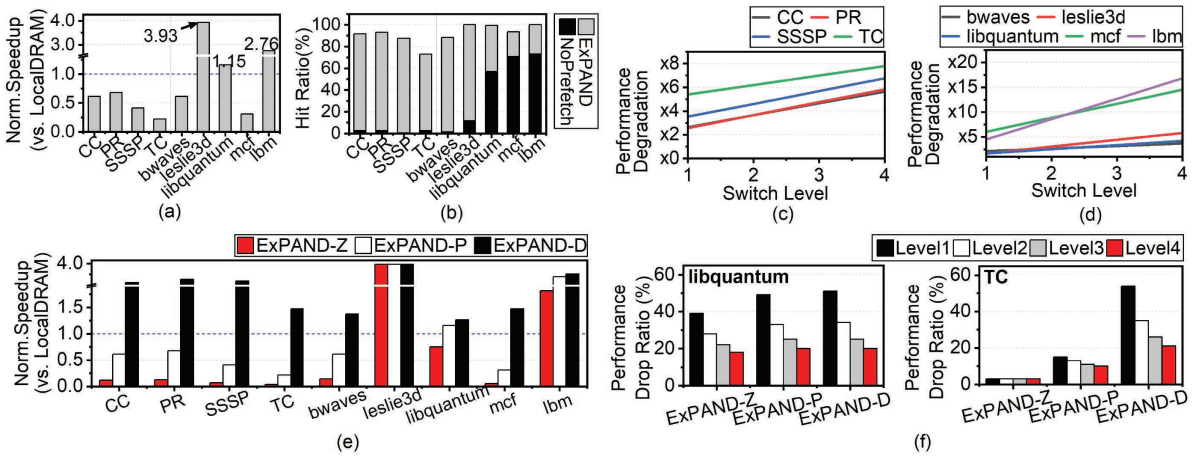


FIGURE 4. Detailed analysis. (a) Speedup, (b) LLC hit ratio, (c) multiswitch (graph), (d) multiswitch (SPEC CPU), (e) impacts of backend media, and (f) multiswitch analysis across different backend media.

accuracy directly correlates with reduced execution time. Low accuracy leads either to early prefetching—causing prefetched data to be evicted before usage—or delayed prefetching, resulting in data being unavailable when required. Performance gains begin saturating at around 68% timeliness accuracy, with marginal improvements beyond 84%. This saturation occurs because LLC associativity typically mitigates eviction issues arising from minor timing inaccuracies.

ExPAND's timeliness model achieves 90% accuracy, significantly improving workload performance. Despite being heuristic-based, this high accuracy is achievable since LLC access frequencies remain relatively constant during workload execution. Figure 3(d) illustrates the intervals between LLC accesses during execution of the TC workload. As seen, LLC access frequency remains stable at runtime, influenced primarily by the workload's inherent memory access frequency and randomness, both of which typically remain constant throughout execution.

Performance Gain

Figure 3(c) analyzes the performance gains from ExPAND's online tuning. The evaluation measures LLC hit rates in scenarios with and without online tuning, focusing on dynamic workload transitions. Two workloads, SSSP and TC, were chosen due to their contrasting memory access patterns: SSSP has sequential accesses, whereas TC involves large-stride patterns.

The results show that online tuning enables significantly faster recovery of LLC hit rates following behavioral changes in workloads. Without tuning, the transformer model struggles to quickly adapt due to reliance on patterns from a large historical window, making it difficult to detect sudden changes. In contrast, ExPAND's decision tree classifier rapidly identifies changes in workload behavior, immediately notifying the transformer model. Consequently, the transformer adjusts swiftly by prioritizing recent memory accesses.

Performance Comparison With Local DRAM

In the previous section, we analyzed the performance improvements from expander-driven prefetching on CXL-SSD. Here, we evaluate its practical effectiveness during application execution. For this analysis, we compare `ExPAND`, the most effective prefetching technique, against a baseline using only local DRAM (`LocalDRAM`).

Impacts of Prefetching

Figure 4(a) shows the normalized performance of `ExPAND` against `LocalDRAM` in terms of application execution time. To better understand the performance gap, we also measured the LLC hit ratio, illustrated in Figure 4(b). The NoPrefetch percentage reflects the baseline LLC hit ratio without prefetching, while `ExPAND` highlights the additional improvements achieved through prefetching. `ExPAND` delivers a 9× performance improvement over `NoPrefetch` across four graph workloads. However, compared to the `LocalDRAM` baseline, it shows a 48% performance degradation due to the 14% cache miss rate, which required accessing CXL-SSD. Consequently, application execution is delayed despite `ExPAND` achieving an 86% LLC hit rate. In contrast, SPEC benchmarks, such as leslie3d, libquantum, and lbm demonstrate 3.9×, 1.2×, and 2.8× performance improvements over `LocalDRAM`, respectively. This

improvement is driven by ExPAND's ability to increase the LLC hit ratio by 46% on average, achieving hit rates as high as 96% for theses workloads.

Impacts of Timeliness

We conducted a sensitivity evaluation by incrementally increasing the VH level of the CXL switch from one to four to understand the impact of latency variation with topology. Figure 4(c) and (d) show the impact of increasing switch levels on performance for each workload. The performance trends align with the LLC hit ratio in Figure 4(b) and (c). Applications with high ExPAND-driven cache hit rates experience significant performance degradation at switch level 1 due to prefetchers that are unaware of the CXL topology, leading to reduced prefetch effectiveness. Conversely, workloads with higher NoPrefetch LLC hit ratios experience steeper degradation as levels increase, as added switch latency affects applications that had benefit from high LLC hit rates. Graph workloads [Figure 4(c)] show consistent degradation, averaging a 1.2× slowdown. SPEC CPU benchmarks [Figure 4(d)] exhibit varied slowdowns reflecting different sensitivities to switch latency.

Diversity of Backend Media

To evaluate the impact of backend media on expander-driven prefetching and application execution time, we conducted experiments with Z-NAND, PMEM, and DRAM as backend media options. These configurations are referred to as ExPAND-Z, ExPAND-P, and ExPAND-D, respectively. ExPAND-Z and ExPAND-P were tested to examine the feasibility of using Z-NAND and PMEM as main memory, while ExPAND-D was analyzed to explore the maximum potential benefits of expander-driven prefetching.

Impacts of Prefetching

Figure 4(e) presents the execution times of graph workloads and SPEC benchmarks using memory expanders with different backend media. ExPAND-Z, which utilizes Z-NAND as its backend (6× slower than PMEM), exhibits an average of 3× higher performance degradation compared to ExPAND-P. However, for workloads, such as leslie3d and lbm, ExPAND-Z achieves 3.9× and 1.8× better performance, respectively, than LocalDRAM. This indicates that, for specific workloads, a PMEM-based memory expansion system can outperform LocalDRAM configurations. ExPAND-D, leveraging DRAM as its backend, outperforms LocalDRAM across all graph workloads and SPEC benchmarks, with performance improvements ranging from 1.27× to 3.93× and an average gain is 1.9×.

Impacts of Timeliness

To evaluate the impact of backend media on prefetching timeliness, we performed a switch-level sensitivity analysis using libquantum (highest LLC hit ratio) and TC (lowest LLC hit ratio) as representative workloads [compare with Figure 4(b)]. For libquantum, the high LLC hit ratio minimizes the impact of backend media latency, making switch latency the dominant factor as shown in Figure 4(f). In contrast, TC's low LLC hit ratio amplifies backend media latency, reducing the impact of switch latency. ExPAND-Z and ExPAND-P show 15% and 3% degradation at switch level 1, with average reductions of 12% and 3% for further levels. ExPAND-D, with its low backend latency, experiences a 54% drop at switch level 1 and an average of 32% for additional levels, highlighting its sensitivity to switch latency.

CONCLUSION

We propose an expander-driven CXL prefetcher that offloads LLC prefetching to CXL-SSDs, employing a heterogeneous prediction algorithm. ExPAND ensures data consistency and provides precise prefetch timeliness estimates, reducing CXL-SSD reliance and enhancing graph application performance and SPEC CPU's performance by 9× and 14.7, respectively.

ACKNOWLEDGMENTS

The work is supported in part by the Institute of Information and Communications Technology Planning and Evaluation's (IITP's) RS-2023-00221040, RS-2024-00460762, RS-2025-02214652, RS-2025-02214654, and RS-2025-02263080; Korea Institute for Advancment of Technology's (KIAT's) P0027923 and P0028225, and Ministry of SMEs and Startups (MSS) Technology Development Program's RS-2023-00303967. The authors thank anonymous reviewers for their constructive feedback. This work is protected by one or more patents.

REFERENCES

1. P. Michaud, "Best-offset hardware prefetching," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2016, pp. 469–480, doi: 10.1109/HPCA.2016.7446087.
2. A. Jain and C. Lin, "Linearizing irregular memory accesses for improved correlated prefetching," in *Proc. 46th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2013, pp. 247–259, doi: 10.1145/2540708.2540730.
3. J. Leskovec and R. Sosič, "SNAP: A general-purpose network analysis and graph-mining library," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 1, pp. 1–20, 2016, doi: 10.1145/2898361.
4. P. Zhang, A. Srivastava, A. V. Nori, R. Kannan, and V. K. Prasanna, "Fine-grained address segmentation

for attention-based variable-degree prefetching," in *Proc. 19th ACM Int. Conf. Comput. Frontiers*, 2022, pp. 103–112, doi: 10.1145/3528416.3530236.
5. P. Zhang, R. Kannan, V. K. Prasanna, "Phases, modalities, temporal and spatial locality: Domain specific ML prefetcher for accelerating graph analytics," 2022, *arXiv:2212.05250*.
6. Z. Shi, A. Jain, K. Swersky, M. Hashemi, P. Ranganathan, and C. Lin, "A hierarchical neural model of data prefetching," in *Proc. 26th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2021, pp. 861–873, doi: 10.1145/3445814.3446752.
7. Q. Duong, A. Jain, and C. Lin, "A new formulation of neural data prefetching," in *Proc. ACM/IEEE 51st Annu. Int. Symp. Comput. Archit. (ISCA)*, 2024, pp. 1173–1187, doi: 10.1109/ISCA59077.2024.00088.
8. P. Zhang, N. Gupta, R. Kannan, and V. K. Prasanna, "Attention, distillation, and tabularization: Towards practical neural network-based prefetching," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2024, pp. 876–888, doi: 10.1109/IPDPS57955.2024.00082.
9. C. Jewitt, J. Bezemer, and K. O'Halloran, *Introducing Multimodality*. New York, NY, USA: Routledge, 2016.
10. A. J. Myles, R. N. Feudale, Y. Liu, N. A. Woody, and S. D. Brown, "An introduction to decision tree modeling," *J. Chemom.*, vol. 18, no. 6, pp. 275–285, 2004, doi: 10.1002/cem.873.
11. N. Binkert et al., "The gem5 simulator," *ACM SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, 2011, doi: 10.1145/2024716.2024718.
12. D. Gouk et al., "Amber: Enabling precise full-system simulation with detailed modeling of all SSD resources," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2018, pp. 469–481, doi: 10.1109/MICRO.2018.00045.

DONGSUK OH is an engineer at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include network/storage, edge computing, RISC-V, and compute express link. Contact him at dsoh@panmnesia.com.

MIRYEONG KWON is the chief strategy officer of Panmnesia, Inc., Daejeon, 34136, South Korea. Her research interests include graph analysis, GPU/solid-state drive integration, on-device artificial intelligence, and compute express link. Kwon received her Ph.D. degree at KAIST, Daejeon, South Korea. Contact her at mkwon@panmnesia.com.

JISEON KIM is an engineer at Panmnesia, Inc., Daejeon, 34136, South Korea. Her research interests include cache coherent interconnect, on-device artificial intelligence, memory, and compute express link. Contact her at jskim@panmnesia.com.

EUNJEE NA is an engineer at Panmnesia, Inc., Daejeon, 34136, South Korea. Her research interests include artificial intelligence infrastructure, disaggregated systems, and compute express link. Contact her at ejna@panmnesia.com.

JUNSEOK MOON is an engineer at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include network/storage, edge computing, artificial intelligence infrastructure, RISC-V, and compute express link. Contact him at jsmoon@panmnesia.com.

HYUNKYU CHOI is an engineer at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include network/storage, edge computing, RISC-V, and compute express link. Contact him at hkchoi@panmnesia.com.

SEONGHYEON JANG is an engineer at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include edge computing, cache coherent interconnect, and compute express link. Contact him at shjang@panmnesia.com.

HANJIN CHOI is an engineer at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include artificial intelligence infrastructure, network/storage, RISC-V, and compute express link. Contact him at hjchoi@panmnesia.com.

HONGJOO JUNG is an engineer at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include machine learning acceleration, artificial intelligence infrastructure, and compute express link. Contact him at hjjung@panmnesia.com.

SANGWON LEE is an engineer at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include persistent memory, RISC-V, and compute express link. Lee received his Ph.D. degree at KAIST, Daejeon, South Korea. Contact him at swon@panmnesia.com.

MYOUNGSOO JUNG is a full professor in the School of Electrical Engineering, KAIST, Daejeon, 34141, South Korea; and the chief executive officer of Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include computer architecture, operating system, storage systems, nonvolatile memory, parallel processing, heterogeneous computing, and compute express link. He is the corresponding author of this article. Contact him at m.jung@kaist.ac.kr.

From Block to Byte: Transforming PCIe Solid-State Devices With Compute Express Link Memory Protocol and Instruction Annotation

Miryeong Kwon, Donghyun Gouk, Junhyeok Jang, Jinwoo Baek, Hyunwoo You, Sangyoon Ji, Hongjoo Jung, and Junseok Moon, *Next-Generation Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea*

Seungkwan Kang and Seungjun Lee, *Korea Advanced Institute of Science and Technology, Daejeon, 34141, South Korea*

Myoungsoo Jung, *Panmnesia, Inc., Daejeon, 34136, South Korea, and Korea Advanced Institute of Science and Technology, Daejeon, 34141, South Korea*

This article explores how compute express link (CXL) can transform peripheral component interconnect express (PCIe)-based block storage into a scalable, byte-addressable working memory. We focus on cacheability as a critical feature for adapting block storage to CXL's memory-centric architecture and introduce type 3 endpoint devices, referred to as CXL-solid-state devices (SSDs). To validate our approach, we prototype a CXL-SSD on a custom field-programmable gate array platform and propose annotation mechanisms, determinism and bufferability, to enhance performance while preserving data persistency. Our simulation-based evaluation demonstrates that CXL-SSD achieves 10.9× better performance than PCIe-based memory expanders and further reduces the latency by 5.4× with annotation enhancements. This work demonstrates the feasibility of integrating block storage into CXL's ecosystem and provides a foundation for future memory-storage convergence.

Cache coherent interconnects have recently emerged to integrate CPUs, accelerators, and memory components into a unified, heterogeneous computing domain. These interconnect technologies ensure data coherency between CPU memory and device-attached private memory, creating a new paradigm of globally shared memory and network space. Among several efforts to establish such connectivity, including Gen-Z,[1] and Cache coherent interconnect for accelerators,[2] *compute express link* (*CXL*) has become the first open interconnect protocol capable of supporting diverse processors and device endpoints. With the absorption of Gen-Z, CXL stands out as a promising interconnect interface due to its high-speed coherence control and seamless compatibility with the widely adopted peripheral component interconnect express (PCIe) standard. This makes it particularly advantageous for a wide range of data center-scale hardware, including CPUs, GPUs, field-programmable gate arrays (FPGAs), and domain-specific ASICs. Furthermore, the CXL consortium has highlighted its potential for memory disaggregation, enabling pooling of dynamic RAM (DRAM) and byte-addressable persistent memory.

Despite its versatility in managing computing resources and memory components, CXL currently does

Digital Object Identifier 10.1109/MM.2025.3581448
Date of publication 7 July 2025; date of current version 24 December 2025.

not support block storage, raising questions about whether storage devices can benefit from CXL's advantages. Storage designers and system architects might wonder what specific benefits CXL could bring to block storage and the significance of integrating the two. Although several studies have explored enabling a CXL interface for block storage,[3,4] the practical benefits of CXL over PCIe in this context remain uncertain. Notably, all current versions of CXL utilize PCIe's physical layers,[5] resulting in identical analog characteristics and low-level performance between the two interfaces. In addition, as CXL is designed to unify various hardware accelerators and computing devices into a coherent memory pool, careful consideration is required to determine the appropriate role of block storage within this ecosystem.

In this article, we argue that CXL can effectively transform PCIe-based block storage into a large, scalable working memory by addressing the key questions outlined earlier. By bridging the block-oriented semantics of PCIe storage with the byte-addressable, memory-compatible semantics of CXL, it enables PCIe storage to function efficiently as working memory.

We begin by examining the limitations that make PCIe storage impractical for use as a memory expander. While both PCIe and CXL allow access through memory instructions (e.g., loads and stores), we emphasize that CXL introduces a critical characteristic for inclusion in the memory hierarchy: cacheability. This capability allows on-chip cache hits to bypass accesses to the underlying memory, significantly reducing memory access latency and improving efficiency when integrating block storage into a coherent memory system. In addition, we advocate for the adoption of type 3 endpoint devices as the optimal choice for implementing flash-based CXL memory devices, referred to as CXL solid-state devices (CXL-SSDs), instead of type 2 devices. To substantiate this argument, we prototype a CXL-SSD by incorporating essential CXL intellectual properties into a custom FPGA platform, demonstrating the potential of CXL-enabled block storage as a scalable memory expander.

Although CXL can bring block storage closer to the CPU, the projected performance of CXL-SSD remains significantly behind that of DRAM-based media. To address this gap, we propose instruction annotation mechanisms: 1) determinism (DT) and 2) bufferability (BF). These mechanisms are designed to enhance the performance of CXL-SSD while preserving data persistency as block storage.

We evaluate this approach using a full-system simulation model based on our CXL-SSD prototype. The results demonstrate that CXL-SSD delivers a 10.9× performance improvement over a PCIe-based memory expander utilizing the same flash media. Furthermore, the annotation-augmented CXL-SSD reduces latency by an additional 5.4×, on average. In scenarios with high locality during application execution, CXL-SSD achieves performance levels comparable to DRAM-based local memory.

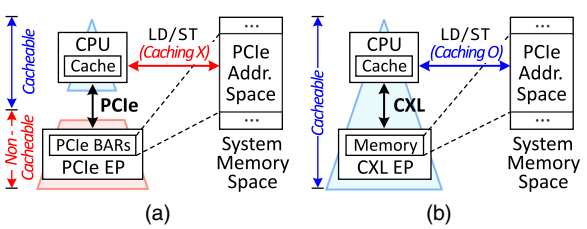


FIGURE 1. Memory expansion based on (a) PCIe and (b) CXL.

CXL MEMORY PROTOCOL

Why CXL Memory for PCIe Storage?

Byte-Addressability

Achieving byte-addressability for PCIe storage and integrating it into working memory devices has long been a key goal. For instance, industry prototypes and the nonvolatile memory express standard provide byte-addressability by exposing an SSD's internal memory or buffer to PCIe base address registers (BARs).[6] Since BARs can be directly mapped to the system memory space, host-side kernels and applications can access these exposed memory or buffer resources like local memory, using load/store instructions instead of block-level operations. To mitigate the long latencies associated with the backend block media of SSDs (e.g., Z-NAND, Flash, Optane SSD), the internal memory or buffer can serve as a write-back inclusive cache for the backend storage, effectively enhancing access performance.

Limits With Noncacheable Accesses

While PCIe bandwidth is sufficient for far memory (e.g., 63 GB/s to 121 GB/s for Gen5/6 with 16 lanes), PCIe treats block storage devices as peripheral components managed by the host-side CPU. As shown in Figure 1(a), although storage devices can process load/store requests through PCIe's BARs, they are inherently limited as working memory. Specifically, memory-mapped BARs serve only as an interface for the host to communicate requests or control signals to the storage, requiring all load/store requests to be noncacheable and directly accessible.

This noncacheable behavior significantly degrades memory access performance targeting BARs. If the CPU were to cache or buffer memory requests targeting the PCIe address space, the PCIe storage device could not detect or respond to these requests, potentially causing critical issues, such as system failure or storage disconnection. To avoid such scenarios, the ×86 instruction set from both Intel and AMD restrict CPU-level caching of PCIe-related memory requests. Unfortunately, this restriction excludes storage-integrated memory expanders from the conventional memory hierarchy, denying them CPU caching benefits.

Compute Express Link

CXL is a cache-coherent interconnect technology originally designed to support various accelerators and memory devices. It enables one or more memory address spaces within the PCIe network domain to be accessed coherently by processors and hardware accelerators, leveraging a multiprotocol approach. This multiprotocol capability extends beyond the PCIe input/output (I/O) semantics, ensuring compatibility between all CXL device types and existing PCIe devices. CXL device types are detailed in the next section.

Although CXL is built on PCIe, it fundamentally ensures cache coherence among computing complexes within the same CXL hierarchy. As shown in Figure 1(b), this allows load/store requests targeting the PCIe address space to be cacheable, addressing one of the critical limitations of PCIe. While current CXL implementations primarily target DRAM and persistent memory for memory pooling, we argue CXL can transform PCIe storage via a memory-like, byte-addressable interface, creating a significantly larger memory pool compared to traditional DRAM- or persistent memory-based memory expansion technologies.

Multiprotocol and Device Type Classification

CXL defines three subprotocols—1) CXL.io, 2) CXL.mem, and 3) CXL.cache—which correspond to three types of CXL devices, categorized as type 1, type 2, and type 3 endpoints.

Multiprotocol

CXL.io serves as the foundational protocol for communication between CXL-attached devices and host CPUs. It extends the full functionality of PCIe, acting as a noncoherent load/store interface for I/O operations, such as device discovery and host address configuration. In contrast, CXL.cache and CXL.mem introduce coherent memory access capabilities. These protocols enable support for multiple device domains and facilitate remote memory management. *CXL's root port* (*CXL RP*) allows memory addresses exposed by underlying CXL devices to be mapped into a host's cacheable system memory space. While this capability is primarily designed to unify diverse memory devices into a single, coherent memory pool, it can also be adapted to enable memory expanders leveraging various storage technologies.

CXL Device Types

CXL defines three device types—type 1, type 2, and type 3—based on how the multiprotocol features of CXL are utilized.

Type 1 devices include a local cache but lack internal memory. These devices access the host's memory directly through their cache using the CXL.cache protocol, ensuring memory coherence with the host. Type 2 devices are designed for discrete accelerators equipped with high-performance memory modules, referred as *host-managed device memory* (*HDM*). HDM differs fundamentally from the private memory modules used in conventional accelerators, such as GPUs, since the CXL.mem allows the host to access the HDM directly. Type 2 devices can also access the host's CPU memory by leveraging CXL.cache. Type 3 devices are designed for nonacceleration purposes, consisting solely of HDM without any processing components. These devices primarily use CXL.mem to handle load/store requests issued by the host, as CXL does not permit type 3 devices to initiate requests to the host via CXL.cache. Type 3 devices are well-suited for host memory expansion, as CXL.mem provides fundamental read/write interfaces for HDM. Details on how HDM is used will be discussed in the next section.

TRANSFORMATION OF PCIe SSD

Integrating CXL Protocol Into Block Storage

Device Type Consideration

PCIe storage is inherently more than a passive device, incorporating internal DRAM for buffering and caching incoming requests and associated data. It also includes computational capabilities for tasks, such as address translation and reliability management. Type 2 could be a viable option for PCIe storage, as they allow HDM usage and integrate data processing functions, while maintaining awareness of host CPU semantics. Several industrial proposals have explored using type 2 CXL memory expanders as data processing units.[7]

However, in this article, we advocate for type 3 devices as the optimal choice for storage-integrated memory expanders in CXL. There are two reasons why we believe that type 3 devices are better than type 2 devices for CXL-SSDs.

First, although type 2 allows the host to handle storage-side HDM directly, it is primarily designed for computationally intensive applications. Due to this, CXL allows only 16 devices per RP when full cache coherence is required. Therefore, type 2 devices do not offer the scalability provided by type 3 devices, which support up to 4095 devices per RP. Second, integrating full CXL.cache and CXL.mem functionalities into CXL-SSDs introduces additional communication overhead, negatively impacting overall performance. Specifically, each load/store request must check cache states within the storage's computing complex, generating multiple CXL transactions. Efficiently managing the internal DRAM of PCIe storage is important; however, there is no need for coherent management of the host CPU's caches, which adds unnecessary complexity. Starting from CXL 3.0, the standard supports CXL switches, enabling a host system to connect multiple CXL memory expansion devices through multilevel switches. Such a switch hierarchy allows expansion beyond the maximum number of ports (approximately four~eight) provided by a single switch, enabling scalability up to the maximum memory capacity supported by CXL, which is 4 PB.

System Integration

Figure 2(a) shows how CXL connects a PCIe storage device to a host and demonstrates how host-side users can directly access the storage device using load/store instructions. In this setup, the host system bus employs a CXL RP to connect a type 3 PCIe storage device.

During boot, the system enumerates all CXL devices connected to its RP and initializes them by mapping their internal memory into system memory. Specifically, the host retrieves the sizes of the CXL BAR and HDM from the storage devices. The CXL BAR and HDM are mapped, respectively, to noncacheable and cacheable memory regions, and these mappings are communicated to the underlying CXL controller [❶ in Figure 2(a)]. When an application issues a load/store request to the HDM, the CXL RP generates a CXL flit and sends it to the storage controller via the CXL.mem protocol [❷ in Figure 2(a)]. The storage controller, endpoint, and CXL controllers parses the flit to extract request details such as the command type and target address [❸ in Figure 2(a)]. Based on this, the controllers process the request in coordination with the storage firmware [❹ in Figure 2(a)].

Preliminary Performance Model of CXL-SSD

Prototypes

Since no commercial products support CXL for both processing complexes and endpoints in a modifiable setup, we prototyped a CXL-enabled CPU and CXL storage to represent a host and a storage-integrated memory expander, respectively. The prototypes are implemented on two separate custom FPGA boards connected via a tailored PCIe backplane.

Figure 2(b) shows the floorplan developed at the register-transfer level upon a 16-nm FPGA. The host node integrates a CXL controller within a custom RISC-V O3 CPU, while the storage node employs OpenExpress-based nonvolatile memory express storage, emulating a 32-GB Z-NAND device. Alongside the CXL prototype (`CXL`), we evaluated two additional configurations: a local DRAM-only system (`DRAM`) and a PCIe-based memory expander (`PCIe`). Both `PCIe` and `CXL` systems utilize the same backend storage, but their RP addresses are mapped to distinct regions in host memory.

Latency Impact of PCIe Cacheable Region

We evaluated the latency impact using *Apex-Map*,[8] a global memory access benchmark for large-scale

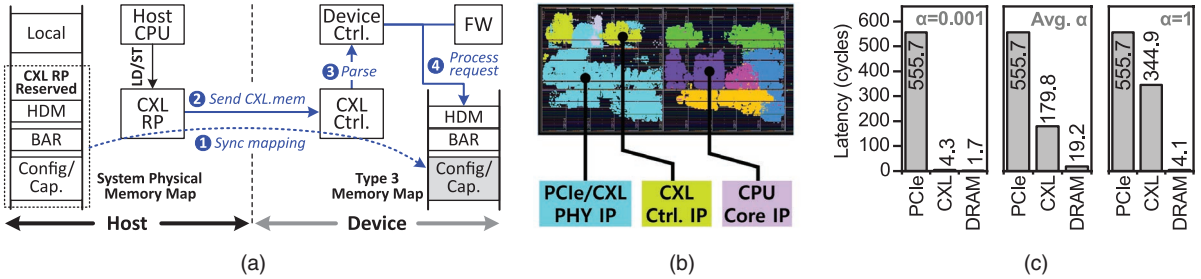


FIGURE 2. CXL-SSD system integration and performance. (a) System memory mapping, (b) CXL-SSD prototype, and (c) performance.

computing. We generated 512 million synthetic memory instructions with varying levels of access locality by controlling the corresponding parameter α from 0.001 (highest locality) to 1 (lowest locality). To focus on performance projection, we excluded time-consuming internal tasks, like garbage collection.

Figure 2(c) presents the latency (measured in CPU cycles) for the best case ($\alpha = 0.001$), average case ($0.001 \leq \alpha \leq 1$), and worst case ($\alpha = 1$). The best-case scenario highlights the significant advantage of CXL over PCIe-based memory expanders. While most memory requests benefit from CPU cache hits, PCIe cannot leverage the host CPU caches, resulting in 129.5× longer latency compared to CXL. In contrast, CXL takes full advantage of CPU caches, achieving latency levels comparable to DRAM.

In the worst-case scenario with no locality, CXL cannot mitigate Z-NANDs latency, making it 84.1× slower than DRAM. Nevertheless, CXL still outperforms PCIe by 1.6×, as it avoids the full synchronization of memory requests required by PCIe BAR. Considering the substantial capacity provided by storage-integrated memory expanders, we posit that CXL can deliver significant benefits for various applications. Moreover, we can optimize these long latencies by effectively leveraging the storage's internal DRAM and backend block media.

INSTRUCTION ANNOTATION

As CXL type 3 is prioritizes memory pooling rather than block storage, two critical challenges arise that warrant further discussion: 1) latency fluctuation and 2) data persistence.

CXL.mem and CXL.io allow asynchronous memory requests without strict load/store turnarounds. However, prolonged latency remains undesirable, as it can degrade overall host performance. Further, the storage device often involves internal tasks, which can significantly worsen the responsiveness. Moreover, when host-side libraries, such as PMDK, require strict data persistence, the current flushing mechanisms provided by CXL (*global persistent flush*) may be insufficient. While it ensures that all data in the CXL network and SSDs' internal DRAMs be written back to the backend persistent media, it introduces additional latency, impacting performance.

To address these issues, we propose two annotation features: 1) *determinism* and 2) *bufferability*. These annotations can be embedded in CXL messages to convey host-level semantics to the underlying CXL controllers. CXL type 3 already accommodates diverse I/O-specific demands via CXL.io, and CXL.mem includes a reserved field suitable for annotations. The proposed annotations minimize overhead without modifying the data payload itself. Instead, they are incorporated into CXL.mem's M2S Req and S2M NDR messages, utilizing the 10-bit reserved fields available at the end of each message.[5] This approach allows the annotations to be seamless without additional transmission costs.

Latency and Persistence Controls

As shown in Figure 3, when memory instructions, such as load and store, arrive, the CXL RP generates the corresponding hints. These hints are appended to the reserved area of M2S Req messages, and the endpoint controller forwards them to the underlying SSD controller. By interpreting these annotations, the SSD controller can reschedule internal tasks, such as garbage collection, or determine whether target data should be cached in its internal DRAM, thereby improving latency and persistence characteristics.

Determinism has two states: deterministic (DT) and nondeterministic (ND). DT requires a type 3 device to process tagged requests without involving internal tasks, thus ensuring predictable performance. In contrast, ND processes requests in a fire-and-forget manner, allowing the device to schedule internal tasks either during subsequent ND requests or idle periods.

We enabled the host system to transparently apply annotations regarding DT to users. SSD tail latency becomes particularly problematic when subsequent instructions depend on specific load instructions. Unlike conventional block read requests for PCIe SSDs, read requests issued to CXL-connected SSDs are memory read requests, inherently limiting the extent of asynchronous handling. In addition, these requests can directly cause CPU pipeline stalls because their latency is significantly longer than typical local memory accesses. Thus, to minimize performance degradation caused by SSD tail latency, it is essential for the CPU to consider the currently executing instructions. However, due to instruction reordering and related techniques, the CPU itself is best suited to evaluate how likely a specific load instruction is to result in pipeline stalls. For this reason, we propose that when the

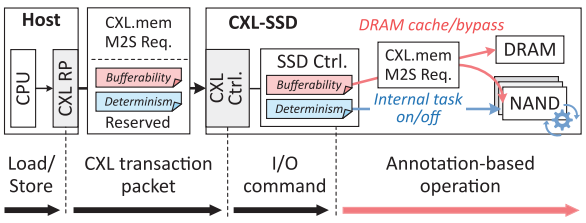


FIGURE 3. Instruction annotation workflow.

proportion of load instructions in the CPU's instruction queue and reorder buffer exceeds a certain threshold, the CPU should issue CXL.mem requests with the DT annotation at runtime. This method can minimize performance degradation caused by SSD tail latency while simultaneously enhancing user convenience.

Bufferability similarly comprises two states: bufferable (BF) and nonbufferable (NB). Requests annotated as BF can be cached or buffered in the SSD's internal DRAM for scenarios in which persistence is unnecessary. This indicates that the data are likely to be accessed repeatedly, ensuring it remains readily available. Conversely, NB requests prioritize persistence, such as when committing database transactions. For such cases, PCIe storage can selectively write these requests directly to block media, avoiding the need to globally flush all data residing in its large internal DRAM simultaneously.

Note that, unlike DT, which relates exclusively to performance, BF depends on application persistence requirements. Therefore, annotations reflecting user intent significantly aid the decision making of the underlying storage. Since this applies only to critical persistence store instructions, we propose enabling CPUs to provide specifically annotated store instructions. Unlike regular store instructions, our store instruction explicitly sends a CXL.mem packet annotated with NB to the CXL-SSD. The SSD then identifies that it should written directly to backend media without buffering. Consequently, when the memory request completes, the host can be certain that the corresponding store has persisted. In database applications, users can apply this store instruction exclusively to memory requests related to transaction journaling, thereby ensuring persistence with minimal overhead.

TABLE 1. Evaluation setup: Simulation setup.

Parameter	Value
CPU	4 GHz, four cores, 64-entry LSQ
L1 cache	I-cache: 64 KB, eight-way, eight-entry MSHR, four-cycle D-cache: 64 KB, 12-way, 16-entry MSHR, five-cycle
L2 cache	2 MB, eight-way, 32-entry, MSHR, 20-cycle
DRAM	tRP = tRCD = tCAS = 12.5 ns two channels, eight ranks, eight banks
CXL	CXL 3.1, PCIe 6.0, 64 GT/s, ×4/×8 bifurcation endpoint device type: type 3
SSD	DRAM cache: tRP = tRCD = 9.1 ns, tRAS = 19 ns NAND flash: Z-NAND, tR = 3 μs, tPROG = 100 μs, tBERS = 1 ms

Since most instructions rely on the operand arrival, load operations typically benefit from DT. However, if there is no subsequent instruction requiring the prior issued result (i.e., no read-after-write dependency), synchronization is unnecessary. In such cases, annotations like BF+DT or BF+ND enable the storage to prefetch data into internal DRAM. For instance, in loop-based code segments (e.g., matrix computations) with spatial and temporal locality, these annotations show data are likely to be accessed repeatedly, ensuring it is available in internal DRAM.

EVALUATION

Evaluation Setup

- *Methodology:* To explore a complete design space, we use a full-system simulation model that covers from the CPU to the block storage. Specifically, the system combines gem59 and SimpleSSD10 for CXL RP and CXL-enabled SSD, and cross-validated with the actual cycles observed in our prototype [Figure 2(b)]. The main configuration parameters that we used for the evaluations are summarized in Table 1.
- *Workloads:* We evaluated 18 workloads sourced from SPEC CPU.11 Their important characteristics are detailed in Table 2. Note that workloads with a misses per kilo instruction (MPKI) greater than 1.2 are identified as exhibiting high cache miss rates.
- *Configurations:* We evaluated three memory expansion systems (`PCIe-SSD`, `CXL-SSD`, and `CXL-ASSD`) against two ideal baseline systems (`DRAM` and `CXL-DRAM`). All three systems (`PCIe-SSD`, `CXL-SSD`, and `CXL-ASSD`) utilize the same underlying media for memory expansion. However, their interfaces differ: `PCIe-SSD` exposes internal memory solely through PCIe BAR,6 making it noncacheable; `CXL-SSD` employs CXL as the media interface, mapping the memory to the host's cacheable system memory; and `CXL-ASSD` integrates internal DRAM management and task scheduling, leveraging BF and DT annotations.

For baseline comparisons, `DRAM` represents DRAM directly attached to the CPU via a conventional double data rate (DDR) interface, while `CXL-DRAM` connects DRAM via CXL interface.

Performance Analysis

Figures 4(a) and (b) present an analysis of the execution time behaviors of PCIe-based storage and three

CXL-based storage configurations, normalized to the execution time of DRAM.

PCIe-Based Storage

PCIe-SSD demonstrates a 406.5× longer execution time than DRAM, on average. This performance gap arises from two factors. First, the SSD controller and flash memory in PCIe-SSDs are optimized for block-based interfaces, which degrade performance in memory-intensive operations due to coarse granularity. Second, every load/store memory request from the host is routed directly to the PCIe BAR, rendering the benefits of on-chip caching ineffective.

The workload-dependent performance variations primarily depend on how the internal DRAM cache is utilized. For instance, load-intensive workloads (e.g., cactus) experience the worst performance because the CPU stalls until the requested data are fully loaded into its registers. In contrast, store-intensive workloads (e.g., astar) suffer less performance degradation since data are buffered in the internal DRAM, allowing the CPU to continue operations without significant

TABLE 2. Evaluation setup: Workload characteristics.

Cache miss	Name	MPKI	Load instance	Store instance	Foot. (GB)	LLC hit ratio	Cache miss	Name	MPKI	Load instance	Store instance	Foot. (GB)	LLC hit ratio
High	gcc	3.08	28%	14%	1	20.1%	Low	hmmer	1.17	12.9%	16.3%	1.1	22.1%
	gobmk	2.38	16.4%	13.8%	2	29.9%		leslie3d	1.16	14.5%	18.5%	12.7	27.4%
	cactus	2.37	23.5%	17.8%	22.7	17.1%		quantum	1.08	21.2%	19.5%	6.8	25.7%
	milc	1.60	21.2%	19%	34.9	21.6%		AES	1.07	8.8%	14.6%	6.9	21.1%
	bzip2	1.5	21.7%	19.7%	96	24.2%		astar	0.88	24.3%	17.1%	0.3	25.4%
	lbm	1.35	16.4%	38.5%	42.2	24.3%		SHA512	0.86	11.7%	4.5%	0.3	22.2%
	sjeng	1.32	15.9%	38.5%	17.8	26.3%		calculix	0.8	26.6%	15.2%	1.0	15.9%
	namd	1.31	22.1%	14.8%	6.	25.7%		povray	0.55	30.4%	12.7%	0.2	4.2%
	—	—	—	—	—	—		tonto	0.54	16.6%	10.9%	0.4	14.2%
	—	—	—	—	—	—		bwaves	0.5	27.6%	9.2%	34.3	17%

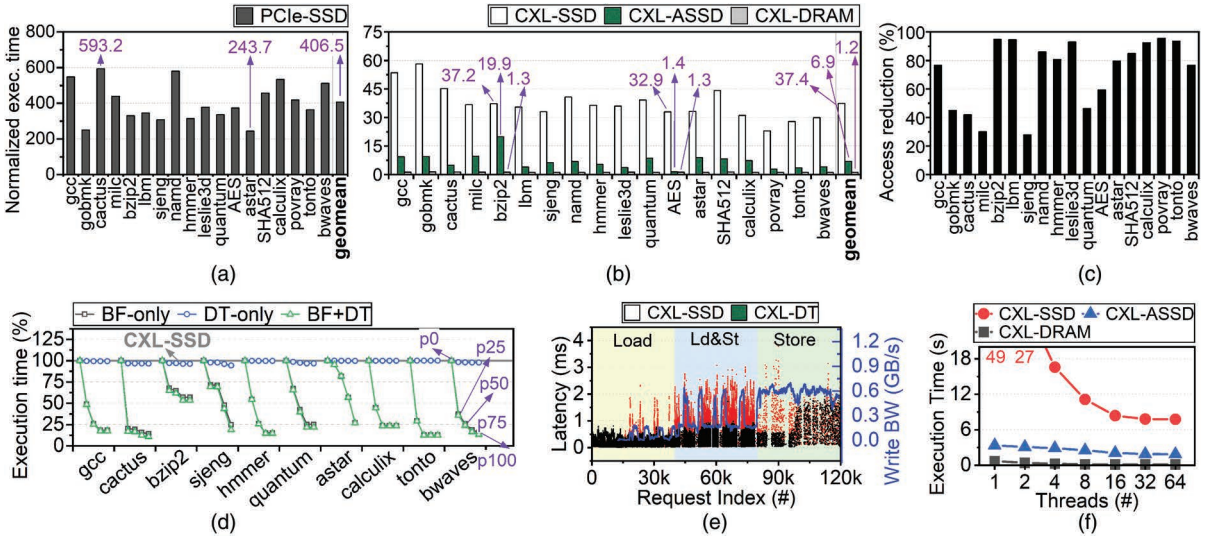


FIGURE 4. Performance analysis. (a) PCIe-based storage performance, (b) CXL-based storage performance, (c) storage access, (d) annotation sensitivity analysis, (e) time-series analysis, and (f) load sensitivity analysis.

interruptions. Nevertheless, PCIe-SSD still struggles with noncacheable region accesses, further impacting its performance.

CXL-Based Storage

Figure 4(b) compares the performance of CXL-SSD, CXL-ASSD, and CXL-DRAM against DRAM. CXL-SSD achieves an average performance improvement of 10.9× over PCIe-SSD. Although the underlying storage media and characteristics remain identical, CXL-SSD benefits from its placement in cacheable memory space via CXL, reducing storage access frequency by 72.1% [see Figure 4(c)], and enhancing performance.

Workload locality, further enhances the performance of CXL-SSD. Workloads with low cache miss rates (e.g., bwaves, tonto, povray) reduce storage accesses by 80.2%, improving performance by 12.2×. In contrast, workloads with high cache miss rates reduce storage accesses by 62.1%, leading to 9.6× improvement on average. These findings underscore the importance of leveraging CXL's cacheable memory architecture in improving overall execution efficiency.

CXL-ASSD improves average performance by 5.4× over CXL-SSD. While the total number of storage accesses remains unchanged between CXL-ASSD and CXL-SSD, the BF/NB and DT/ND annotations allow the underlying CXL and SSD controllers to better utilize internal DRAM. This optimization effectively hides the long latency associated with flash media.

Note that CXL-DRAM demonstrates performance approaching that of DRAM. Although CXL-DRAM uses a slightly slower PCIe interface than high-speed DDR, application locality and on-chip caching help mitigate this disadvantage.

Sensitivity Tests

While the previously discussed CXL-ASSD applies annotations to all commonly used functions, annotating broadly may not always be practical. Our analysis shows that even with fewer annotations, CXL-ASSD still achieves comparable performance. To investigate this further, we evaluated CXL-ASSD under varying degrees of annotation coverage. In addition, we examined how BF and DT annotations affect performance both individually and jointly.

For DT, we adjusted the threshold of load instructions in the instruction queue/reorder buffer at which the CPU begins applying DT annotations, thereby annotating a specific percentage of total instructions. For BF, we varied the proportion of functions annotated as BF. The results, shown in Figure 4(d), present execution time normalized to that of CXL-SSD. BF-only and DT-only apply a single type of annotation, while BF+DT applies both types together. The pN labels (e.g., p25, p50, etc.) represent the performance achieved when *N*% of the total instructions (for DT) or functions (for BF) are annotated.

As shown, annotating only 25% of commonly used functions reduces average execution time by 50.1%. This occurs because few functions dominate storage accesses; for instance, just eight functions (e.g., _raw_spin_lock) account for 50.5% of total storage accesses. Although workloads like *bzip2* initially show limited performance gains, additional per application analyses can effectively address these cases. Specifically annotating clear_page_erms, the primary storage access source for *bzip2*, reduces its execution time by 77.4%.

The results further indicate that BF annotations contribute most significantly to enhance CXL-ASSD performance. This is because BF-only configurations consistently leverage the internal DRAM of SSDs to reduce latency, while DT-only configurations mitigate the long-tail latency issue. Nevertheless, DT annotations remain essential for maintaining stability, as tail latency occasionally reaches several milliseconds. Combining BF and DT annotations provides a balanced approach, enhancing overall efficiency and reliability.

Figure 4(e) shows how DT annotations mitigate SSD tail latency. To evaluate this, we compared a load-intensive workload, cactus, under two conditions: without DT annotations (CXL-SSD) and with DT annotations (CXL-DT). The workload cactus initially exhibits a load-intensive behavior, gradually becoming store-intensive pattern over time. Memory write bandwidth over time is indicated by the right *y*-axis [blue line in Figure 4(e)]. For this evaluation, the CPU configuration matched that of the p75 scenario in Figure 4(d), meaning the threshold was set to annotate approximately 75% of all instructions with DT. The figure shows memory latency experienced by each load/store instruction, presented in execution order.

The evaluation shows CXL-SSD experiences prolonged tail latencies from random SSD-internal tasks, even with a high proportion of load instructions. These millisecond-scale memory delays cause previously unseen, prolonged CPU pipeline stalls, significantly degrading performance. In contrast, CXL-DT avoids such issues by enabling the CPU to monitor the proportion of load instructions, providing hints to suppress internal SSD activities. As a result, CXL-DT successfully avoids tail latency throughout the entire load-intensive period (until the 90,000th instruction). However, DT annotations cannot fundamentally

eliminate SSD tasks. As the application transitions to a store-intensive pattern around the 90,000th instruction, the CPU stops sending DT annotations, allowing internal SSD tasks to resume, causing long tail latencies to reappear. Thus, although CXL-DT still encounters long tail latencies, these occur only when the application is store-intensive. Under these conditions, the CPU effectively avoids performance degradation due to pipeline stalls, preserving the efficiency of the fire-and-forget policy.

BY LEVERAGING CXL'S CACHEABILITY AND ADOPTING TYPE 3 ENDPOINT DEVICES (CXL-SSDS), WE BRIDGE BLOCK-BASED PCIE SEMANTICS WITH MEMORY-COMPATIBLE OPERATIONS.

On the other hand, Figure 4(f) illustrates the latency and bandwidth characteristics of CXL memory expansion according to workload type. For evaluation, we utilized the STREAM microbenchmark suite,[12] with vector kernels (copy, scale, add, triad), and varied host threads across three configurations: CXL-SSD, CXL-ASSD, and CXL-DRAM. For the CXL-ASSD, we adopted the same p75 settings used in Figure 4(d), applying both BF and DT annotations. The results show that as the number of workload threads decreases, the performance gap between CXL-SSD and CXL-ASSD widens. This occurs because fewer threads become more sensitive to latency rather than bandwidth provided by CXL memory. Consequently, by leveraging the SSD's internal DRAM, CXL-ASSD achieves up to 14.6× higher performance compared to CXL-SSD.

In contrast, as the thread count increases, application performance relies more on CXL memory bandwidth, reducing the performance gap between CXL-SSD and CXL-ASSD. Nevertheless, even with 64 threads, CXL-ASSD still achieves a 4.2× performance improvement over CXL-SSD. This is because the internal DRAM within the SSD provides significantly higher bandwidth than Z-NAND. Thus, despite being Z-NAND based memory expansion, CXL-ASSD performance is 4.7× lower than CXL-DRAM.

CONCLUSION

This article explores how CXL can transform PCIe-based block storage into scalable, byte-addressable working memory. By leveraging CXL's cacheability and adopting type 3 endpoint devices (CXL-SSDs), we bridge block-based PCIe semantics with memory-compatible operations. To narrow the performance gap with DRAM, we propose annotation mechanisms, DT and BF, to improve performance while maintaining data persistence. Our FPGA-based prototype and simulation demonstrate that CXL-SSDs significantly outperform PCIe-based memory expanders and approach DRAM-like performance in workloads with high locality, demonstrating the feasibility of integrating block storage into CXL's memory ecosystem.

ACKNOWLEDGMENTS

The work is supported in part by the Institute of Information and Communications Technology Planning and Evaluation's (IITP's) RS-2023-00221040, RS-2024-00460762, RS-2025-02214652, RS-2025-02214654, and RS-2025-02263080; Korea Institute for Advancment of Technology's (KIAT's) P0027923 and P0028225; and Ministry of SMEs and Startups (MSS) Technology development Program's RS-2023-00303967. The authors thank anonymous reviewers for their constructive feedback. This work is protected by one or more patents.

REFERENCES

1. "Gen-Z consortium." Gen-Z. [Online]. Available: https://genzconsortium.org
2. "CCIX consortium." CCIX. [Online]. Available: https://www.ccixconsortium.com/
3. S.-P. Yang et al., "Overcoming the memory wall with CXL-enabled SSDs," in *Proc. USENIX Conf. Annu. Tech. Conf.*, 2023, pp. 601–617.
4. H. Chen et al., "ICGMM: CXL-enabled memory expansion with intelligent caching using Gaussian mixture model," in *Proc. 61st ACM/IEEE Des. Automat. Conf. (DAC)*, 2024, pp. 1–6, doi: 10.1145/3649329.3656239.
5. "CXL® specification." Compute Express Link. [Online]. Available: https://computeexpresslink.org/cxl-specification/
6. D.-H. Bae et al., "2B-SSD: The case for dual, byte- and block-addressable solid-state drives," in *Proc. ACM/IEEE 45th Annu. Int. Symp. Comput. Archit. (ISCA)*, Los Angeles, CA, USA, 2018, pp. 425–438, doi: 10.1109/ISCA.2018.00043.
7. J. Hermes, J. Minor, M. Wu, A. Patil, and E. Van Hensbergen, "UDON: A case for offloading to general purpose compute on CXL memory," 2024, *arXiv:2404.02868*.
8. E. Strohmaier and H. Shan, "Apex-Map: A global data access benchmark to analyze HPC systems and

parallel programming paradigms," in *Proc. ACM/IEEE Conf. Supercomput. (SC)*, Seattle, WA, USA, 2005, pp. 49–49, doi: 10.1109/SC.2005.13.
9. N. Binkert et al., "The gem5 simulator," ACM SIGARCH Comput. Archit. News, vol. 39, no. 2, pp. 1–7, 2011, doi: 10.1145/2024716.2024718.
10. D. Gouk et al., "Amber: Enabling precise full-system simulation with detailed modeling of all SSD resources," in *Proc. 51st Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Fukuoka, Japan, 2018, pp. 469–481, doi: 10.1109/MICRO.2018.00045.
11. J. L. Henning, "SPEC CPU2006 benchmark descriptions," *ACM SIGARCH Comput. Archit. News*, vol. 34, no. 4, pp. 1–17, 2006, doi: 10.1145/1186736.1186737.
12. J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," in *Proc. IEEE Tech. Committee Comput. Archit. Newslett.*, 1995, pp. 19–25.

MIRYEONG KWON is currently the chief strategy officer of the Next-Generation Silicon and Research Division, Panmnesia, Inc., Daejeon, 34136, South Korea. Her research interests include edge computing, on-device artificial intelligence, and link solutions, including compute express link, UALink, and ethernet. Kwon received her Ph.D. degree at the Korea Advanced Institute of Science and Technology, Daejeon, South Korea. She is a Member of IEEE. Contact her at mkwon@panmnesia.com.

DONGHYUN GOUK is the chief architect officer of Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include computing switch/network, artificial intelligence infrastructure, RISC-V, and link solutions, including compute express link, UALink, and ethernet. Contact him at donghyun@panmnesia.com.

JUNHYEOK JANG is a director at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include machine learning, nonvolatile memory express, reliability, security, and link solutions, including compute express link, UALink, and ethernet. Contact him at jhjang@camelab.org.

JINWOO BAEK is a senior professional at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include machine learning acceleration, computing switch/network, artificial intelligence infrastructure, and link solutions, including compute express link, UALink, and ethernet. Contact him at jwbaek@camelab.org.

HYUNWOO YOU is a senior professional at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include network/storage, edge computing, RISC-V, and link solutions, including compute express link, UALink, and ethernet. Contact him at airplasma@camelab.org.

SANGYOON JI is a senior professional at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include machine learning acceleration, artificial intelligence infrastructure, and link solutions, including compute express link, UALink, and ethernet. Contact him at syoon@camelab.org.

HONGJOO JUNG is an assistant director at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include machine learning acceleration, artificial intelligence infrastructure, and link solutions, including compute express link, UALink, and ethernet. Contact him at hjjung@panmnesia.com.

JUNSEOK MOON is an assistant director at Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include network/storage, edge computing, artificial intelligence infrastructure, RISC-V, and link solutions, including compute express link, UALink, and ethernet. Contact him at jsmoon@panmnesia.com.

SEUNGKWAN KANG is working toward his Ph.D. degree at the Korea Advanced Institute of Science and Technology, Daejeon, 34141, South Korea. His research interests include storage-integrated accelerator. Contact him at kangsk@camelab.org.

SEUNGJUN LEE is working toward his Ph.D. degree at the Korea Advanced Institute of Science and Technology, Daejeon, 34141, South Korea. His research interests include operating systems and storage stack optimization. Contact him at sjlee@camelab.org.

MYOUNGSOO JUNG is a full professor in the School of Electrical Engineering, Korea Advanced Institute of Science and Technology, Daejeon, 34141, South Korea, and is also the chief executive officer of Panmnesia, Inc., Daejeon, 34136, South Korea. His research interests include computer architecture, operating systems, storage systems, nonvolatile memory, parallel processing, heterogeneous computing, edge computing, artificial intelligence infrastructure, and link solutions, including compute express link, UALink, and ethernet. He is the corresponding author of this article. Contact him at mj@panmnesia.com.

Ginkgo: A Learned-Index Enhanced Tiered Memory System

Xiran Yang, *Peking University, Beijing, 100871, China*
Yifei Yu, *Huazhong University of Science and Technology, Wuhan, 430074, China*
Chuandong Li and Jianqiang Zeng, *Peking University, Beijing, 100871, China*
Ke Zhou, *Huazhong University of Science and Technology, Wuhan, 430074, China*
Diyu Zhou and Xiaolin Wang, *Peking University, Beijing, 100871, China*
Zhenlin Wang, *Michigan Technological University, Houghton, MI, 49931, USA*
Yingwei Luo, *Peking University, Beijing, 100871, China*

Compute Express Link (CXL), as an emerging high-speed interconnect protocol, offers a promising approach to memory expansion. Organizing fast double data rate (DDR) dynamic random-access memory (DRAM) and large CXL memory as a tiered memory system is preferred to obtain the large memory capacity while maintaining high memory performance. Additionally, such a tiered memory system also adopts huge pages to alleviate address translation overhead. However, due to memory bloating issues in huge pages, existing hardware-based tiered memory management systems suffer from severe cache conflicts and DDR DRAM underutilization, resulting in significant performance degradation. We introduce Ginkgo, a learned-index enhanced tiered memory system, which builds a distribution-aware cache index mechanism while leveraging the learned index to minimize metadata overhead. Evaluation reveals that Ginkgo reduces average memory access latency with an average of 3.1% and 13.5% compared to state-of-the-art Alloy Cache and Unison Cache, respectively.

With the rapid advances of modern large-scale memory-intensive applications, such as artificial intelligence and graph computing, there are huge demands for memory capacity. However, traditional double data rate (DDR) dynamic random-access memory (DRAM) has inherent drawbacks in cost efficiency and scalability, making it difficult to expand the memory capacity. An emerging cache-coherent high-speed interconnect protocol, called Compute Express Link (CXL), is a promising approach to address this issue. Specifically, it allows the host CPU to access CXL device-attached memory (CXL memory) via load/store instructions. The CXL memory decouples memory technologies from the specific memory interface standard, enabling memory resource disaggregation. The CXL-based memory disaggregation provides a flexible and promising way to expand memory capacity. However, due to additional protocol and data transfer overheads, the access latency of CXL memory still lags behind that of traditional DDR DRAM. To achieve both large capacity and high performance, a common practice is to organize DDR DRAM and CXL memory as a tiered memory system.

A tiered memory system can be managed by either OS or hardware. Compared to OS-based schemes, hardware-based schemes support fine-grained data management and eliminate software management overhead inherent in OS-based schemes. Thus, we

Digital Object Identifier 10.1109/MM.2025.3564395
Date of publication 30 April 2025; date of current version 24 December 2025.

focus on the hardware-based schemes in this work. In the traditional hardware-based management scheme, DDR DRAM functions as a cache of CXL memory, also referred to DRAM cache. The core of managing the tiered memory is the hardware-based index mechanism, which defines how cache blocks map from the CXL memory to the DRAM cache.

Considering its superior capacity, the tiered memory system adopts *huge pages* to alleviate the address translation overhead. Unfortunately, due to the large page size, several cache blocks within huge pages remain untouched during the execution of a program, referred to *huge page memory bloating*. Since most hardware index mechanisms use fixed mapping schemes and are agnostic to memory access patterns, memory bloating leads to severe cache conflicts and DRAM resource waste. Our research goal is to build a distribution-aware and lightweight index mechanism which alleviates the impacts of memory bloating when managing the DRAM cache.

To this end, the key challenge is minimizing the index-related metadata storage overhead due to the limited resources in the memory controller. To be specific, when a memory request arrives, the memory controller calculates where the target data are located, either in DDR DRAM or CXL memory, by referring to the index-related metadata. If the metadata size is too large to fit in the memory controller, the spilled metadata are placed in DDR DRAM. The memory controller has to fetch both metadata and cache block data from DDR DRAM, resulting in increased DRAM access latencies. Existing works[3,4] based on traditional index structure either suffers from higher DRAM access latency or severe cache conflicts. Recent advances in learned index techniques provide a new opportunity to minimize the aforementioned overhead. By training from the data distribution, the learned index achieves high cache hit ratio and extremely low storage overhead compared to traditional index structure.

In this work, we propose *Ginkgo*, a learned-index enhanced tiered memory system. To improve the DRAM cache utilization and reduce cache conflict misses among huge pages, Ginkgo allocates DRAM spaces of different sizes to each huge page according to the cache block distribution. In addition, Ginkgo leverages the learned index to approximate the distribution and build the mapping from the CXL memory to the DRAM cache, reducing index-related metadata storage overhead. Ginkgo consists of two components: an offline learned index construction tool and an online hardware-based tiered memory management scheme.

Specifically, Ginkgo adopts a static profiling approach to collect the memory access trace and analyze the cache block distribution. Ginkgo utilizes the piecewise linear function (PLF) as the model of the learned index and constructs the learned index offline using a second derivative segmentation algorithm based on the data distribution. When managing the tiered memory system online, the memory controller dynamically allocates varying amounts of DRAM space to different huge pages according to the cache block distributions. Meanwhile, the memory controller calculates the position of the target DRAM cache block based on the learned index with low overhead. We implement Ginkgo in a tiered memory simulator and evaluate its performance in BTree and PR benchmarks. Our experiments reveal that Ginkgo reduces the average memory access latency with an average of 3.1% and 13.5% compared to state-of-the-art Alloy Cache[3] and Unison Cache,[4] respectively.

The main contributions in this work can be summarized as follows:

- an in-depth analysis of the performance impact of memory bloating on tiered memory management schemes
- an offline approach to profile applications and construct the learned index
- a distribution-aware and lightweight tiered memory management system enhanced by learned index.

BACKGROUND

CXL-Based Tiered Memory

CXL is a high-speed interconnect protocol built from the standard PCIe bus, which enables cache coherence between any CXL devices, including CPUs and memories. CXL allows the host to access device-attached memory through load/store instructions. A CXL-enabled memory device is referred to as *CXL memory*. It decouples memory technologies from the specific memory interface standard, enabling memory resource disaggregation. The CXL-based memory disaggregation provides a flexible and promising way to expand memory capacity. However, the access latency of CXL memory lags behind that of DDR DRAM because of the additional protocol overhead and the data transmission latency of the serial PCIe interface. To avoid performance degradation and fully reap the benefits from CXL memory in terms of both capacity and scalability, a typical approach is to organize DDR DRAM and CXL memory as a tiered memory system.

Tiered Memory Management Schemes

The management schemes of tiered memory systems can be primarily categorized as OS-based and hardware-based ones. We focus on the hardware-based schemes in this work.

In hardware-based schemes, DDR DRAM is organized as a fast cache layer of CXL memory and managed by the memory controller. When a memory request arrives, the memory controller first attempts to look up the DRAM cache, returning data if it hits. Otherwise, it accesses the CXL memory. For each DRAM cache lookup, the memory controller calculates where the target data are located by referring to the index-related metadata and then checks the tag of the cache block. If the index-related metadata size is too large to fit in the memory controller, the spilled metadata is placed in the DRAM cache. The memory controller has to fetch both the index-related metadata and the cache block data from the DRAM cache, leading to higher DRAM access latency. This is referred to as *double-read problem*.

DUE TO THE FIXED MAPPING SCHEME, ALLOY CACHE AVOIDS THE INDEX-RELATED METADATA WHILE SUFFERING FROM SEVERE CACHE CONFLICTS.

Due to the limited resources in the memory controller, existing indexing mechanisms fail to achieve the low DRAM cache miss ratio and eliminate double-read problems simultaneously. Alloy Cache[3] adopts a direct-map-based index mechanism, which maps each cache block to a fixed address. Due to the fixed mapping scheme, Alloy Cache avoids the index-related metadata while suffering from severe cache conflicts. Unison Cache[4] leverages set-associative-based index mechanism. It organizes DRAM into sets, where each cache block can be mapped into any block within a specific set. Although it reduces cache conflicts compared to Alloy Cache, Unison Cache requires additional index-related metadata to identify the exact position of a cache block within a given set. The performance challenge for Unison Cache is the double-read problem resulting from the significant metadata storage overhead.

Huge Pagevs in Tiered Memory

The operating system uses the page table to record the mapping between virtual and physical addresses. The page table is typically organized in a multilevel structure and stored in memory. During address translation, the operating system has to access every level of the page table, which is called *page walking*, resulting in additional memory access and performance degradation. Although a translation look-aside buffer (TLB) is used in hardware to cache the virtual-physical address mappings in order to speed up the address translation, the large capacity of tiered memory results in severe TLB misses.

The tiered memory system adopts huge pages to alleviate address translation overhead. By extending the page size from 4 kB to 2 MB, the huge page reduces the page walking overhead and increases the TLB coverage. Unfortunately, due to large page size, several cache blocks within huge pages remain untouched during the execution of a program, referred to as *memory bloating* in huge pages.

MOTIVATION

In this section, we first analyze the memory bloating problem and its impact on different tiered memory management schemes. Then, we explore the opportunity to apply the learned index to huge pages. We leverage Valgrind to get the memory trace, which is inspired by previous work.[5] It is worth noting that our experiments do not collect memory traces from the entire lifetime of applications but rather focus on the core execution phase.

Impact of Memory Bloating on Tiered Memory Management Schemes

We first demonstrate the degree of memory bloating in different applications. We calculate the memory footprints of different applications when using various page granularities and demonstrate the results in Figure 1(a). Compared to the 64-B baseline, using 2-MB huge pages results in an average increase in memory footprint of 3.85x, with a maximum increase of 5.6x. This indicates that memory bloating in huge pages is a common and severe problem in different types of applications. Furthermore, considering that the memory footprint increases by an average of 2.61x even when using 4 K base pages, and the 4 K page is the smallest unit of memory management in the operating system, it is hard for OS-based management schemes to address the memory bloating issue.

Regarding the hardware-based management scheme, it adopts a finer granularity (i.e., 64-B cache line) to manage the DRAM cache and has an advantage over OS-based schemes in addressing the memory bloating problem. To quantify the impact of the

memory bloating problem on hardware-based management schemes, we carried out experiments on a hardware-based management scheme using a direct-map based index mechanism and set the DRAM capacity to be 25% of the memory footprint of applications. The DRAM underutilization (i.e., the fraction of the untouched memory space in the DRAM cache) is shown in Figure 1(b). There is an average of 10.23% waste of the DRAM cache among different benchmarks, which indicates that the memory bloating leads to severe DRAM resource waste. We then construct an ideal index scheme, which ensures that each accessed cache block is mapped to the DRAM cache with minimal conflicts. Figure 1(c) demonstrates the miss ratio reduction compared to the direct-map based index scheme. The ideal index scheme eliminates almost all cache conflict misses for benchmarks such as BTree, CC, and PR. Meanwhile, the ideal index scheme has an average of 15.53% reduction of miss ratio compared to the direct map. Considering the memory bloating, there is still room for significant performance improvements in hardware-based management schemes.

Learned Index Construction Based on Cache Block Distribution

Understanding the cache block distribution can aid and guide the design of hardware-based index mechanisms. However, the distribution varies across different workloads and different pages within each workload. It is challenging to approximate the distribution with high accuracy and low storage overhead.

Recent advances in the learned index provide a new opportunity to address the aforementioned challenge, which is an emerging technique in the field of database. Traditional tree-based and table-based index data structures tend to have high storage overheads, which record the complete key-to-position mappings. In contrast, the learned index builds a model to approximate the data distribution, and it calculates all of the data positions based on this model, along with several parameters. To be specific, the learned index approximates the cumulative distribution function (CDF) of the key-to-position mappings. As a result, the learned index achieves extremely low storage overhead.

The regularity of data distribution determines the effectiveness of the learned index. We found that multiple workloads, such as BTree and PR, have relatively regular cache block distributions within huge pages. Specifically, the cache blocks exhibit either contiguous distribution or highly uniform distribution within huge pages. Approximating the cache block distributions in these benchmarks can be accomplished with a maximum of 32 PLFs for each huge page.

DESIGN

System Design Principle

The design principle of Ginkgo is illustrated in Figure 2. In contrast to the traditional index mechanism, which maps the entire huge page from CXL memory to the DRAM cache, Ginkgo maps different huge pages in CXL memory to disjoint spaces in the DRAM cache, which eliminates the cache conflicts among different huge pages. However, the huge page with memory bloating typically exhibits a nonuniform distribution of accessed cache blocks. It leads to DRAM underutilization and cache conflicts inside the huge page. To address this issue, Ginkgo divides each huge page in CXL memory into segments and allocates different DRAM space budgets to segments based on the accessed cache block distribution. To minimize the metadata storage overhead, Ginkgo uses the learned index to approximate distribution and record the mapping scheme from CXL memory to the DRAM cache.

System Overview

Figure 3 illustrates the system overview of Ginkgo, which consists of two components: an offline learned

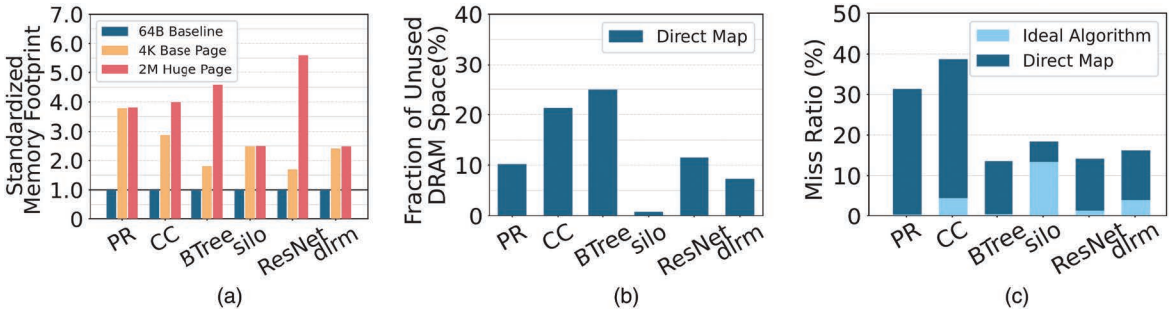


FIGURE 1. Impact of memory bloating on hardware-based management schemes.

index construction tool and an online hardware-based tiered memory management scheme.

Ginkgo first adopts a static profiling approach to collect the memory access trace for each application (1.1). It then exploits the memory access trace to calculate the CDF of the cache blocks within each huge page. Based on the distribution, Ginkgo adopts a PLF as the model of the learned index, and it builds the learned index offline by using a second derivative segmentation algorithm (1.2). At last, Ginkgo calculates the size of memory space allocated to each huge page in CXL based on the learned index (1.3). The learned index metadata is placed into a dedicated area of the DRAM cache before the application starts to execute.

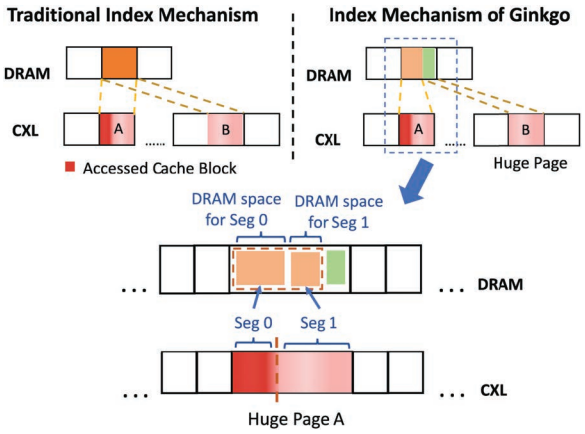


FIGURE 2. Design principle of Ginkgo.

TO BALANCE THE METADATA OVERHEAD AND THE DISTRIBUTION APPROXIMATION ACCURACY, WE BUILD A SEPARATE LEARNED INDEX FOR EACH HUGE PAGE.

Regarding the online management of the tiered memory system, the memory controller first loads the learned index metadata into its internal SRAM buffer and allocates the DRAM space to each huge page (2.1). For each memory request, the memory controller calculates the DRAM cache block address based on the learned index and then looks up the target cache block (2.2). The memory controller fetches cache block data from the DRAM cache if hit (2.3). Otherwise, it fetches data from the CXL memory.

Offline Learned Index Model Training

Before the execution of applications, we analyze the cache block distributions in huge pages and train the learned index offline. To balance the metadata overhead and the distribution approximation accuracy, we build a separate learned index for each huge page.

First, we use an instrumentation-based tool, Valgrind,[5] to collect memory traces for every application. Then, we analyze the distributions of cache blocks that were actually accessed within each huge page, preparing for the subsequent learned index construction.

Considering the limited computational and storage resources in the memory controller, we use a lightweight data structure called PLF as the model of the learned index. The intuition behind the PLF is using multiple linear segments to represent the distribution curve. Compared to models based on neural networks, such as recursive-model indexes,[6] the PLF model has lower computational and storage overhead while providing considerable accuracy.

Inspired by prior work,[7] we use a second derivative segmentation algorithm to train the PLF model. The principle of this algorithm is shown in Figure 4. In order to better approximate the distribution curve with limited linear segments, we consider the points where the slope changes most significantly (i.e., with a higher second derivative) to be the segmentation points of

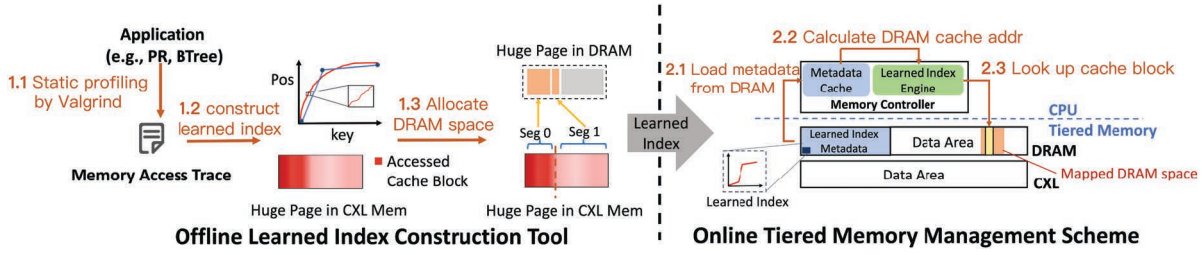


FIGURE 3. System overview of Ginkgo.

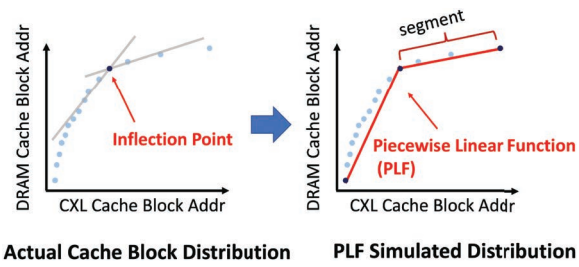


FIGURE 4. Second derivative segmentation algorithm.

the PLF model. To be specific, we first sort the cache blocks by their addresses and calculate their index number within the huge page. The cache block address and index number represent the *x*- and *y*-axis values, respectively. Then, we calculate the changing rate of the linear function slope (second derivative) at each cache block and identify the inflection points with the highest changing rate. Finally, we construct PLF based on these inflection points.

Offline Distribution Guided DRAM Resource Allocation

To reduce the DRAM resource waste and the cache conflict misses caused by memory bloating, Ginkgo allocates DRAM resources to each huge page in CXL memory based on the cache block distribution. For convenience, we refer to the huge pages located in the DRAM cache and the CXL memory as DRAM huge pages and the CXL huge pages, respectively.

Considering that only a limited number of cache blocks are accessed due to memory bloating, most DRAM huge pages are sufficient to accommodate all of the accessed cache blocks while sparing additional free DRAM space. Evaluation reveals that for applications like BTree and PR, more than 80% of DRAM huge pages have the additional free DRAM space, referred to as *memory surplus*. Furthermore, the DRAM huge page with memory surplus can be divided into multiple disjoint spaces, each of which is sufficient to cover all of the accessed cache blocks within the CXL huge pages. Inspired by the aforementioned insights, Ginkgo allocates dedicated DRAM space to each CXL huge page and ensures that the DRAM spaces are distinct and nonoverlapping to avoid cache conflicts among different CXL huge pages.

However, directly mapping the entire CXL huge page to the DRAM space leads to cache conflicts inside the CXL huge page. To address this issue, Ginkgo adopts a distribution-guided DRAM resource allocation scheme. Ginkgo divides each CXL huge page into multiple segments and allocates DRAM spaces to each segment based on the accessed cache block numbers. To be specific, each segment in a CXL huge page corresponds to a segment of the linear function in the PLF model, and the size of allocated DRAM space (i.e., the accessed cache block within segments) can be calculated by using the PLF model. It is worth noting that the allocation scheme is only applied to the DRAM huge pages with memory surplus to avoid additional cache conflict misses.

Online Hardware-Based Index Mechanism

Ginkgo places the index-related metadata into a dedicated area of the DRAM cache before the application starts execution. The memory controller first loads the metadata into its internal SRAM buffer and indexes the cache block online. The cache block indexing consists of two steps: finding the mapped DRAM space and locating the cache block address within the DRAM space. Following the offline allocation scheme mentioned previously, the CXL huge page is divided into several segments, and each segment is mapped to a dedicated DRAM space.

First, in order to find out the mapped DRAM space of segments, the memory controller first reads the metadata to get the location of the mapped DRAM space at the huge page level. Then, the memory controller uses the learned index to locate the mapped DRAM space at the segment level. It first determines which PLF the current cache block is in and then calculates the location of mapped DRAM space based on the slope and intercept of the PLFs. Since the computation of learned index involves only limited multiplication and addition operations, the hardware implementation of learned index incurs minimal computation overhead. To manage each allocated DRAM space, the memory controller adopts a direct-map-based scheme to map from the CXL segment to the allocated DRAM space, which is simple and relatively effective. By taking the modulus of the cache block address in CXL memory with the DRAM space size, the memory controller calculates the exact address of the DRAM cache block. Finally, the memory controller looks up the cache block and fetches the data.

EVALUATION

Experiment Setup

To evaluate the performance of different index mechanisms, we implement a trace-based tiered memory simulator. We collect memory traces from various

TABLE 1. System configurations of tiered memory system.

Memory controller SRAM size	64 kB
DRAM access latency (row buffer hit)	80 ns
DRAM access latency (row buffer miss)	100 ns
CXL memory access latency	200 ns

workloads by leveraging Valgrind.[5] We then simulate the memory access behaviors of these workloads in the simulated tiered memory system. Note that Ginkgo is designed to optimize performance only for benchmarks with a regular cache block distribution. We select two representative workloads, including graph analysis (PR) and database (BTree). In real-world tiered memory configurations, the DDR DRAM and CXL memory capacities are typically configured to a fixed ratio. In our experiments, we adopt a 1:4 ratio of DDR DRAM to CXL memory, and the size of the CXL memory is determined by the memory footprint of the workload. The CXL memory capacity of BTree and PR is 32 GB and 16 GB, respectively. Table 1 details the system configurations of the tiered memory system.

We compare Ginkgo to state-of-the-art hardware-based tiered memory management schemes: Alloy Cache and Unison Cache. For the index mechanism, we compare the direct map used by Alloy Cache and the four-way set associativity adopted by Unison Cache. To avoid the impact of different cache block granularities on the miss ratio results, we standardize the management unit to 64-B cache line for all three systems. Regarding the metadata overhead, since Unison Cache introduces additional hardware support to pass the PC information to the memory controller, we disable the PC-based features in Unison Cache to ensure fairness in the evaluation.

DRAM Cache Miss Ratio

We measure the DRAM cache miss ratio of different index mechanisms. The results are shown in Figure 5(a). `Ginkgo` outperforms `Alloy Cache` across all benchmarks and reduces the miss ratio by an average of `5.24%`. `Alloy Cache` suffers from severe cache conflicts due to its simple and fixed mapping scheme, which is based on the direct map. For benchmarks with relatively regular distribution patterns, such as `BTree`, `Ginkgo` achieves a similar miss ratio compared to the four-way set associativity used by `Unison Cache`. For benchmarks with less regular cache block distribution, such as `PR`, `Ginkgo` increases `10.35%` miss ratio compared to `Unison Cache` due to the inaccuracy when approximating the distribution. However, `Unison Cache` reduces miss ratio at the cost of severe double-read problems, while `Ginkgo` eliminates most of the double-read problems.

Double-Read Problem

We first calculate the storage overhead of index-related metadata. There is no index-related metadata for `Alloy Cache` since it is based on the direct map, which calculates the cache block address by directly taking the modulus of the DRAM cache capacity. The metadata storage overhead of Ginkgo is around `10 B–24 B` per huge page and is `1000x` lower than that of `Unison Cache`. Specifically, the metadata storage overhead of `Unison Cache` is `512 MB` and `256 MB` for `BTree` and `PR`, respectively. For `Ginkgo`, the metadata storage overhead is `80 kB` and `360 kB` for `BTree` and `PR`, respectively. `Ginkgo` builds the learned index at the granularity of the huge page, which reduces metadata storage overhead effectively.

Figure 5(b) demonstrates the fraction of memory requests that experience the double-read problem in different benchmarks. The SRAM buffer in the

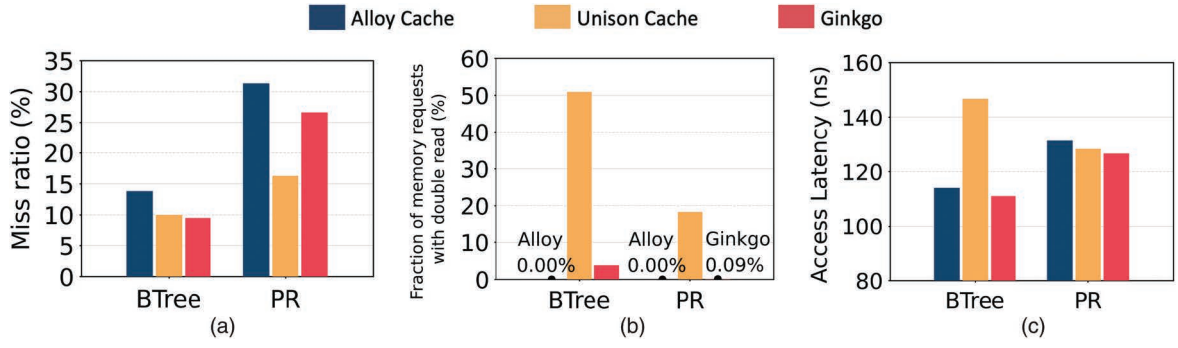


FIGURE 5. System performance of different hardware-based management schemes.

memory controller can accommodate most of the index-related metadata of Ginkgo, thus eliminating almost all additional DRAM access. Due to large metadata storage overhead, Unison Cache suffers from severe double-read problems. For Unison Cache, there are 50.92% and 18.29% memory requests that incur additional DRAM access in BTree and PR.

Given that the metadata overhead will grow with the number of huge pages. We evaluate the metadata scalability of Ginkgo. We set up the SRAM capacity as 16 kB and calculate the double-read fraction. Experimental results reveal that Unison Cache suffers from severe double-read problems, and the double-read fraction is 60.64% and 18.28% for BTree and PR, respectively. However, Ginkgo incurs an 8.72% double-read fraction in BTree and almost eliminates double-read in PR. Due to the extremely low storage overhead and locality of memory access, the Ginkgo is scalable even if the metadata storage overhead exceeds the memory controller SRAM capacity.

Average Memory Access Latency

We measure the average memory access latency to represent the system performance of the tiered memory system. The average memory access latency is calculated as follows:

$$Lat_{avg} = Hit_{DRAM} \times Lat_{DRAM} + Miss_{DRAM} \times Lat_{CXL}$$

where the Lat_{avg}, Lat_{DRAM}, and Lat_{CXL} are the access latencies of average memory access, DRAM cache access, and CXL memory access, respectively, the Hit_{DRAM}, and $Miss_{DRAM}$, are the hit and miss ratios of the DRAM cache.

Figure 5(c) shows the average memory access latency of different workloads. Ginkgo reduces access latency by 3.1% on average compared to Alloy Cache due to the lower cache miss ratio. Meanwhile, Ginkgo reduces access latency by an average of 13.5% compared to Unison Cache since it suffers from severe double-read problems. In the PR benchmark, although the miss ratio of Unison Cache is 10% lower than that of Ginkgo, large amounts of additional DRAM access increase its average memory access latency, which is 1.68% higher than that of Ginkgo. Furthermore, if we consider the latency overhead introduced by the computation of learned index and quantify it as 5 ns, Ginkgo reduces access latency by an average of 11.6% and 1.0% compared to Unison Cache and Alloy Cache, respectively. Consequently, Ginkgo reduces the miss ratio effectively while eliminating most double-read problems, resulting in lower memory access latency.

DISCUSSION

Dynamic Distribution-Aware Index Mechanism

Due to the significant overhead of distribution analysis and model construction, Ginkgo builds the learned index offline. However, such a static profiling-based approach is inflexible and cannot adapt to diverse benchmarks and datasets. Furthermore, the cache block distribution varies significantly during different execution phases of the application. In order to adapt to the diverse cache block distribution, it is essential for Ginkgo to support dynamic memory access tracking and online learned index construction, which are left for future work.

Enhance the Adaptability of Learned Index Model

Due to the limited computational and storage resources in hardware, Ginkgo can only build the learned index based on the PLF model. This model is suitable primarily for benchmarks with linear and regular distributions, such as BTree and PR. However, a large portion of benchmarks has nonlinear cache block distribution, like Deep Learning Recommendation Model and Silo. A flexible learned index model is required. There is still room for future optimization by enhancing the adaptability of the learned index model.

CONCLUSION

We introduced Ginkgo, a hardware-based tiered memory system, which adopts a distribution-aware index mechanism to reduce the cache conflicts and minimize the metadata overhead using the learned index. For benchmarks with regular cache block distributions, Ginkgo reduces cache conflicts effectively while eliminating additional DRAM access, achieving the lowest memory access latency compared to prior work.

ACKNOWLEDGMENTS

This work is mainly supported by the National Key Research and Development Program of China under Grant 2023YFB4502702.

REFERENCES

1. Memory modes for persistent memory systems, by S. S. Sood. (2019, Mar. 26). U.S. Patent 10,241,912 B2

[Online]. Available: https://patents.google.com/patent/US10241912B2/en
2. S. Sha, C. Li, Y. Luo, X. Wang, and Z. Wang, "vTMM: Tiered memory management for virtual machines," in *Proc. 8th Eur. Conf. Comput. Syst.*, 2023, pp. 283–297, doi: 10.1145/3552326.3587449.
3. M. K. Qureshi and G. H. Loh, "Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design," in *Proc. 45th Annu. IEEE/ACM Int. Symp. Microarchit.*, Vancouver, BC, Canada, 2012, pp. 235–246, doi: 10.1109/MICRO.2012.30.
4. D. Jevdjic, G. H. Loh, C. Kaynak, and B. Falsafi, "Unison cache: A scalable and effective die-stacked DRAM cache," in *Proc. 47th Annu. IEEE/ACM Int. Symp. Microarchit.*, Cambridge, U.K., 2014, pp. 25–37, doi: 10.1109/MICRO.2014.51.
5. S. P. Yang et al., "Overcoming the memory wall with CXL-enabled SSDs," in *Proc. USENIX Annu. Tech. Conf.*, Boston, MA, USA, 2023, pp. 601–617.
6. T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proc. Int. Conf. Manag. Data*, 2018, pp. 489–504, doi: 10.1145/3183713.3196909.
7. L. Liang, G. Yang, A. Hadian, L. A. Croquevielle, and T. Heinis, "SWIX: A memory-efficient sliding window learned index," *Proc. ACM Manage. Data*, vol. 2, no. 1, pp. 1–26, Feb. 2024, doi: 10.1145/3639296.
8. G. H. Loh and M. D. Hill, "Efficiently enabling conventional block sizes for very large die-stacked DRAM caches," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, Porto Alegre, Brazil, 2011, pp. 454–464, doi: 10.1145/2155620.2155673.

XIRAN YANG is a Ph.D. student at Peking University, Beijing, 100871, China. His research interests include tiered memory systems and memory disaggregation. Yang received a B.S. degree from Wuhan University. Contact him at yangxr@stu.pku.edu.cn.

YIFEI YU is a Ph.D. student at Huazhong University of Science and Technology, Wuhan, 430074, China. His research interests include tiered memory systems. Yu received a B.S. degree from Huazhong University of Science and Technology. Contact him at yfyu@hust.edu.cn.

CHUANDONG LI is Ph.D. student at Peking University, Beijing, 100871, China. His research interests include virtualization, persistent memory, and operating systems. Li received a B.S. degree from Peking University. Contact him at lichuand@pku.edu.cn.

JIANQIANG ZENG is a Ph.D. student at Peking University, Beijing, 100871, China. His research interests include locality theory, memory system optimization, and system virtualization. Zeng received a B.S. degree from Tsinghua University. Contact him at zengjq@stu.pku.edu.cn.

KE ZHOU is a professor in the Wuhan National Laboratory for Optoelectronics, Huazhong University of Science and Technology, Wuhan, 430074, China. His research interests include storage systems and file systems. Zhou received his Ph.D. degree from Huazhong University of Science and Technology. He is a Member of IEEE. Contact him at yzhke@hust.edu.cn.

DIYU ZHOU is an assistant professor at Peking University, Beijing, 100871, China. His research interests include operating systems and storage systems. Zhou received his Ph.D. degree from the University of California, Los Angeles. Contact him at diyu.zhou@pku.edu.cn.

XIAOLIN WANG is a professor at Peking University, 100871, China. His research interests include system virtualization and cloud computing. Wang received his Ph.D. degree from Peking University. Contact him at wxl@pku.edu.cn.

ZHENLIN WANG is a professor at Michigan Technological University, Houghton, MI, 49931, USA. His research interests include operating systems, and system virtualization. Wang received his Ph.D. degree from University of Massachusetts, Amherst. Contact him at zlw@mtu.edu.

YINGWEI LUO is a professor at Peking University, Beijing, 100871, China. His research interests include system virtualization, cloud computing, locality theory, and memory system optimization. Luo received his Ph.D. degree from Peking University. Contact him at lyw@pku.edu.cn.

Tiered Cache-Sharing Service for Virtual Machine Images Based on Memory Pool

Zhihao Zhang and Weinan Liu, *Xiamen University, Xiamen, 361005, China*
Zhenlong Song and Xinbiao Gan, *National University of Defense Technology, Changsha, 410000, China*
Yue Yu, *Peng Cheng Lab, Shenzhen, 518055, China*
Yiming Zhang, *Xiamen University, Xiamen, 410000, China*

In virtualization-based cloud computing, numerous redundant tiered memory pages exist across independent virtual machines (VMs) deployed on different physical hosts within the data center. In this article, we propose fexible memory (FlexMem), a preliminary architecture designed to eliminate memory redundancy for qcow2-based VMs. First, FlexMem introduces a virtual persistent memory (PMem) device that facilitates cross-host memory consolidation by leveraging remote direct memory access (RDMA) for fast remote read operations. Second, FlexMem employs flexible backing specifications that are memory-mapped to sparse and layered qcow2 images. By delivering images to VMs through FlexMem and enabling guest direct access, user programs within the guest can directly access identical images already cached in the remote host's memory. Our evaluation demonstrates that FlexMem significantly improves memory efficiency and the availability of VMs with reduced hardware costs, ultimately enhancing the competitiveness of cloud vendors.

Virtualization-based cloud computing, provided by leading cloud service providers,[1,2,3] has gained widespread adoption due to its ability to provide system-isolation security and near-native performance for virtual machines (VMs). Cloud data centers utilize VMs to efficiently harness hardware resources, enabling the execution of multiple workloads on a single physical machine while delivering high-availability cloud services. A significant challenge in cloud computing arises from the presence of redundant tiered memory pages across independent VMs hosted on different physical servers within a data center. This redundancy reduces memory efficiency in large-scale distributed VM deployments. For instance, the page cache, an inherent operating system (OS) mechanism designed to cache file content in main memory to minimize input–output (I/O) requests, tends to consume available free memory extensively. Unused pages in the page cache are retained indefinitely unless there is insufficient free memory for allocation. In large-scale VM deployments, each VM's page cache may independently load identical content from shared images without considering the existing cached content in the memory of other local or remote VMs (referred to as the *local* and *remote tiers*, respectively). This leads to substantial memory redundancy across VMs that reside on distributed physical hosts.

Remote direct memory access (RDMA) technology[4,5] provides a fast and efficient way to access remote memory, potentially avoiding CPU bottlenecks through its one-sided verbs that can bypass server-side CPUs. However, using RDMA to scale VMs and share image pages among them introduces significant challenges. Memory pages in VMs are typically managed by the guest OS kernel, which lacks awareness of

Digital Object Identifier 10.1109/MM.2025.3574139
Date of publication 2 June 2025; date of current version 24 December 2025.

whether specific image pages reside in local or remote memory. As a result, when a VM starts, the guest OS kernel cannot directly employ one-sided RDMA verbs to access required remote memory pages.

The advent of virtio-pmem devices[6] introduces a novel memory-like storage solution that, when combined with RDMA, offers an opportunity to share image pages among distributed VMs. In this article, we propose FlexMem, an architecture designed to eliminate tiered memory redundancy for VMs. At its core, FlexMem employs a virtual persistent memory (PMem) device (as opposed to physical PMem) that leverages RDMA's high-speed remote read capability to fetch remote memory efficiently. Furthermore, FlexMem incorporates flexible backing specifications that are memory-mapped to sparse and layered qcow2 images. By delivering images through FlexMem, tiered memory redundancy can be proactively addressed before it occurs. With guest direct access (DAX) enabled, when VMs access the same offset in FlexMem, one-sided RDMA verbs are used to retrieve corresponding image pages from the remote host's memory and map them to the guest's memory, eliminating the need to download the image from the remote registry server. Unmodified data within the image, including read-only data and code from libraries and executables, can thus be shared across all distributed VMs.

Our evaluation demonstrates that FlexMem significantly improves memory efficiency, with a notable ~35% reduction in memory usage for a single VM.

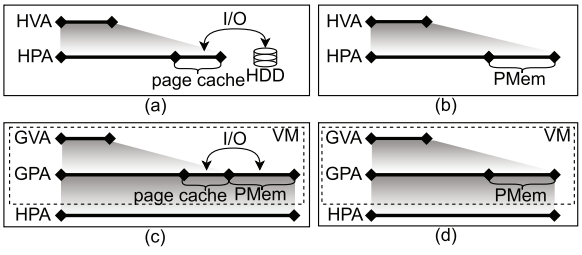


FIGURE 1. Evolutions of architectures after adding PMem, DAX, and virtualization. (a) Hard disk + page cache. Storage is provided by a hard disk, with the page cache mechanism caching data. (b) PMem + DAX. The hard disk is replaced with PMem, bypassing the page cache. (c) Virtual PMem + guest page cache. PMem is virtualized, and the page cache remains active, resulting in a virtual PMem with an unbypassed guest page cache. (d) Virtual PMem + guest DAX. With guest's DAX enabled, guest user programs directly access the host physical address (HPA) space through the virtual PMem. GPA: guest physical address. HDD: hard disk drive; HVA: host virtual address; GVA: guest virtual address.

Furthermore, there is a considerable ~48% increase in the maximum number of VMs that can be hosted on a single machine, indicating a substantial improvement in overall VM density across the data center.

BACKGROUND

RDMA in Virtualization Environments

In virtualization environments, RDMA can be used to accelerate VM migration[7] and container start-up.[4,5] Huang et al.[7] proposed leveraging RDMA's low software overhead and one-sided communication capabilities to enhance VM migration efficiency. They designed an efficient RDMA-based migration protocol that incorporates memory registration. This protocol significantly reduces migration overhead compared to traditional TCP/IP-based methods, offering a valuable contribution to VM management and data center operations. In Wie et al., KRCore[5] utilizes kernel-space RDMA and virtualizes pre-initialized RDMA connections to achieve rapid connection setup times, which is critical for container-based elastic applications. Its key innovations include the use of dynamically connected transport and hybrid connection pooling, which minimize control path overheads while maintaining high networking speeds. Similarly, in Wei et al.,[4] MITOSIS employs RDMA for remote forking to accelerate container start-up. Although these works have demonstrated the benefits of RDMA in virtualization environments, the use of RDMA in such environments remains limited. There is no existing work that combines virtio-pmem and RDMA to accelerate VM start-up and eliminate memory redundancy in cloud computing environments.

Page Cache and DAX and Virtual PMem

The page cache is an inherent OS mechanism that uses memory to cache file contents, thereby improving I/O performance. All I/O requests to a file are first directed to the cached content in memory. By maintaining consistency between the pages in the page cache and the corresponding file content on the storage device, this mechanism allows user programs to map their virtual address regions to the file's logical addresses, merging the file and memory into a unified user space, as shown in Figure 1(a). In contrast, with a DAX-enabled file system, virtual addresses can be mapped directly to physical addresses backed by byte-addressable devices, such as PMem.[8] This enables user programs to access data using memory instructions, bypassing the page cache entirely, as illustrated in Figure 1(b).

Virtio-pmem provides a virtual PMem device that the guest OS kernel can access via the virtio-pmem

driver, enabling the guest to bypass its own page cache and directly utilize the host's page cache. The primary benefits of virtio-pmem include reduced guest memory usage and improved memory management efficiency through host-side page caching. From the guest's perspective, the virtual PMem appears as a block device accessed through a designated portion of the guest physical address (GPA) space, as shown in Figure 1(c). By enabling guest DAX, the virtual PMem device can bypass the guest OS's page cache entirely, as shown in Figure 1(d). In this setup, the backing of the virtual PMem, i.e., the GPA space portion, is fully determined by the host.

MOTIVATION

Cloud tenants frequently use similar applications, prompting cloud vendors to provide standardized cloud computing services. Tenants often customize these standard images by incorporating their unique content and programs. Consequently, multiple tenants may utilize the same libraries and runtime environments from a base image, resulting in substantial redundancy in page cache usage.

The OS page cache is a built-in mechanism that uses memory to cache file contents, improving I/O performance. If a page in the cache is not accessed again, the OS does not actively evict it unless there is insufficient free memory. As a result, the page cache can consume a significant portion of available memory when sufficient memory is present. For example, as shown in Figure 2, which depicts the memory breakdown of a VM running Apache httpd without active connections, the page cache occupies 44% of the total 93 MB of used memory in the VM, whose image is provided via *virtio-blk*.

On a host machine, VMs created from the same base image often contain identical read-only data and initial states, such as server program code and libraries, within their respective page caches. However, each VM's page cache operates independently, caching data without accounting for other local or remote VMs. Consequently, multiple duplicates of server programs and libraries exist across host machines in the data center, as depicted in Figure 3(a). This leads to significant page cache redundancy and reduced efficiency during VM initialization.

The integration of virtual PMem (with guest DAX enabled) and RDMA technology offers an opportunity to fetch identical pages from the remote memory of other servers and share them among VMs. By leveraging these technologies and treating vCPUs as threads, VMs can share pages in the host physical address (HPA) space, as shown in Figure 3(b). This involves reading image data from the remote server's memory, configuring shared pages as the backing for virtual PMems across all VMs, and enabling guest DAX. As a result, all guest users within the VMs can directly access executables and libraries, which exist as a single copy on the host machine.

DESIGN

System Overview

We present the system overview of the FlexMem architecture in Figure 4. Within a single host machine, the FlexMem architecture comprises a daemon and multiple VMs equipped with FlexMem. The daemon is responsible for pre-analyzing image data and distributing the corresponding mappings to the VMs. Furthermore, if a requested image is not available on the local disk, the daemon acts as an intermediary between the local and remote machines.

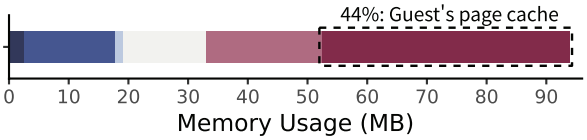


FIGURE 2. The memory breakdown of a VM using an image provided via virtio-blk shows that 44% of the total 93 MB of used memory is attributed to the page cache.

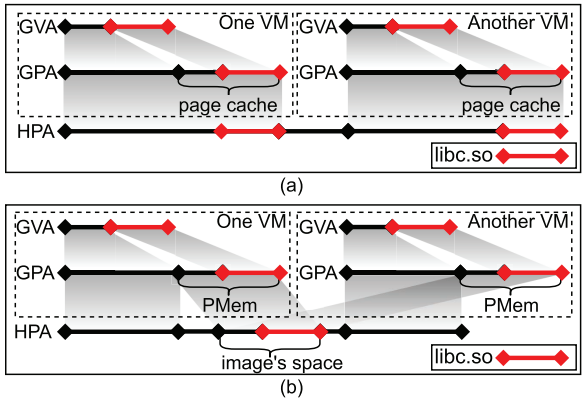


FIGURE 3. An example of duplicated *libc.so*. (a) Duplicated libc.so in HPA space. Independent VMs each load libc.so into their individual page caches, resulting in duplicated content on the host. (b) Merge libc.so by providing image in the form of virtual PMem. After enabling guest DAX, user programs on VMs access libc.so through their respective virtual PMems, all backed by the same image space on the host.

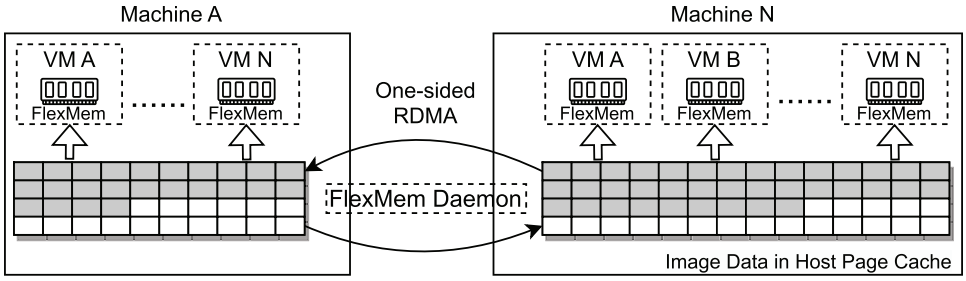


FIGURE 4. System overview of FlexMem architecture.

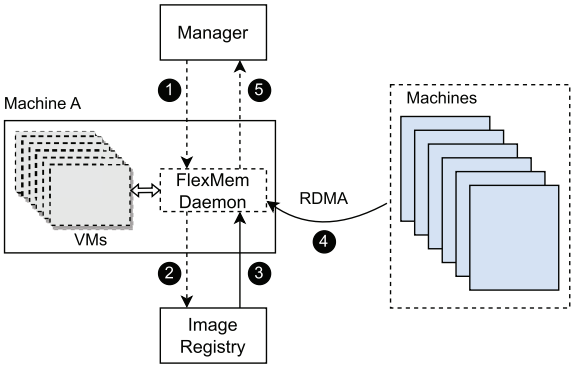


FIGURE 5. Control flow and data flow of FlexMem. The dashed line is control flow and the solid line is data flow.

For instance, in Figure 4, VM A on machine A is prepared for initialization once the FlexMem daemon receives an instantiation request from the user. Using the image information provided, the daemon first checks whether the requested image layers are available on machine A's local disk. If the requested layers are not present on the local disk, the FlexMem daemon connects to the remote machine N and uses one-sided RDMA READ to directly access the remote memory where the requested layers are stored, thereby bypassing all software overheads. Subsequently, FlexMem maps the image data pages to the virtual PMem of VM A, enabling DAX to the image data without downloading it from the remote registry server.

Control Flow and Data Flow

In this section, we illustrate both the control and data flows of FlexMem. As illustrated in Figure 5, ❶ the FlexMem daemon on each machine is initially responsible for processing initialization requests from the manager to set up VMs. ❷ If the requested image is not available on the local disk and no other VMs on the same machine have cached the required image pages, the daemon sends a request to the remote image registry server to download the image. ❸ The corresponding image layers are then pulled and stored on the local disk. Once the image is retrieved, the daemon loads the image data into the host's page cache and maps it to the virtual PMem of the target VM. ❹ If the manager provides metadata indicating that another machine has already loaded the same image pages into its page cache, FlexMem leverages one-sided RDMA READ operations to directly fetch the required image pages from the remote machine's physical memory. This eliminates the need to download the image from the remote registry, significantly reducing latency. The fetched data are then mapped to the initializing VM's virtual PMem, enabling DAX to the image data. ❺ After the VM initialization is complete, FlexMem communicates the following details to the manager: image metadata (including identifiers and layers) and the host's physical memory address and length as well as the offset and length of the corresponding image file.

When a VM is closed and cleaned up, FlexMem retains the image pages in the host's page cache until either a predefined time threshold (typically 3 min) is reached or the pages are accessed by other VMs on the same host machine. If no other VMs—whether local or remote—access the image pages during this period, FlexMem sends a cleaning request to the manager. Once the manager confirms the cleaning request, the image pages are released from the host's page cache. This approach ensures efficient memory utilization while minimizing unnecessary data eviction.

Image Mapping of FlexMem

FlexMem provides a flexible backing specification for delivering formatted images to VMs. To specify the appropriate backing, it is necessary to know which layered image file and what offset of each block in the image's space maps to. Algorithm 1 facilitates this determination process by systematically iterating through the blocks within the image's space. For each block, the algorithm examines the metadata of the

image to identify the layer and its corresponding offset where the block is stored. The time complexity of Algorithm 1 is $O(mn)$, where m represents the length of the image's space, and n denotes the number of layers. This algorithm is asymptotically optimal as each block needs to be mapped, and for each block, all layers need to be sequentially checked to determine whether a layer stores the block.

Algorithm 1. The algorithm for the mappings of image's space.

Input: $\langle \text{layer}_1, \text{layer}_2, \ldots, \text{layer}_n \rangle$ as layers of sparse images
Output: The mappings from blocks to image files

```
map = ∅
Using tmpfile to create a raw zeroed_layer whose length is equal to the length of layer1's space.
layers = ⟨layer1, layer2, ..., layern, zeroed_layer⟩
  for each block in layer1's space do
    for each layer in layers do
      if the layer stores the block then
        Analyzing the metadata, get the offset of the image file where the block is located.
        mpa = map ∪ {⟨block,⟨image, offset⟩⟩}
        break
      end if
    end for
  end for
  return map
```

It is essential to introduce a dedicated FlexMem daemon before initializing any VM. Without the daemon, when a layer is either compressed or encrypted, each VM would need to independently decode the data and maintain its own set of pages. This process reintroduces image redundancy within the host, leading to inefficient memory usage. Additionally, the daemon enhances overall system performance by ensuring that Algorithm 1 is executed only once per image, rather than being run separately for each image within every VM. By doing so, the daemon efficiently distributes the resulting mappings of the image's space to the VMs. Consequently, each VM requires only $O(m)$ time to restore the mappings for FlexMem, further optimizing resource utilization and reducing overhead.

IMPLEMENTATION

We have developed a prototype of FlexMem based on QEMU,[9] designed to operate on a 100-Gbps RDMA network. This prototype incorporates a new memory back end for virtio-pmem,[6] along with a daemon deployed on each host machine. The daemon is responsible for managing layered images and fetching remote memory from other servers. Specifically, the FlexMem daemon executes Algorithm 1 to analyze and retrieve the address space of qcow2 images. The memory back end then fetches the mappings to these layered images from the daemon and maps them into the GPA space. To optimize performance, the FlexMem daemon reduces the overhead associated with creating RDMA connections (commonly referred to as reliable connected queue pairs [RCQPs]) on the critical path of remote memory fetching. Additionally, it avoids caching the RCQPs connected to all servers, thereby conserving memory resources. Furthermore, FlexMem leverages dynamic connected transport (DCT) and hybrid connection pooling[5,10] to establish a connectionless, low-overhead RDMA communication mechanism on the critical path of data flow, enhancing efficiency and scalability.

DCT supports an unlimited cluster size, which is critical for large-scale distributed systems. This capability is enabled by a single queue pair object that can dynamically manage multiple connections. Additionally, DCT maintains the functionality of reliable connected transport, ensuring dependable data transfer through mechanisms like acknowledgment and retransmission. Although DCT has known performance issues related to extra reconnection messages, this overhead does not significantly impact image transfers, as the time required for data transfer dominates the process. Given that FlexMem's workload pattern primarily involves large transfers, such as reading remote pages in 4-KB granularity, we empirically observed no significant influence from the reconnection issue. Furthermore, DCT is supported by the latest RDMA NICs, such as the Nvidia ConnectX-5,[10] underscoring its relevance in modern high-performance networking environments.

To protect pages from write access, the `userfaultfd`[11] mechanism is developed to allow a user program to write-protect a virtual page. When a thread attempts to write to a write-protected page, it is blocked, and another thread waiting for a page fault is awakened to handle the fault. In our scenario, the `userfaultfd` cannot protect named pages, which are commonly used as the backing for virtual PMem. Therefore, this limitation renders `userfaultfd` unsuitable for our use case. Instead, we can use the error code `EFAULT` returned by `KVM_RUN` when a write operation occurs to a page that has a non-writable mapping, but is set as writable using `KVM_SET_USER_MEMORY_REGION` in GPA space. By intentionally setting conflicting write permissions for a page, we can halt a vCPU if it attempts to write to that page, effectively protecting

it from unauthorized write operations. This approach provides an alternative method to enforce write protection on pages without relying on `userfaultfd`.

DISCUSSION

PMem Overhead

Due to the total ~1.65% overhead of virtual PMem mentioned previously, employing a large-capacity image results in significant overhead for numerous small VM instances. To mitigate this issue, one potential solution is to eliminate the requirement for a `struct page` for the PMem device. As PMem's pages cannot be swapped out or moved, it may be unnecessary to track information about their references.

FlexMem Overhead and Benefit

FlexMem introduces its own overhead, primarily stemming from the establishment and revocation of mappings between GPA and HPA. When a VM accesses a page for the first time, either for reading or writing, the reverse mapping must be updated. This operation is mutually exclusive and utilizes a red-black tree to record virtual pages that reference the host's physical page, incurring logarithmic overhead. However, if thousands of VMs are already referencing a specific host's physical page, the cost of modifying the reverse mapping becomes nearly constant due to the efficient structure of the red-black tree. Furthermore, as the start-up processes of multiple VMs are not entirely simultaneous, FlexMem avoids significant lock contention, ensuring scalability and minimizing performance degradation.

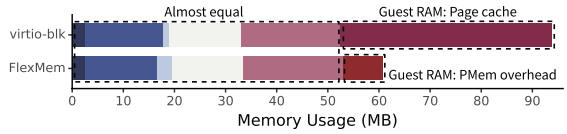


FIGURE 6. VM memory breakdown of two types VMs. RAM: random-access memory.

Memory Usage (KB)	virtio-blk	FlexMem
QEMU	2630	2629
QEMU's stack and head	15592	14316
The image's space	0	190
Metadata in host kernel	1339	2874
Guest RAM: Kernel	14220	14220
Guest RAM: Metadata + Anonymous	19924	19976
Guest RAM: Page cache	42424	0
Guest RAM: PMem overhead	0	8000

FIGURE 7. VM memory usage of two types VMs.

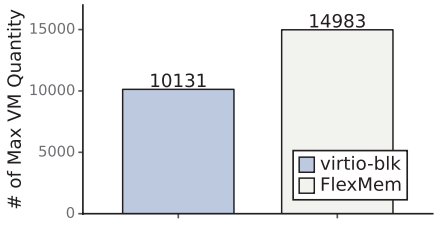


FIGURE 8. VM quantity improvement.

EVALUATION

To demonstrate the functionality of FlexMem, we selected the official WordPress image,[12] which is ~500 MB and a trimmed 14-MB Linux-6.1 kernel as the guest OS. In the subsequent evaluations, we perform a performance comparison between two distinct types of VMs, both configured with 96 MB of random-access memory (RAM) (see Figure 6). One type employs the standard virtio-blk, while the other integrates FlexMem. The evaluation setup involves three host machines, each equipped with dual Intel Xeon Silver 4316 processors, featuring 20 cores and 1 TB of RAM. These machines are interconnected via a 100-Gbps RDMA network. Among the three machines, one serves as the manager, while the remaining two host all the VMs under evaluation.

VM Instance Memory Breakdown

The memory breakdown of the VM's instance, as depicted in Figure 7, is obtained by analyzing `pmap` of a QEMU process and `/proc/meminfo` after 1000 VMs completed booting, showcasing the memory usage of one VM for both the guest and host. It is worth noting that, the *Guest RAM: Page cache* in the FlexMem VM (60 MB) leads to an ~35% decrease compared to the *virtio-blk* VM (93 MB). Although FlexMem can eliminate page cache redundancy, the *Guest RAM: PMem overhead* increases due to the expansion of the GPA space for PMem. Each additional 4-KB page in the GPA space contributes to an additional 64 B struct page in the guest kernel and a 4 B page table entry in the host kernel, resulting in an overhead for virtio-pmem of ~1.56% for the guest and ~0.09% for the host. The memory usage of the other components remains roughly unchanged.

VM Instance Density

As illustrated in Figure 8, compared to *virtio-blk* VMs, the maximum quantity of FlexMem VMs is ~48% higher on the same 1-TB RAM machine, indicating that FlexMem improves the VM's density. The quantity of these two types of VM instances is roughly inversely proportional to the memory usage illustrated in Figure 7.

VM Instance Start-Up Time

We measure the average time required for 64 parallel VMs to complete both cold and warm start-ups. A cold start-up refers to the process where all VMs boot from scratch, with no preexisting VMs available. This involves downloading the image from the registry via HTTP, loading the image into the guest's memory, and initializing the guest OS. In contrast, a warm start-up measures the time taken for all VMs to boot using a preexisting VM. For FlexMem, the warm-up process involves preparing the VMs on a remote machine and then utilizing RDMA to fetch the image pages already loaded into the remote machine's memory. For *virtio-blk* VMs, we pre-download the image from the registry.

As depicted in Figure 9, for both types of VMs, the preexisting VMs accelerate the VM's start-up by up to ~48.5%. The FlexMem VMs exhibit an average start-up time that is ~37.4% faster than that of the *virtio-blk* VMs. This improvement can be attributed to FlexMem VMs eliminating the need to load image content into the guest's memory, allowing them to directly access the image content from remote memory. As the number of existing VMs increases, *virtio-blk* VMs consume memory more rapidly, resulting in prolonged start-up times for subsequent VMs. In contrast, our FlexMem method not only enhances I/O performance but also significantly improves system availability at the same VM density. Moreover, when the number of VMs is between 64 and 1024, the difference in VM start-up time decreased because the overhead of FlexMem becomes more apparent.

FlexMem and Kernel Same-Page Merging

We explore memory deduplication technologies using kernel same-page merging (KSM)[13] to identify redundant content between two VMs, as shown in Figure 10. In the experiment, KSM is activated at 0 s. At 10 s, the first VM is started, followed by a second VM at 20 s. The scanning period is configured to the default value of 20 ms, and memory usage stabilizes after ~200 s. The memory usage of FlexMem without KSM and *virtio-blk* with KSM converges to nearly the same level, demonstrating that KSM can effectively detect and eliminate the redundant guest pages removed by FlexMem. Furthermore, the memory usage of FlexMem with KSM is 27-MB lower than that of FlexMem without KSM, indicating the presence of additional redundant content between the two VMs. Notably, the majority of this redundancy corresponds to the 14-MB Linux kernel.

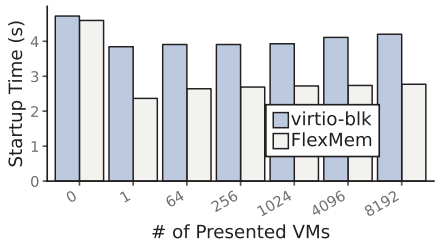


FIGURE 9. VM start-up time speedup.

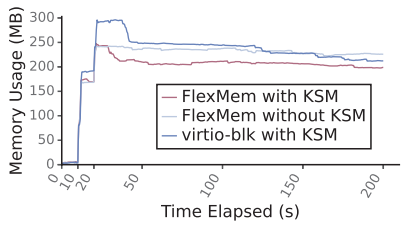


FIGURE 10. Memory usage changes over time with or without KSM.

CONCLUSION

In this article, we analyzed the memory redundancy prevalent in virtualization-based cloud platforms and proposed a novel architecture, FlexMem, which is designed to eliminate this redundancy. By leveraging RDMA-based FlexMem and enabling guest DAX, the system effectively eliminates guest-level redundancy. Our evaluation shows that, compared to traditional architectures, FlexMem significantly improves VM start-up time by up to ~37.4%, achieves an ~35% reduction in memory usage, and an ~48% increase in the maximum number of VMs. The FlexMem architecture can be seamlessly integrated into cloud data centers in conjunction with remote image services, enhancing VM memory efficiency, improving I/O performance, and boosting VM availability. These improvements not only reduce hardware costs but also strengthen the overall competitiveness of cloud vendors.

ACKNOWLEDGMENTS

We sincerely thank the anonymous reviewers for their valuable comments and suggestions. The work is supported by the National Key Research and Development Program of China (Grant 2022YFB4500302) and the National Natural Science Foundation of China (Grant 62302514). Yiming Zhang and Xinbiao Gan are the corresponding authors.

REFERENCES

1. "AWS Lambda features." Amazon Web Services (AWS). Accessed: Nov. 10, 2024. [Online]. Available: https://aws.amazon.com/lambda/features/
2. "Azure virtual machines." Microsoft. Accessed: Nov. 10, 2024. [Online]. Available: https://azure.microsoft.com/en-us/products/virtual-machines/
3. "Google Cloud platform: Compute engine." Google. Accessed: Nov. 10, 2024. [Online]. Available: https://cloud.google.com/products/compute
4. X. Wei et al., "No provisioned concurrency: Fast RDMA-codesigned remote fork for serverless computing," in *Proc. 17th USENIX Symp. Operating Syst. Des. Implementation (OSDI)*, Boston, MA, USA: USENIX Association, Jul. 2023, pp. 497–517. [Online]. Available: https://www.usenix.org/conference/osdi23/presentation/wei-rdma
5. X. Wei, F. Lu, R. Chen, and H. Chen, "KRCORE: A microsecond-scale RDMA control plane for elastic computing," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Carlsbad, CA, USA: USENIX Association, Jul. 2022, pp. 121–136. [Online]. Available: https://www.usenix.org/conference/atc22/presentation/wei
6. "virtio pmem." QEMU. [Online]. Available: https://www.qemu.org/docs/master/system/devices/virtio-pmem.html
7. W. Huang, Q. Gao, J. Liu, and D. K. Panda, "High performance virtual machine migration with RDMA over modern interconnects," in *Proc. IEEE Int. Conf. Cluster Comput.*, 2007, pp. 11–20, doi: 10.1109/CLUSTR.2007.4629212.
8. A. Baldassin, J. Barreto, D. Castro, and P. Romano, "Persistent memory: A survey of programming support and implementations," *ACM Comput. Surv.*, vol. 54, no. 7, pp. 1–37, 2021, doi: 10.1145/3465402.
9. "QEMU." QEMU. Accessed: Nov. 10, 2024. [Online]. Available: https://www.qemu.org/
10. "NVIDIA advanced transport," *NVIDIA*, 2024. Accessed: Nov. 10, 2024. [Online]. Available: https://docs.nvidia.com/networking/display/mlnxofedv23070512/advanced+transport
11. "Userfaultfd," *Linux Kernel Docs*, 2024. [Online]. Available: https://www.kernel.org/doc/html/next/admin-guide/mm/userfaultfd.html
12. "Wordpress – Official image." Docker Hub. [Online]. Available: https://hub.docker.com/_/wordpress
13. "Kernel samepage merging," *Linux Kernel Docs*, 2024. [Online]. Available: https://docs.kernel.org/admin-guide/mm/ksm.html

ZHIHAO ZHANG is a Ph.D. student at the School of Informatics, Xiamen University, Xiamen, 361005, China. His research interests include virtualization, cloud stroage, and computing. Zhang received his M.S. degree in computer science from the Xiamen University. Contact him at zhihaoz@stu.xmu.edu.cn.

WEINAN LIU is a Ph.D. student at the School of Informatics, Xiamen University, Xiamen, 361005, China, and Peng Cheng Lab, Shenzhen, 410000, China. His research interests include virtualization, artificial intelligence, and cloud computing. Liu received his B.S. degree in computer science from the Henan University. Contact him at wnliu@stu.xmu.edu.cn.

ZHENLONG SONG is with National University of Defense Technology, Changsha, 410000, China. His research interests include cloud storage and high performance computing. Song received his Ph.D. degree in computer science from the National University of Defense Technology. Contact him at songzhl@sina.com.

XINBIAO GAN is an associate professor with the National University of Defense Technology, Changsha, 410000, China. His research interests include high performance computing, artificial intelligence, and big data. Gan received his Ph.D. degree in computer science from the National University of Defense Technology. Contact him at xinbiaogan@nudt.edu.cn.

YUE YU is a professor with Peng Cheng Lab, Shenzhen, 518055, China. His research interests include software engineering, distributed and cloud computing, and artificial intelligence. Yu received his Ph.D. degree in software engineering from the National University of Defense Technology. Contact him at yuy@pcl.ac.cn.

YIMING ZHANG is with the School of Informatics, Xiamen University, Xiamen, China; Peng Cheng Lab, Shenzhen, China. His research interests include operating, networking, storage, and artificial intelligence systems. Zhang received his Ph.D. degree in computer science from the National University of Defense Technology. He is a Senior Member of IEEE. Contact him at sdiris@gmail.com.

Improving SQL Join Algorithms for Distributed Systems: A Case Study of Compute Express Link-Based Multihost Shared Memory

JaeYung Jun, HyunWoong Ahn, Joohee Lee, Jungmin Choi, Byungil Koh, Donguk Moon, and Hoshik Kim, *SK hynix Inc., Icheon, 17336, South Korea*

The advent of compute express link (CXL) has introduced the possibility of multihost shared memory architectures. Despite this advancement, there has been limited exploration of shared memory at the application layer. Traditional distributed systems typically partition data across multiple servers, enabling independent processing. However, cross-partition operations, such as joins, require data repartitioning, leading to significant communication overhead. To address this challenge, we propose merge hash join (MHJ), a novel structured query language join algorithm that leverages shared memory to eliminate the need for repartitioning. By storing the joining table in shared memory and making it directly accessible to all servers, MHJ significantly reduces communication overhead. To validate our approach, we implemented MHJ and the necessary shared memory functionalities on a CXL-based shared memory prototype. Extensive evaluations using the industry-standard TPC-DS benchmark demonstrate that MHJ achieves up to 1.5× performance improvement compared to conventional join algorithms.

For more than a decade, the concept of distributed shared memory, which offers multiple processes a unified virtual address space, has been actively researched in distributed systems. However, achieving consistency and coherency within these systems incurs high costs due to the data movement across locally attached memories in the distributed servers.[10] This has driven the majority of software frameworks to employ the shared-nothing memory model,[5,14] where data sharing among processes relies on explicit network communication.

In recent years, a new interconnect protocol, compute express link (CXL),[4,12] has emerged, providing a novel approach for physically sharing memory across multiple servers. Originally, CXL memory was designed to enhance per server memory capacity or bandwidth. It has since evolved to facilitate the composition of memory resource pooling from multiple servers. Although various studies have explored memory pool utilization within data centers,[2,6,7,8] no previous work has effectively harnessed the data-sharing capabilities offered by the memory pooling.

In this article, we present a case study exploring the use of data-sharing capability in a distributed system. Specifically, we focus on Spark SQL, a framework for executing data analytics tasks described in standard structured query language (SQL) on distributed systems. We specifically concentrate on optimizing equi-join algorithms within a shared memory model, a complex operation incurring data movement overhead in traditional distributed systems. An equi-join combines two tables based on matching entries for specified key columns. This operation typically requires a repartitioning stage to create smaller disjoint tables, which are, in turn, processed in parallel. However, this

Digital Object Identifier 10.1109/MM.2025.3574357
Date of publication 30 May 2025; date of current version 24 December 2025.

stage involves data shuffle operation, a form of all-to-all communication, causing significant network and file system overhead. Despite such overheads, conventional equi-join algorithms remain the preferred option under the shared-nothing memory model.

The advent of shared memory in distributed systems, facilitated by the new CXL interconnect, motivates us to reassess the efficiency of the existing equi-join algorithms under the shared memory model. Those algorithms involve the repartition stage to reduce the search space from the entire table to small disjoint tables, which each executor then processes on its local memory. Yet, we need to ask: What if all executors could directly access the entire table via shared memory? Such a scenario would eliminate the need for reducing the search space, rendering the repartition phase unnecessary.

Inspired by this proposition, we propose a novel equi-join algorithm tailored for distributed systems with shared memory. This algorithm not only minimizes data movement but also preserves data parallelism for concurrent execution without any lock contention. The primary contributions of this article can be summarized as follows:

- We enhance the performance of SQL join operations by designing a new join algorithm optimized for the shared memory model.
- We present our approach to adapting system software for shared memory deployment with minimal implementation effort. We believe this modified system software proves beneficial before traditional system software evolves to formally support shared memory.
- We have developed and tested a CXL-based multihost shared memory prototype and evaluated the effectiveness of our new join algorithm in the distributed system using this prototype.

BACKGROUND

Compute Express Link

CXL is an open industry-standard interconnect between processors, memory, and accelerators. By supporting memory semantics over the peripheral component interconnect express (PCIe) physical layer, CXL memory expanders can be integrated into a server like conventional dynamic RAM (DRAM). However, compared to DRAM, CXL memory exhibits unique characteristics stemming from the PCIe physical layer: 1) Its bandwidth is constrained by the PCIe interface and 2) its serial link-based communication results in higher latency compared to DRAM.

While the initial CXL specification (CXL 1.0/1.1) emphasizes a point-to-point connection between CPUs and CXL memory expanders, CXL 2.0 brings on a novel concept of memory pooling. CXL 3.0 and subsequent versions support memory sharing, enabling multiple hosts to share CXL memory coherently. For the coherency between the host caches, a snoop filter and back-invalidation are employed.

Unfortunately, implementing memory pooling is not yet possible because only CXL 2.0-compatible processors and memory expanders are currently commercially available. Thus, we prototyped a CXL-based memory pooling system supporting shared memory by extending CXL 2.0 devices.

Spark SQL and Join Algorithms

Apache Spark is a popular open source, unified analytics engine for large-scale data processing across clusters of computers. It provides built-in modules to handle a wide range of workloads for big data analytics. Spark SQL, a module within Apache Spark, supports querying structured and semistructured data through SQL.

Among various SQL operations supported in Spark SQL, join operations can particularly become a performance bottleneck. Apache Spark partitions data across multiple executors within a cluster and broadcasts operations to these executors. Each executor then processes a partition of the data independently and in parallel. Join operations introduce cross-partition dependencies,[9] which means processing a partition is associative to other partitions. In particular, an item in one partition of a dataset might need to be joined with an item in any partition of the other dataset. To handle this, Spark SQL employs a preprocessing stage involving data shuffling between executors. While this stage resolves cross-partition dependencies and allows join operations to be processed in parallel, it introduces significant network and disk input/output (I/O) overhead, potentially degrading the overall query performance.

Spark SQL uses three distinct join algorithms: shuffled hash join (SHJ), broadcast hash join (BHJ), and sort merge join (SMJ), each differentiated by their preprocessing and join methods. The operation flow of SHJ and BHJ is illustrated in Figure 1. SHJ and BHJ employ a hash-based join method, which involves the verification of equivalent entries by looking up a hash map.[3,11,a]

[a]A hash map, also known as a hash table, is a data structure that maps keys to values storing (key, value) pairs in an index obtained by applying a hash function to the key.

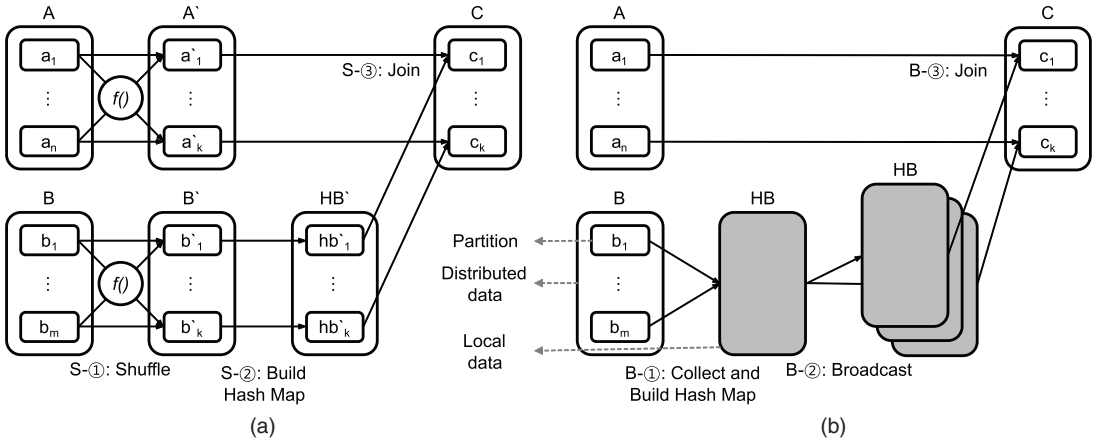


FIGURE 1. Operation flows of joining tables A and B into table C according to join algorithm. (a) SHJ algorithm and (b) BHJ algorithm.

In the preprocessing stage, SHJ repartitions two joining tables, *A* and *B*, into *A′* and *B′* using the same hash function, *f()* to remove cross-partition dependency (S-①). For example, the following query collects the first column from rows in table A where the second columns of table A and B have the same value; "*SELECT A.col1 FROM A JOIN B ON A.col2 == B.col2*." The hash function is applied to key columns, which are A.col2 and B.col2 in the example. After the stage, it is guaranteed that *i*th partition of table *A′* is disjoint with all partitions of table *B′* except for *i*th partition. In other words, between different partitions, the joining condition, e.g., "*A.col2 == B.col2*," is not satisfied. As a result, join operations can be performed in a partition-wise manner. Each executors creates a hash map using the *i*th partition of table *B′* (S-②), and then uses this map to perform the join with the corresponding partition from table *A′* (S-③).

In contrast, BHJ gathers the distributed partitions of a smaller table, builds a hash map from the table (B-①), and then broadcasts the hash map to all the executors within the cluster (B-②).[3] As a result, each executor can search for matching entries using its own local copy of the hash map (B-③). Sort merge join (SMJ) employs a strategy similar to SHJ in its preprocessing stage by repartitioning the data. The actual join operation, however, involves comparing and merging entries from two sorted partitions in an iterative manner.

Each of these algorithms incurs data movement overhead from repartitioning and broadcasting, potentially increasing the query response time. Furthermore, BHJ entails additional memory overhead because it duplicates the hash map as many as executors in the cluster. This overhead imposes a constraint on its use: By default, Spark enables BHJ only if the size of the table is under 10 MB.[1]

A NEW JOIN ALGORITHM: MERGE HASH JOIN

In this section, we introduce a new join algorithm leveraging shared memory and address how the algorithm can relax the data movement overhead of existing algorithms. We observed that the broadcasted hash map in BHJ can be shared among executors by placing it in the shared memory, as shown in Figure 2(a). Based on this observation, we modified the BHJ implementation to place the smaller table in the shared memory, thereby eliminating the data movement overhead of existing join algorithms.

We further improved the algorithm by building hash maps of the shared table in parallel with multiple executors. Original BHJ implementation builds a hash map for a small table with a single executor. Due to the memory capacity overhead, Spark SQL limits the execution of BHJ to table sizes of 10 MB or less. Because of the small size, original Spark does not parallelize building a hash map of BHJ. However, as merge hash join (MHJ) aims to support a much larger table, we needed to parallelize building hash maps. In MHJ, hash maps are built in parallel as follows, considering concurrency and shared memory. Executors locally build hash maps from partitions of the distributed table and store them in the shared memory (M-②). Then, the hash maps are merged and treated as a single hash map for lookup. This approach incurs no lock contention since each executor independently builds its own hash maps.

However, looking up the generated hash maps becomes more complex than before. To ensure fast

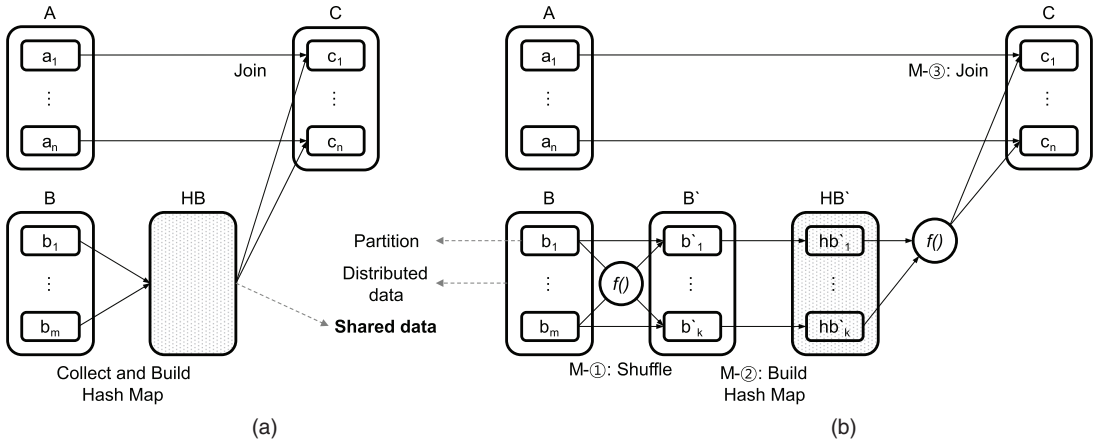


FIGURE 2. Operation flows of joining tables A and B into table C in the proposed algorithm using shared memory. (a) BHJ algorithm with zero-copy and (b) merge hash join algorithm.

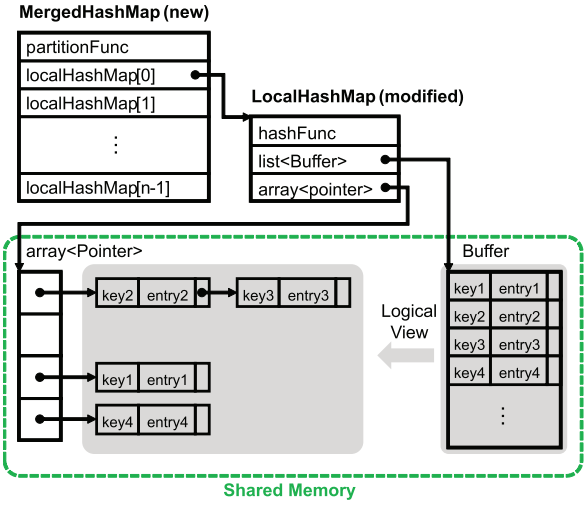


FIGURE 3. Data structure of the hash map used in our algorithm.

lookups, MHJ repartitions the shared table (M-①) using a hash function, *f()*, before building its hash maps (M-②), similar to SHJ. What is different from SHJ is that the opposite table is not repartitioned for MHJ. This is possible since MHJ's locally generated hash maps are visible to all executors through the shared memory. MHJ handles the lookup in two steps: selection and search (M-③). In the selection step, MHJ selects the target hash map by using the same hash function, *f()*, for repartitioning and a key of the opposite table as input to the hash function. Then, the selected hash map is searched with the key for actual join operations. In summary, although introducing repartitioning may incur some data movement overhead, it helps MHJ build hash maps in parallel, avoiding being bottlenecked by the hash map build process for a large table.

Due to the mechanism described above, MHJ offers several advantages over SMJ and SHJ. First, data movement is reduced since MHJ repartitions only a joining table, whereas SMJ and SHJ require repartitioning both tables. Second, MHJ demonstrates greater robustness against data skew, which could incur partition size imbalances in case of SMJ and SHJ.

IMPLEMENTATION

Algorithm Implementation in Spark SQL

Our MHJ is implemented by extending existing BHJ in Spark SQL. The hash map build stage was redesigned for parallel processing, and the created local hash maps are collected by the master executor in the form of a primitive array. A merged hash map is created from the local hash maps and made visible as a single hash map for lookup, as presented in Figure 3. The merged hash map is then broadcasted to all of the executors for join operation. Since the interface of the merged hash map is kept the same as the original hash map interface, hash map lookup occurs in the same way as BHJ.

The above process incurs data movement: 1) when the master executor collects local hash maps from executors and 2) when the master broadcasts the merged hash table back to executors. In Spark, such data movement requires object serialization and deserialization when writing to and reading from a file. To reduce serialization and deserialization overheads, we designed MHJ to move the address of data while keeping the content of data in the shared memory.

Specifically, serialization and deserialization functions were overridden to write the address of an object to a file and construct the object from the address, respectively.

Figure 3 shows the data structures involved with MHJ. A local hash map (*LocalHashMap*) has three major variables, *hashFunc*, and *list <buffer>*, *array <pointer>*. During data insertion, data is appended to the buffer, and the data address in the buffer is registered to the array entry, which is indexed with the specified hash function, *hashFunc*. And if the buffer is full, a new buffer is allocated and inserted to the list of buffers. The list and the array are allocated from the shared memory. Compared to BHJ, MHJ incurs I/O overheads only for moving *MergedHashMap* and *LocalHashMap* between executors, while the actual data objects remain in the shared memory and are accessed by the executors via load/store instructions.

Shared Memory Management

This section introduces how to manage shared memory. At the user level, shared memory is managed with Java's off-heap mechanism and memory management library, memkind. The memkind library manages the virtual address space mapped to the shared memory. Our Java module for shared off-heap management was implemented with Java native interface to use the memkind library in the Java environment. By leveraging Spark's original join implementation, which stores hash maps in byte array format, we could simply allocate the hash map data structure in the shared off-heap, as shown in Figure 3, by using the module.

For the operating system (OS)-level management, we employ static partitioning and linear mapping of the address space with direct access (DAX) to resolve shared memory resource contention and establish the common address mapping among executors running on multiple OS instances. When an OS manages a memory device as a DAX device, it directly and linearly maps the corresponding memory space to a process through file-mapped memory syscall, instead of using the space as system memory. Using the DAX device mechanism, we mapped the entire DAX device to the same virtual address space of each Spark executor so that all of the executors could directly access any data on the virtual address space. To prevent resource contention, we statically and evenly partitioned the shared memory space, reserving only one partition per executor for memory allocation. And only the executor corresponding to a partition has the allocation right.

Software Coherency Mechanism

Currently, hardware cache coherency mechanisms are maintained among processors within a single host server. CXL 3.0 introduces a new hardware cache coherency mechanism for shared memory across servers, but compatible hardware is unavailable in the market. To address this gap, we implemented cache coherency across servers using cache flush instructions, circumventing hardware limitations with software-based cache coherency.

Fortunately, our algorithm's coarse-grained sharing and explicit data dependency enable a straightforward and efficient implementation of the software cache coherency mechanism. During hash map construction, there is no sharing of hash maps between executors as each executor independently constructs a local hash map. In the subsequent joining operation, executors perform only read operations to the hash maps. Each hash map has a unique writer and multiple readers, with distinct write and read phases. As such, performing a cache flush just before the joining operation is sufficient to maintain the system's cache coherency.

To minimize the overhead of cache flushing, we employed an instruction that flushes the entire cache. Flushing the entire cache takes around 50 ms. However, we expect that this impact is negligible due to our algorithm's coarse-grained synchronization. In addition, this overhead will be reduced by hardware cache coherency support in CXL 3.0.

Shared Memory Prototype

Conceptually, the sharing feature is designed by mapping memory spaces of multiple CXL memory devices to the same physical memory, as shown in Figure 4(a). Host-managed device memory (HDM) is an addressable memory space accessible via memory semantic protocol (.mem). In our design, the HDM of each CXL memory device is mapped to a 512-GB memory pool composed of four 128-GB DRAM dual inline memory modules (DIMMs) through the internal bus. Each host enumerates and maps the HDM to host system address space. For example, a 512-GB HDM is mapped to the system address *X* by *Host*1. The OS manages the mapped HDM as a DAX device by default, and we use the DAX device.

When a processor core accesses address *A*, the request is routed to CXL RP if *A* belongs to the host physical address space mapped to the HDM, i.e., $X \leq A < + 512$ GB. The request is then sent to the connected CXL memory device, where address *A* is converted to internal bus address, i.e., $A - X$, to access the memory

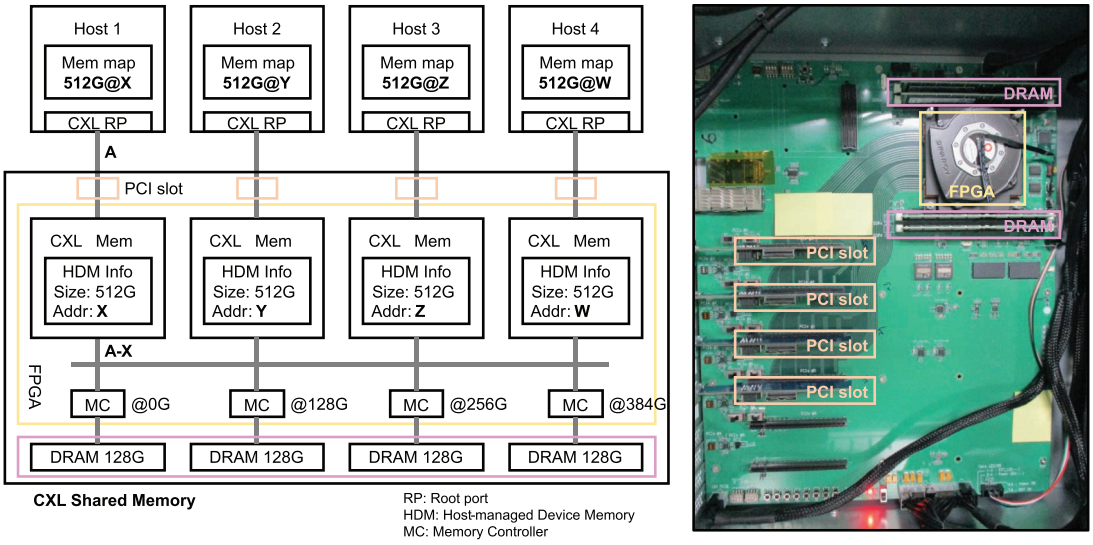


FIGURE 4. (a) Hardware block diagram of our CXL-based shared memory prototype. (b) Picture of the prototype.

TABLE 1. Latency and bandwidth of CXL memory.

	Latency (ns)	Bandwidth (GB/s)	
		ALL Reads	R:W = 1:1
cCXL local memory*	250	13	14
ASIC-implemented CXL shared memory†	365		
FPGA-implemented CXL shared memory prototype*	620	4.6	4.8

*These are measured with the memory latency checker tool.

†The latency is estimated by adding the latency of CXL local memory to the latency difference between a two socket and 32 socket pool,[2] i.e., 250 + (270 – 155) = 365 ns. The bandwidth is ideally not degraded, assuming a processor has enough capabilities buffering memory requests for the increased latency.

pool. Finally, the internal bus request with address $A - X$ is routed to one of the four DRAM DIMMs in the memory pool according to the mapping information of DRAM in Figure 4(a).

Figure 4(b) shows our field-programmable gate array (FPGA)-based CXL shared memory prototype, which has limited performance due to FPGA implementation. We expect enhanced performance with ASIC implementation. The actual number for the latency and bandwidth is shown in Table 1.

Our prototype builds on recently proposed CXL-based pooled memory implementations.[2,7] Both designs provide connection to memory pool via multiple ports. Unlike the previous designs, our implementation allows overlapping memory allocation from the memory pool to multiple hosts, even though the entire memory pool is statically allocated to all hosts.

EVALUATION

Experimental Setup

Workload Description

To evaluate our algorithm, we used TPC-DS, a widely-used analytics benchmarks. From 99 TPC-DS queries, we selected join-heavy queries and generated the input dataset at a scale factor of 1000.

Additionally, we used three synthetic queries to highlight how MHJ differentiates from other join algorithms. These synthetic queries involve calculating aggregated sums of numeric columns after joining a pair of tables. We selected three pairs of relatively large tables that are frequently joined by the TPC-DS queries.

Join Algorithms and Software Setup

We developed a new scheme, the MHJ scheme, that enables MHJ in target queries' join operations. The results of the MHJ scheme are compared with Spark's baseline join schemes (SMJ scheme, SHJ Scheme). The BHJ algorithm-only scheme was excluded from

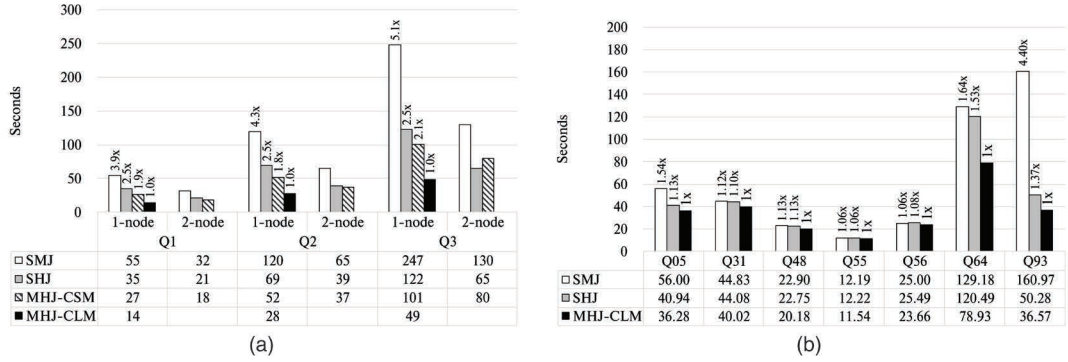


FIGURE 5. End-to-end execution times of (a) the synthetic and (b) TPC-DS queries on the TPC-DS dataset.

the comparison group due to memory capacity overhead, making it unsuitable as a single join scheme in Spark.

- *SMJ scheme:* Uses BHJ for tables smaller than 10 MB; otherwise, SMJ is applied (Spark's default scheme).
- *SHJ scheme*: Uses BHJ for tables smaller than 10 MB; otherwise, SHJ if join type and partition size are applicable.
- *MHJ scheme:* Always uses MHJ but disables parallel hash map builds for tables smaller than 10 MB.

We implemented the MHJ scheme in Spark version 3.4 and used the same version for the other schemes to ensure fairness. The rest of the software settings remain consistent: Ubuntu 20.04 LTS, Oracle Java Standard Edition 8 JDK, and Linux 5.17.

To estimate the implementation efforts, we tallied the number of added or modified lines in the software source code. Implementing the MHJ algorithm and our custom hash map in Spark required 992 lines of code. Additional shared off-heap module implementation and memkind modification added 300 and 283 lines, respectively.

Hardware Setup

The schemes are evaluated on the following cluster environments. First, a cluster consists of two server nodes, where the shared memory prototype is enabled only for the evaluation of the MHJ scheme. Second, a cluster composed of a single server node is used for distinguishing the performance benefits of the MHJ algorithm and the performance penalties by FPGA implementation. In detail, to estimate the performance of ASIC implementation without the penalties, we employed CXL local memory to a single node cluster since its performance is expected to be similar to ASIC-implemented shared memory. In a previous study,[2] a memory pool with 32 CPU sockets required a single switch and two retimers, which incurred 115-ns additional latency. As shown in Table 1, the latency of the CXL local memory is slightly better than, but much closer to, that of ASIC-implemented shared memory (which includes the additional latency), and both exhibit significantly lower latency than our prototype. Each server has one CPU socket with 64 cores and 16 DDR5 32-GB DIMMs. Spark is configured to 16 and 32 executors with four cores and 20-GB memory per executor in single and dual nodes, respectively.

Performance of Synthetic Query

Figure 5(a) displays the end-to-end execution times of the synthetic queries for our schemes and the baseline. On a single node, our MHJ-CXL shared memory (CSM) and MHJ-CXL local memory (CLM) outperform the baseline schemes, SMJ and SHJ, for all queries. Also, in these tests, we found that SHJ consistently outperforms SMJ, so we focused on comparing SHJ with our MHJ. The numbers on top of the bars in Figure 5(a) represent the execution times normalized to our MHJ-CLM. The results show that MHJ-CLM achieves approximately 2.5 times better performance than SHJ. In addition, although MHJ-CSM performs 1.8 ~ 2.1 times slower than MHJ-CLM due to FPGA implementation, it still slightly surpasses SHJ. In a two-node setup, MHJ-CSM outperforms SMJ but shows comparable performance to that of SHJ. Nevertheless, considering the performance gap between MHJ-CLM and MHJ-CSM, we anticipate that our MHJ on ASIC-implemented shared memory would outdo SHJ by around two times.

Performance of TPC-DS

In this section, we evaluate our MHJ algorithm with experimental results of the TPC-DS benchmark, presented in Figure 5(b). In this experiment, we only consider MHJ-CLM to purely confirm the benefits of our algorithm. On selected queries, MHJ-CLM outperforms SMJ and SHJ by up to 4.4 and 1.53 times, respectively. Similar to the synthetic query results, SHJ generally performs better than SMJ. Additionally, MHJ-CLM shows a 19% performance enhancement over SHJ in terms of geometric means.

Our performance gains on the TPC-DS falls short compared to the synthetic queries for two reasons. First, by predicate pushdown optimization,[13] filter operations after join are moved before joining, reducing the size of joining tables as a consequence. As table sizes become smaller, our MHJ algorithm loses the opportunity to reduce the repartitioning overhead of SMJ and SHJ algorithms. This case is observed in queries Q05, Q64, and Q93. Second, Spark uses the BHJ algorithm when one of the joining tables is smaller than 10 MB, regardless of setting the preferred join scheme. This occurs in queries Q31, Q48, Q55, and Q56, where the BHJ algorithm is employed instead of SMJ or SHJ. Nonetheless, MHJ outperforms BHJ by 6%~10% because it avoids copying hash maps.

CONCLUSION

In this article, we proposed a new join algorithm as a case study to identify the benefits of multihost shared memory enabled by CXL technology. We confirmed that the proposed algorithm outperforms conventional partition-wise algorithms, although additional effort is required for unaccustomed implementation under the shared memory model.

We believe that software algorithms should be modified to fully exploit the potential of shared memory, as demonstrated in our study. Further, we expect future work to determine whether the benefits justify the effort for modification in general use cases, beyond ours.

REFERENCES

1. "Spark SQL guide, performance tuning, configuration options." Apache Spark. Accessed: Jun. 23, 2025. [Online]. Available: https://spark.apache.org/docs/latest/sql-performance-tuning.html#other-configuration-options
2. D. S. Berger et al., "Design tradeoffs in CXL-based memory pools for public cloud platforms," *IEEE Micro*, vol. 43, no. 2, pp. 30–38, Mar./Apr. 2023, doi: 10.1109/MM.2023.3241586.
3. S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian, "A comparison of join algorithms for log processing in MaPreduce," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 975–986.
4. D. D. Sharma and I. Agarwal, "Compute express link™ 3.0 white paper," Compute Express Link, Beaverton, OR, USA, 2022. [Online]. Available: https://computeexpresslink.org/wp-content/uploads/2023/12/CXL_3.0_white-paper_FINAL.pdf
5. J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008, doi: 10.1145/1327452.1327492.
6. D. Gouk, M. Kwon, H. Bae, S. Lee, and M. Jung, "Memory pooling with CXL," *IEEE Micro*, vol. 43, no. 2, pp. 48–57, Mar./Apr. 2023, doi: 10.1109/MM.2023.3237491.
7. M. Ha et al., "Dynamic capacity service for improving CXL pooled memory efficiency," *IEEE Micro*, vol. 43, no. 2, pp. 39–47, Mar./Apr. 2023, doi: 10.1109/MM.2023.3237756.
8. H. Li et al., "Pond: CXL-based memory pooling systems for cloud platforms," in *Proc. 28th ACM Int. Conf. Architectural Support Programm. Lang. Operating Syst.*, 2023, *vol.* 2, pp. 574–587.
9. A. Metwally, "Scaling equi-joins," in *Proc. Int. Conf. Manag. Data*, 2022, pp. 2163–2176.
10. J. Nelson et al., "Latency-tolerant software distributed shared memory," in *Proc. USENIX Annu. Tech. Conf.*, Jul. 2015, pp. 291–305.
11. A.-C. Phan, T.-C. Phan, and T.-N. Trieu, "A comparative study of join algorithms in spark," in *Proc. Int. Conf. Future Data Secur. Eng.*, 2020, pp. 185–198.
12. Y. Sun et al., "Demystifying CXL memory with genuine CXL-ready systems and devices," in *Proc. 56th Annu. IEEE/ACM Int. Symp. Microarchit.*, 2023, pp. 105–121.
13. C. Yan, Y. Lin, and Y. He, "Predicate pushdown for data science pipelines," *Proc. ACM Manage. Data*, vol. 1, no. 2, pp. 1–28, 2023, doi: 10.1145/3589281.
14. M. Zaharia et al., "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proc. 9th USENIX Symp. Netw. Syst. Des. Implementation*, Apr. 2012, pp. 15–28.

JAEYUNG JUN is a senior staff engineer at SK hynix Inc., Icheon, 17336, South Korea. His research interests include distributed system and memory subsystems. Jun received his Ph.D. degree in electrical engineering from Korea University. Contact him at jaeyung.jun@sk.com.

HYUNWOONG AHN is a senior staff engineer at SK hynix Inc., Icheon, 17336, South Korea. His research interests include deep learning and distributed systems. Ahn received his M.S. degree in electrical engineering from Korea University. Contact him at hyunwoong.ahn@sk.com.

JOOHEE LEE is a staff engineer at SK hynix Inc., Icheon, 17336, South Korea. Her research interests include emerging memory system architecture and its application to improve data analytics and artificial intelligence applications. Lee received her M.S. degree in electronic engineering from Sogang University. Contact her at joohee.lee@sk.com.

JUNGMIN CHOI is a senior staff engineer at SK hynix Inc., Icheon, 17336, South Korea. His research interests include compute express link pooled memory architecture and near memory processing. Choi received his M.S. degree in electronic engineering from Sungkyunkwan University. Contact him at jungmin.choi@sk.com.

BYUNGIL KOH is a principal engineer at SK hynix Inc., Icheon, 17336, South Korea. His research interests include compute express link memory architecture and memory-centric computing architecture. Koh received his Ph.D. degree in electrical and computer engineering from University of Florida. Contact him at byungil.koh@sk.com.

DONGUK MOON is the director of the AI Platform Software team in Memory System Research at SK hynix Inc., Icheon, 17336, South Korea. His research interests include emerging memory solutions and storage solutions. Moon received his M.S. degree in electrical engineering from Stanford University and M.S. degree in data science from the University of California, Berkeley. He is the corresponding author of this article. Contact him at donguk.moon@sk.com.

HOSHIK KIM is senior vice president and fellow of Memory Systems Research at SK Hynix Inc., Icheon, 17336, South Korea. His research interests include processing-in-memory/processing-near-memory, CXL memory expansion/disaggregation, and AI training/inference systems. Kim received his Ph.D. degree in electrical engineering from University of Southern California. Contact him at hoshik.kim@sk.com.

Maximizing Interconnect Bandwidth and Efficiency in Nonvolatile Memory, Express-Based Key-Value Solid-State Devices With Fine-Grained Value Transfer

Junhyeok Park, Chang-Gyu Lee, and Soon Hwang, *Sogang University, Seoul, 04107, South Korea*
Seung-Jun Cha, *Electronics and Telecommunications Research Institute, Daejeon, 61012, South Korea*
Woosuk Chung, *Memory Systems Research, SK hynix, Seongnam, 13558, South Korea*
Youngjae Kim, *Sogang University, Seoul, 04107, South Korea*

The key-value solid-state drive (KV-SSD) redefines storage interfaces by integrating a key-value store directly within the device, offering native support for non-page-aligned key-value pairs. This architectural innovation enables KV-SSDs to offload storage management from the host system, positioning them as ideal candidates for resource disaggregation. However, KV-SSDs face significant challenges, notably, input–output amplification caused by conflicts with traditional storage protocols like Non-Volatile Memory Express (NVMe), which are designed around memory page units. Specifically, this results in a substantial increase in data traffic over interconnect between the host and SSD. This article introduces BandSlim, a novel solution addressing these challenges in data transfer by leveraging 1) NVMe command piggybacking for universally compatible, fine-grained transfers without requiring interconnect-level support and 2) the remote memory access capabilities of emerging Compute eXpress Link interconnects for higher-performance fine-grained transfers. Through extensive evaluations, BandSlim achieves up to a 97.9% reduction in Peripheral Component Interconnect Express traffic compared to conventional NVMe KV-SSDs.

Designing an efficient storage system necessitates reducing data movement costs from the host's memory to the storage media. However, traditional key-value stores (KVSs) like RocksDB[1] function as middleware on top of file systems. Consequently, user input–output (I/O) requests must navigate through the kernel's file system and block layers to execute data reads and writes to storage. This multilayer traversal incurs significant memory copying and kernel context switch overheads during I/O operations. In contrast, key-value solid-state drives (KV-SSDs) offer a substantial reduction in these overheads. KV-SSDs implement KVSs at the device controller level and provide native support for key-value operations. This allow users to bypass the file system and block layer when processing I/O requests, enabling lower latency and higher throughput compared to traditional SSD-based host-side KVSs. Such KV-SSDs are well suited for modern resource disaggregation architectures as they can effectively decouple storage management from the host system.[2]

KV-SSDs hold significant potential as next-generation storage devices but still face critical challenges,

Digital Object Identifier 10.1109/MM.2025.3572475
Date of publication 23 May 2025; date of current version 24 December 2025.

with the biggest issue being I/O amplification caused by limitations in existing protocols. Commercially and academically released KV-SSDs[3,4,5,6] utilize the Non-Volatile Memory Express (NVMe) protocol,[7] which is specifically engineered for block-based storage devices. These KV-SSDs implement the Physical Region Page (PRP) list for conveying payload, essentially inducing I/O amplification that originates from the size difference between the block and the key value. This results in a substantial increase in data traffic over the interconnect between the host and the SSD.

One common solution to this problem has been host-side batching, where enough key-value pairs are grouped to align with the memory page size, as implemented in KV-SSDs like KV-CSD[4] through bulk PUT operations. However, buffering entries on the host side risks data loss during power failure, making this approach unsuitable for mission-critical applications where data integrity is essential. For those scenarios where data persistence is critical and I/O transactions occur at the key-value pair level as defined by NVMe standard, a more fundamental solution is required.

To address these issues, we introduce *BandSlim*, which minimizes data traffic over host–SSD interconnect when transferring small key-value pairs. *BandSlim* employs two approaches: the first utilizes NVMe commands to achieve an inline value transfer at the storage protocol level, while the second leverages the emerging Compute eXpress Link (CXL) interconnect to address the transfer at the interconnect protocol level. The first approach piggybacks values smaller than a memory page to NVMe commands using reserved fields. We observed that this method significantly reduces data traffic over the host–SSD interconnect. The second approach, which uses CXL's remote memory access capability, aims to improve this further by offering a more fine-grained, higher-performance value transfer solution, bypassing the limitations of piggybacking. It exposes the device-side NAND page buffer to the host's CPU and transfers values using CXL.mem operations. The first method can be implemented directly in existing NVMe protocols without requiring new interconnect support, while the second approach is optimized for systems that support CXL, allowing for more advanced and efficient fine-grained transfers.

In both methods, however, as the value size grows, the performance of fine-grained transfer becomes less favorable compared to the traditional PRP-based transfer. Therefore, *BandSlim* employs an adaptive transfer strategy, dynamically switching between fine-grained and PRP-based transfers to balance traffic reduction with optimal response times for varying value sizes.

For evaluation, we implemented *BandSlim* on a state-of-the-art field-programmable gate array (FPGA)-based NVMe KV-SSD[5] using the Cosmos+ OpenSSD platform[8] and estimated CXL.mem-based value transfer's benefits using real CXL.mem measurements. We demonstrated that *BandSlim* achieves up to 97.9% traffic reduction compared to an NVMe KV-SSD without *BandSlim*.

The contributions of this article are as follows:

- We Identify traffic amplification in the host–SSD interconnect, specifically in KV-SSDs, and propose NVMe and CXL-based solutions to minimize waste.
- We propose the design of a CXL-based storage interface and demonstrate use cases where the interface shows clear advantages, particularly in KV-SSDs.
- We demonstrate the tradeoff between data traffic and response time in fine-grained value transfer and effectively resolve it by using an adaptive approach.

BACKGROUND AND MOTIVATION

Storage Stack of KV-SSDs

The storage stack of KV-SSDs consists of user-level key-value APIs, a key-value device driver and controller based on protocols like NVMe, and an in-storage KVS. The key-value application programming interface (API) offers point and range queries (e.g., PUT and SEEK). The size of the key and the value in these APIs is handled as arbitrary lengths, not in block units (*key-value interfaces*). In case of log-structured merge (LSM)-based KV-SSDs with a key-value separation,[3,4,5,6] a pair of key and value address is stored in the LSM tree, and a value is stored in the Value Log (vLog). The vLog is a linear logical NAND flash address space that can be divided into multiple logical NAND pages. Each value is appended to the vLog sequentially, filling logical NAND pages, which are mapped to physical NAND pages by the Flash Translation Layer (FTL). The entries of the LSM tree point to corresponding values inside the vLog.

NVMe-Based Key-Value Pair Transfer

Within the NVMe key-value interface, when writing key-value pairs, the NVMe driver stores a key and metadata in the reserved fields of the NVMe command. The payload, which is the value in this context, is transferred via the PRP as the block interface part of the NVMe specification does.[7] The PRP is a linked

list whose entry describes the addresses of physical memory pages of the host memory. One or more memory pages where the value is stored are specified to be transferred. Subsequently, the driver inserts the NVMe command into the submission queue and rings the doorbell to notify the device of the write request. The NVMe controller fetches the command from the queue, interprets it, and identifies the pages for copying from the received PRP list.

To initiate the value transfer, the controller triggers a direct memory access (DMA) transaction, which copies pages from the host memory to the device memory. Later, the controller inserts the received key to the memory component of the LSM tree and MemTable and writes the value to the vLog (see Figure 1). The reverse operation, involving the transfer of values from the KV-SSD to the host, follows a similar process.

Peripheral Component Interconnect Express Traffic Amplification in NVMe KV-SSDs

As in typical KVSs, the key and the value size are variable and not necessarily aligned to a memory page. According to Meta, RocksDB in a production environment experiences a size of values that nearly do not reach 100 bytes on average,[9] which is far less than the 4-KB memory page size. Consequently, a KV-SSD must be capable of effectively handling requests for such variable-sized, small values.

However, the current NVMe key-value interface causes significant inefficiencies because it adheres to the same procedure as the block-based NVMe protocol when transferring values to or from the device. *Specifically, its data transfer method, the PRP, restricts DMA transfers to occur in units of 4 KB, the size of a memory page*. As shown in Figure 1, consider a scenario where the value size is 32 bytes (❶). In this case, one 4-KB memory page that temporarily holds the value is specified by the PRP, and a DMA copy of 4 KB occurs. On the other hand, if the value size slightly exceeds the memory page size, such as (4 K + 32) bytes (❷), two memory pages are required to accommodate the value. Consequently, the DMA facilitated by the PRP transfers 8 KB of data. This amplified data traffic significantly raises energy consumption of the system, possibly increasing the total cost of ownership.

Experimental Analysis of Traffic Amplification

We measured Peripheral Component Interconnect Express (PCIe) traffic from a host to a KV-SSD[5] by issuing 1 million key-value writes with variable-sized values by using the Intel PCM[10] to track PCIe traffic.

Figure 2(a) shows the total PCIe traffic during the experiments, with traffic increasing stepwise at 4-KB value-size boundaries. For example, total PCIe traffic for 1 and 4-KB values is approximately 4 GB, indicating constant transfer volumes up to 4 KB. This pattern repeats for value-size ranges like 5–8 KB. Average transfer response times display similar cascading patterns.

The issue becomes more severe with smaller values. The traffic amplification factor, defined as *the ratio of PCIe traffic to data size*, surges for values under 1 KB. As shown in Figure 2(b), transferring a value of 32 bytes generates approximately 4 KB of PCIe data traffic, roughly 130 times the requested data size.

Limitations of Existing Methods

The NVMe protocol currently offers another transfer method besides the PRP, called the *Scatter-Gather List (SGL)*.[7] Unlike the PRP, the SGL supports variable-sized DMAs across scattered memory segments. However, it has been reported that the cost of enabling the SGL

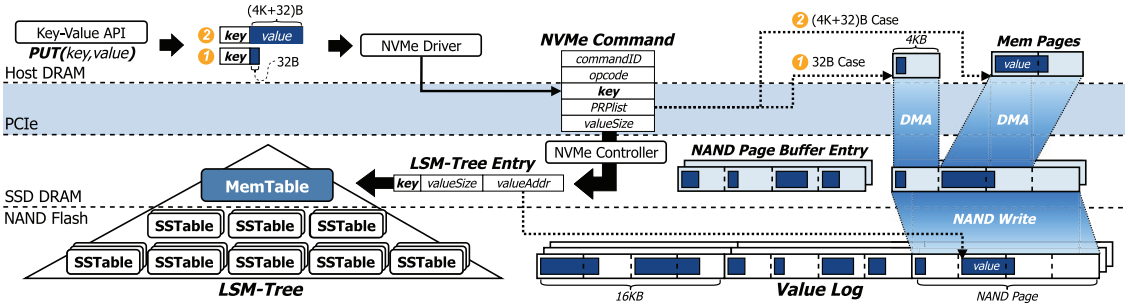


FIGURE 1. Data flow of two cases of key-value transfers with a sub-4-KB payload [32 bytes (B)] and a more than 4-KB payload [(4K + 32)B] regarding Peripheral Component Interconnect Express (PCIe) interconnect traffic amplification in LSM-based NVMe KV-SSDs.

outweighs the benefit for I/Os smaller than 32 KB,[11] indicating that using the SGL for small-value transfers is not advisable and realistic.

PROPOSED DESIGN: *BANDSLIM*

Fine-Grained Value Transfer Over NVMe

Due to space constraints, a detailed discussion of *BandSlim*'s first proposed technique, NVMe command piggybacking, is provided in the conference version of this article.[12] The summary is as follows: the method leverages the NVMe command, which is 64 bytes in size, to enable fine-grained transfer of small values. Considering most values in the real-world are less than 64 bytes,[9] it repurposes up to 35 bytes of unused fields in the NVMe key-value write command for value transfer. For values that exceed 35 bytes, the piggybacking method introduces a `transfer command`, which sends the remaining bytes in 56 bytes per command after the initial write command. This approach significantly reduces PCIe traffic under real-world KVS workloads.

Fine-Grained Value Transfer Using CXL

Although the piggybacking method resolves the traffic amplification issue, it is still constrained by several key limitations. First, the overhead associated with creating, submitting, processing, and releasing NVMe commands is significant, particularly when dealing with a large value that demands issuing multiple commands, which makes this method effective only for transfers of tiny values, typically in the range of tens of bytes. Second, it cannot completely eliminate traffic bloating as it still depends on NVMe commands, which requires extra traffic for signaling doorbells and completions.

To overcome the limitations of repetitive NVMe command overheads in the piggybacking method, we also propose a design that fundamentally supports fine-grained transfer at the interconnect level using CXL. For this, *BandSlim* defines the NAND page buffer as host-managed device memory (HDM), making it directly accessible to the host through CXL.mem operations. During a CXL device enumeration, the driver queries the Base Address Register (BAR) and the size of the HDM to map the BAR and HDM within the host's system memory. The host CPU's system bus includes CXL root ports (RPs), establishing connections with KV-SSDs as endpoints. After that, the driver can transmit values via CXL.mem write operations.

Figure 3 illustrates the value transmission process in this method. The driver manages a CXL.mem write pointer (CMWP) to track the current position in the NAND page buffer. When a user initiates a key-value write, the NVMe command includes only the key, unlike the previous piggybacking method. ❶ The driver initiates a memory copy request to the CMWP. ❷ The host CPU then sends the value to the NAND page buffer at the CMWP location via CXL.mem. ❸ The CXL RP receives this request, converting it into *flits*, that is, CXL transaction units (*BandSlim* operates based on the default 68-byte *flit* mode). ❹ The CXL controller decompresses the *flits*, adjusts target addresses by subtracting the HDM base address, and forwards the request to the dynamic random-access memory (DRAM) controller, ❺ which sends the payload to the DRAM location indicated by the CMWP. ❻ The controller notifies the driver to update the CMWP via NVMe command completion.

As the FTL operates independently, the host cannot determine when each buffer entry is flushed. Thus, *BandSlim* always calls *clflush()* for CXL.mem writes to ensure that the payload is sent to device memory.

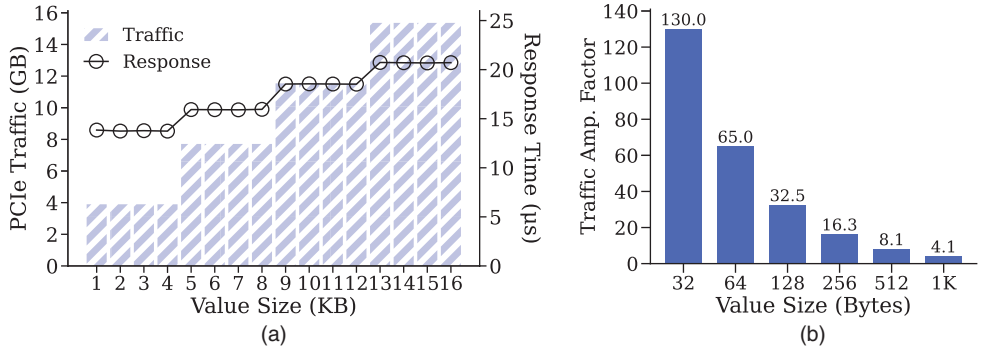


FIGURE 2. Measurements of total PCIe traffic with average response time and PCIe traffic amplification (amp.) factor for varying value sizes, with NAND I/O disabled. (a) Total PCIe traffic and average response time. (b) Traffic amplification time.

Adaptive Value Transfer Method

As the value size increases, the performance of both fine-grained value transfer methods starts to degrade. The first method, in particular, suffers due to the accumulation of overheads in generating NVMe commands and synchronously handling them within the device, resulting in much longer transfer times compared to a conventional PRP-based transfer. To tackle these issues, *BandSlim* utilizes a threshold-based reactive method that selects the most suitable transfer method from both the fine-grained and PRP-based transfers based on the size of the value. The threshold is identified by exploratory runs using benchmarks depicted in the evaluation. During the benchmark runs, various value sizes ranging from 4 bytes to 8 KB are tested through millions of PUT commands to compare transfer times.

The threshold is denoted as $\tau_\alpha = \alpha \cdot \tau$, where τ represents the value size at which the fine-grained transfer becomes less efficient than PRP-based transfers. *BandSlim* employs the following strategy:

$$\text{Transfer method} = \begin{cases} \text{Fine-grained transfer}, & \text{if value size} < \tau_\alpha, \\ \text{PRP-based transfer}, & \text{if value size} \geq \tau_\alpha. \end{cases}$$

The scaling factor α allows users to adjust the threshold: increasing α raises the threshold, while setting α to one retains the default threshold. For users who prioritize response time, α can be set to one. For those who focus on traffic reduction, α can be increased to delay the transition point where PRP-based transfers become more efficient. This approach ensures efficient handling of value sizes ranging from subpage to large.

In-Device Value Packing Mechanism

Our conference paper proposed the *selective packing with backfilling policy* to address the NAND page write amplification problem in NVMe KV-SSDs.[12] This policy aims to reduce the overhead of copying large values during the packing process within the device under adaptive value transfer by selectively packing only small values transmitted via fine-grained transfer, while large values are placed to page-aligned addresses via PRP-based DMA without packing. To minimize internal fragmentation, the policy allows small values to fill gaps between large values at page-aligned addresses and also avoid them with a DMA log table (DLT).

Figure 4 illustrates the process of packing under CXL.mem-based transfer. ❶ When a user requests a large value write that exceeds the threshold τ_α, the value is transferred via PRP DMA over NVMe (CXL.io). ❷ The controller completes the DMA and creates a DLT entry based on the DMA destination address and value size. ❸ This entry is sent to the host in the NVMe completion queue entry (CQE), ❹ where the *BandSlim* driver caches it. For a small value below the threshold τ_α, the driver checks if adding the value size to the CMWP exceeds the specified address in the cached DLT head entry. ❺ and ❻ If it does not, the value is transferred to the CMWP address over CXL.mem with *clflush()* enabled, while only the key is sent in an NVMe command. ❼ and ❽ If the next small-value size exceeds the address in the cached DLT head entry, the CMWP updates to the DLT entry's address plus the value size, and the value is transferred to the updated CMWP address via CXL.mem. ❾ For another large value, the transfer occurs via PRP DMA, with the destination address aligned to the nearest page boundary, tracked by the controller. ❿ The new address and value size are added to the DLT and cached on the host, as shown in Figure 4.

As only the NAND page buffer is exposed as HDM, while CMWP and DLT prevent conflicts between CXL.mem and PRP DMA, *BandSlim* preserves data consistency and metadata integrity. Plus, because *BandSlim* does not modify the core components of KV-SSDs,

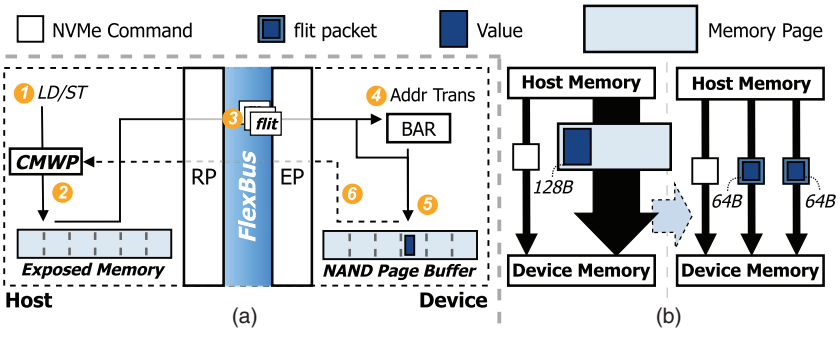


FIGURE 3. (a) CXL.mem memory write mechanism and (b) fine-grained value transfer via CXL.mem. CMWP: CXL.mem write pointer. EP: endpoint; load/store (LD/ST).

including key-value indexing and FTL, the existing read and query performance remains unaffected.

EVALUATION

Evaluation Setup

We used the state-of-the-art NVMe KV-SSD[5] on the Cosmos+ OpenSSD platform[8] as the baseline to verify our solutions. The platform consists of an ARM-based Xilinx Zynq-7000 system on chip (SoC), 1 TB of NAND (four channels, eight ways), and a PCIe Gen2 × 8 interconnect, paired with a host node featuring 64 cores of Intel Xeon Gold 6226 R CPU, 384 GB of DDR4 memory, and Ubuntu 22.04 operating system. The SoC operates the *BandSlim* key-value, PCIe interface, DRAM, and NAND flash controllers, while the host node runs the *BandSlim* key-value driver and benchmarks.

As publicly accessible SSDs and FPGA boards that support CXL remain extremely scarce (including ours) we validated our designs by implementing the piggybacking mechanism on the OpenSSD-based setup, while the CXL.mem-based transfer was evaluated through simulations using measured CXL.mem response times. The traditional PRP-based transfer and the NVMe piggybacking method underwent thorough system-level validation on the OpenSSD setup, providing reliable performance measurements. We then scaled the measured payload transfer times of baseline and piggybacking to PCIe Gen5 and incorporated the NVMe overheads into the CXL.mem response times to simulate the CXL.mem-based transfers. This approach allowed us to directly compare the performance of PRP-based transfers, NVMe piggybacking, and CXL.mem-based transfers, demonstrating the potential benefits of CXL.mem optimizations.

For performance evaluations, we used db_bench, a benchmarking tool used in RocksDB.[1] We enabled db_bench to send NVMe key-value commands to the OpenSSD platform through the NVMe pass through.

We conducted various workload patterns to verify our proposed design, which we describe as follows:

- *WorkloadA* [*W(A)*]: A db_bench's *fillseq* pattern where keys are sequential and value sizes are fixed. It serves as a baseline for evaluating performance under uniform and predictable write patterns.
- *WorkloadB* [*W(B)*]: This writes 1 million random key-value pairs with value sizes of 8 bytes or 2 KB at a 9:1 ratio. This tests the transfer method's efficiency in optimizing for frequent small-value writes.
- *WorkloadC* [*W(C)*]: Similar to *W(B)* but reverses the value-size ratio to 1:9, emphasizing scenarios with large-value dominance. The goal is to reveal the tradeoffs in handling small versus large values.
- *WorkloadD* [*W(D)*]: This workload writes values of sizes (eight; 16; 32; 64; 128; 256; and 512 bytes; 1 KB; and 2 KB) in random order, totaling 1 million, with each size having an equal ratio. This evaluates the transfer method's adaptability to diverse data sizes.
- *WorkloadM* [*W(M)*]: This is a db_bench's *mixgraph All_random*9 workload that reflects real-world characteristics with a maximum value size of 1 KB and almost 70% of values being less than 35 bytes. We have modified *mixgraph* to issue only 1 million PUTs.

In all the experiments, we used unique keys of 4 bytes generated by a hash function with a random seed.

We conducted evaluations for the following designs:

- *Baseline:* The state-of-the-art NVMe KV-SSD.5 It employs the PRP-based page-unit value transfer.

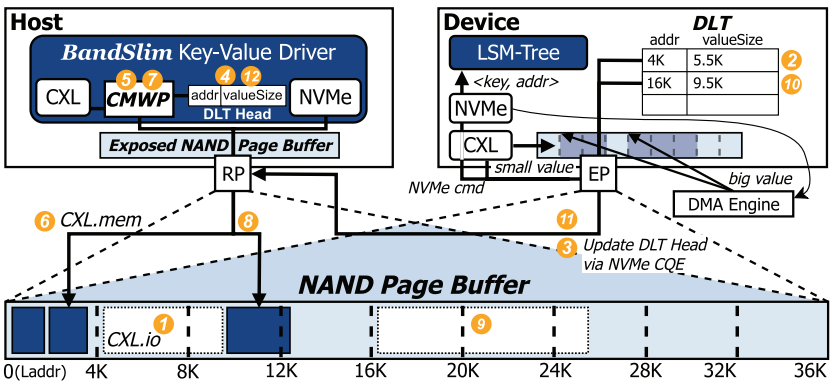


FIGURE 4. The process of *selective packing with backfilling* over the NAND page buffer with CMWP and DLT. CQE: completion queue entry.

- *Piggyback:* Transfers values only via piggybacking.
- *CXL.mem-Based Fine-Grained Value Transfer (CXL-FGT):* Transfers values only via CXL.mem.
- *Adaptive:* Transfers values via the adaptive method.

Effects of Fine-Grained Value Transfer

Piggybacking-Based Value Transfer

The evaluation of PCIe traffic and response times for *Piggyback* on a real platform is detailed in our conference paper[12] as space constraints prevent an in-depth discussion here. Briefly, *Piggyback* achieves up to 97.9% traffic reduction for values of 4–32 bytes but becomes less efficient as the value size increases due to the overheads of trailing commands, approaching *Baseline* at approximately 2 KB and surpassing it for larger sizes. Response times are halved for values up to 32 bytes but degrade for larger sizes due to the trailing commands' overheads.

CXL.mem-Based Value Transfer

To evaluate *CXL-FGT*, we first measured write response times for various value sizes using a 256-GB CXL memory device connected via PCIe Gen5, consistently invoking *clflush()*. These results, obtained on a dedicated server with a CXL-supported AMD EPYC 9754 configured as a CPU-less nonuniform memory access node, served as the foundation for a performance simulation model of *CXL-FGT*.

Next, we analyzed the response times[12] for *Baseline* and *Piggyback*, measured from our OpenSSD (PCIe Gen2) setup. We extracted the payload transfer time for various payload sizes from the total response times by separating the time spent on generating, submitting, processing, and completing NVMe commands. We then scaled the extracted payload transfer times by a factor of 6.4, reflecting the higher giga transfer rate of Gen5 (32 GT/s) compared to Gen2 (5 GT/s). Although actual performance gains may be smaller due to system overheads, this adjustment assumes an ideal scenario focused purely on bandwidth improvements. We added the NVMe overhead back to the scaled payload transfer times of *Baseline* and *Piggyback* to estimate their response times on PCIe Gen5. Finally, we created a performance simulation model for *CXL-FGT* by combining the measured CXL.mem write response times with the same NVMe overheads used for *Baseline* (*CXL-FGT* always require a single NVMe command). This approach enables a direct comparison of *Baseline*, *Piggyback*, and *CXL-FGT* designs. Figure 5(a) illustrates the time breakdown and adjustment used to estimate the response times of *Baseline* and *Piggyback* on PCIe Gen5 for values of 32 bytes.

Using the *CXL-FGT* simulation model and the estimated response times of *Baseline* and *Piggyback*, we evaluated the performance of each transfer mode

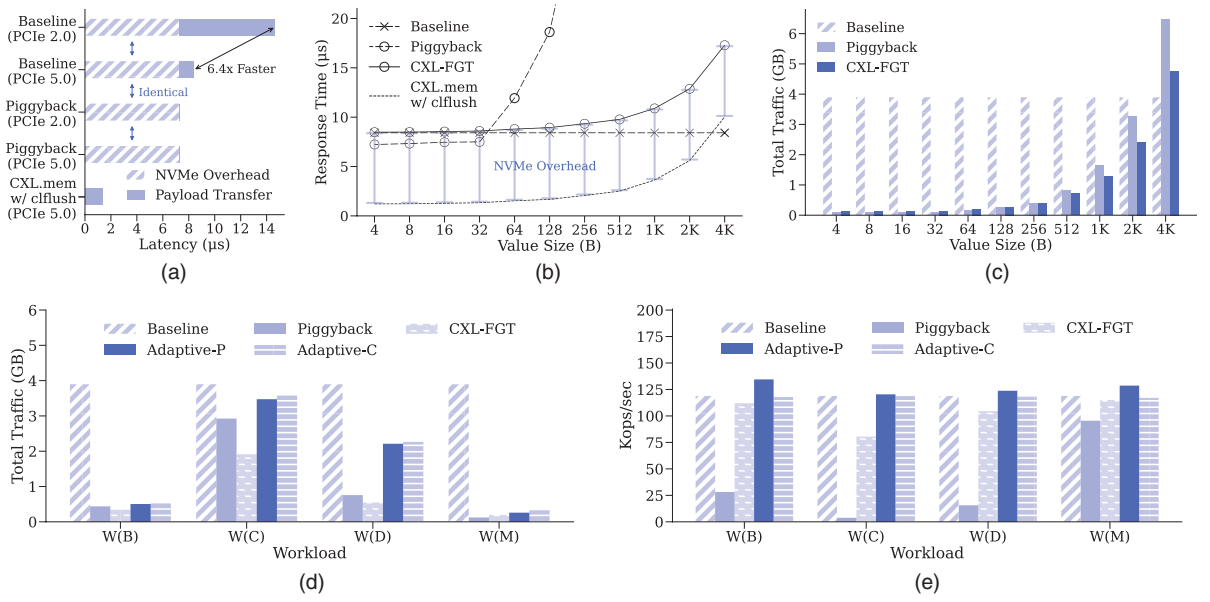


FIGURE 5. (a) Breakdown of estimated response times for values of 32 bytes on PCIe Gen5 using *Baseline* and *Piggyback*. (b) Estimated response times [*W*(*A*)] and (c) total PCIe traffic[*W*(*A*)] for *Piggyback* and *CXL-FGT* with 1 million key-value pairs [*W*(*A*)] of varying value sizes transferred from host to device memory. (d) Estimated PCIe traffic and (e) average throughput for various workloads [*W*(*B*), *W*(*C*), *W*(*D*), and *W*(*M*)] on PCIe Gen5. B: bytes.

across different value sizes using *WorkloadA*. The results, presented in Figure 5(b), reveal key insights. First, *CXL-FGT* performs worse than *Piggyback* for value sizes of 4–32 bytes due to the mandatory NVMe command overhead separated from CXL transfers. This limitation could be mitigated by integrating piggybacking with *CXL-FGT*. Second, *CXL-FGT* does not outperform *Baseline* on PCIe Gen5, again, highlighting a limitation due to the mandatory NVMe command overhead. However, as expected, *CXL-FGT* significantly outperforms *Piggyback* starting at the 64-bytes case, demonstrating how CXL.mem mitigates the performance degradation of *Piggyback* for larger data sizes. For instance, in the 25-bytes case, *CXL-FGT* achieves nearly 4× faster response times compared to *Piggyback*, with an even greater performance gap as value sizes increase.

Figure 5(c) shows that *CXL-FGT* generates less traffic compared to *Piggyback*. Using CXL *flit* units of 68 bytes, we estimated *CXL-FGT*'s traffic, while *Baseline* and *Piggyback* reflect measured values.[12] CXL enables finer granularity because *Piggyback* incurs additional traffic with each `transfer command` submission, such as doorbell rings and tail pointer reads. CXL avoids this overhead, allowing more efficient data movement. However, note that piggybacking's primary strength lies in resolving KV-SSD bottlenecks without requiring new interconnect technologies.

Effects of Adaptive Value Transfer

The evaluation results of the adaptive value transfer using *Workloads B*, *C*, *D*, and *M* on a real platform is detailed in our conference paper.[12] Briefly, *Adaptive* consistently outperforms *Piggyback* and *Baseline* by transitioning from piggybacking to PRP at 128 bytes [identified as the threshold based on experiments with [*W*(*A*)], achieving the best performance in all workloads. It reduces traffic by 93.3% for *W*(*M*) while improving throughput by 12% over *Piggyback*, and in *W*(*C*), it increases throughput nearly 13-fold despite generating 18% more traffic than *Piggyback*.

To assess the impact of CXL.mem-based transfer under various workloads, we simulated *Workloads B*, *C*, *D*, and *M* using the results from Figure 5(b) and (c). Figure 5(d) shows the traffic results, where *CXL-FGT* achieved lower overall traffic than *Piggyback* in all workloads except *W*(*M*) despite the additional traffic overhead compared to *Piggyback* for transferring values of 4–32 bytes. This reduction stems from avoiding frequent NVMe command overheads by using CXL.mem, with the benefit most evident in *W*(*C*) due to its larger values. For the *Adaptive* method, we evaluated scenarios employing piggybacking (*Adaptive-P*) and CXL.mem (*Adaptive-C*). The threshold for switching the transfer mode in *Adaptive-P* was set to 64 bytes, based on the results in Figure 5(b). The same threshold was applied to *Adaptive-C* to ensure a fair comparison between piggybacking and CXL.mem. As expected, both methods exhibited higher traffic than fine-grained transfers. Moreover, as *Piggyback* inherently resulted in lower traffic compared to *CXL-FGT* for value sizes between 4 and 32 bytes, *Adaptive-C* exhibited slightly higher traffic than *Adaptive-P*.

Figure 5(e) presents the average throughput derived by the simulated response times. Across all workloads, *CXL-FGT* consistently outperformed *Piggyback*, further demonstrating the effectiveness of CXL.mem-based transfers. Meanwhile, *Adaptive-P* achieved the best performance in all workloads, while *Adaptive-C* generally showed performance comparable to *Baseline*. Again, this result is attributed to the inherent limitations of *CXL-FGT*, which retains the mandatory NVMe command overhead. In particular, workloads with a high proportion of value sizes between 4 and 32 bytes [e.g., *W*(*B*) and *W*(*M*)] highlighted the limitations of *CXL-FGT*, which adheres to NVMe routines, resulting in *Adaptive-C* performing worse than *Adaptive-P*.

In summary, *CXL-FGT* achieves finer-grained traffic and significantly better performance than *Piggyback* as value sizes increase. However, as the method maintains NVMe transactions, the performance advantage of fine-grained transfer over *Baseline* diminishes with the increased bandwidth of PCIe Gen5. Furthermore, *CXL-FGT* does not fully utilize the cache coherency of CXL as it actively invokes *clflush()* for each write. We plan to address these limitations by developing new storage protocols that leverage the full potential of CXL.

CONCLUSION

BandSlim tackled the PCIe traffic amplification challenges in KV-SSDs, which arose from conflicts between non-page-aligned key-value pairs and NVMe protocols designed for memory page units. *BandSlim* leveraged NVMe command piggybacking and CXL.mem to transfer values in a fine-grained and efficient manner. Extensive evaluations showcased that *BandSlim* significantly reduced traffic and optimized data transfer efficiency.

ACKNOWLEDGMENTS

This work was funded in part by the National Research Foundation of Korea (NRF) grant funded by the Korean Government (MSIT) (Grant Nos. RS-2024-00453929 and RS-2025-00564249), in part by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea Government (MSIT) under Grant RS-2021-II210528, and in part by the Institute

of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (Grant No. 2018-0-00503, Researches on next generation memory-centric computing system architecture). Youngjae Kim is the corresponding author.

REFERENCES

1. "A persistent key-value store," *RocksDB*, 2014. Accessed: Nov. 30, 2024. [Online]. Available: http://rocksdb.org
2. A. Tomlin. "Why KV SSD will replace ZNS." SNIA, 2022. Accessed: Nov. 30, 2024. [Online]. Available: https://www.snia.org/educational-library/why-kv-ssd-will-replace-zns-2022
3. C. G. Lee et al., "iLSM-SSD: An intelligent LSM-tree based key-value SSD for data analytics," in *Proc. IEEE 27th Int. Symp. Model., Anal., Simul. Comput. Telecommun. Syst. (MASCOTS)*, 2019, pp. 384–395, doi: 10.1109/MASCOTS.2019.00048.
4. I. Park et al., "KV-CSD: A hardware-accelerated key-value store for data-intensive applications," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, 2023, pp. 132–144, doi: 10.1109/CLUSTER52292.2023.00019.
5. S. Lee et al., "Iterator interface extended LSM-tree-based KVSSD for range queries," in *Proc. 16th ACM Int. Syst. Storage Conf. (SYSTOR)*, 2023, pp. 60–70.
6. D. Min et al., "A multi-tenant key-value SSD with secondary index for search query processing and analysis," *ACM Trans. Embedded Comput. Syst.*, vol. 22, no. 4, pp. 1–27, Jul. 2023, doi: 10.1145/3590153.
7. "Key value command set specification," *NVM Express Inc.*, 2024. [Online]. Available: https://nvmexpress.org/specifications/
8. J. Kwak, S. Lee, K. Park, J. Jeong, and Y. H. Song, "Cosmos+ OpenSSD: Rapid prototype for flash storage systems," *ACM Trans. Storage*, vol. 16, no. 3, pp. 1–35, Jul. 2020, doi: 10.1145/3385073.
9. H. Cao, S. Dong, S. Vemuri, and D. H. C. Du, "Characterizing, modeling, and benchmarking RocksDB key-value workloads at facebook," in *Proc. 18th USENIX Conf. File Storage Technol. (FAST)*, 2020, pp. 209–224.
10. "Intel: PCM," GitHub, 2010. Accessed: Dec. 1, 2024. [Online]. Available: https://github.com/intel/pcm
11. "NVMe: Add scatter-gather list (SGL) support in NVMe driver," OpenWRT, 2017. [Online]. Available: https://merlin.infradead.org/pipermail/linux-nvme/2017-July/011956.html pcm
12. J. Park et al., "BandSlim: A novel bandwidth and space-efficient KV-SSD with an escape-from-block approach," in *Proc. 53rd Int. Conf. Parallel Process. (ICPP)*, 2024, pp. 1187–1196.

JUNHYEOK PARK is pursuing his M.S. degree at Sogang University, Seoul, 04107, South Korea. His research interests include next-generation NAND flash drives and file and storage systems. Park received his B.S. degree in computer science and engineering Sogang University. Contact him at junttang@sogang.ac.kr.

CHANG-GYU LEE is pursuing his Ph.D. degree in computer science and engineering at Sogang University, Seoul, 04107, South Korea. His research interests include operating systems and file and storage systems. Lee received his B.S. degree in software and computer engineering from Ajou University. Contact him at changgyu@u.sogang.ac.kr.

SOON HWANG is pursuing his Ph.D. degree at Sogang University, Seoul, 04107, South Korea. His research interests include file and storage systems for high-performance computing and artificial intelligence. Hwang received his B.S. and M.S. degrees in computer science and engineering from Sogang University, in 2021 and 2023, respectively. Contact him at soonhw@sogang.ac.kr.

SEUNG-JUN CHA is a senior researcher at the Electronics and Telecommunications Research Institute, Daejeon, South Korea. His research interests include enhancing performance in system software, such as operating systems and database management systems, and he is currently focused on advancing memory interconnect technologies. Cha received his B.E., M.E., and Ph.D. degrees in computer engineering from Chungnam National University, in 2006, 2008, and 2013, respectively. Contact him at seungjunn@etri.re.kr.

WOOSUK CHUNG is a vice president at SK hynix, where he leads the Software Solution Group in the Memory Systems Research organization. Chung received his B.S. degree in electronic engineering from Hanyang University in 1997 and his M.S. degree in engineering from Hanyang University in 1999. His research interests include AI and big data analytics optimized memory and storage systems. Contact him at woosuk.chung@sk.com.

YOUNGJAE KIM is a professor with the Department of Computer Science and Engineering, Sogang University, Seoul, 04107, South Korea. His research interests include operating systems, file and storage systems, and parallel and distributed systems. Kim received his Ph.D. degree in computer science and engineering from The Pennsylvania State University. Contact him at youkim@sogang.ac.kr.

FS^2: A Fast, Scalable, and Flexible Switching System for Emerging Interconnects

Heetaek Jeong, Kanghyun Choi, and Hamin Jang, *Seoul National University, Seoul, 08826, South Korea*
Dongup Kwon and Eunjin Baek, *MangoBoost, Seoul, 08806, South Korea*
Pyeongsu Park, *Meta, CA, 94089, USA*
Jangwoo Kim, *Seoul National University, Seoul, 08826, South Korea*

Cache-coherent interconnects such as Compute Express Link and Cache Coherent Interconnect for Accelerators have been introduced thanks to their cache coherence for a shared address space. However, we observe that it is difficult to scale such interconnects with 10–100-s devices to fulfill the ever-increasing memory demands of big data applications. In this article, we propose a switch-assisted scalable system architecture on top of those interconnects. Specifically, we introduce shared caches in interconnect switches to efficiently exploit data reuse opportunities and overcome topological limitations by flexibly changing topologies based on workload patterns. Our evaluation shows that our switch-assisted architecture provides higher scalability and up to 4.4 × higher performance than native designs.

Emerging cache-coherent (cc) interconnects, such as Compute Express Link (CXL)[1] and Cache Coherent Interconnect for Accelerators (CCIX),[2] are rapidly gaining industry support from more than 50 companies. With cc interconnects, devices and CPUs can directly access remote memory using standard load/store operations, reducing programming complexity and communication overhead while improving performance. These features align with the needs of big data applications, such as recommendation systems[3] and in-memory databases, which rely on fast, efficient data transfers across devices.

Unfortunately, we observe that the scalability of emerging interconnects is limited when applied to big data applications. We identify that the limited scalability of the emerging interconnects stems from *high overheads of accessing shared memory* with two major sources. First, the cache-hit ratio decreases significantly with more devices as each device's private cache remains inaccessible to other devices. Although the overall cache size increases, every private cache miss requires shared memory accesses. Figure 1(b) shows that the cache-hit ratio reduces from 42% of single-device systems to 5% of 64-device systems. Second, the cache-miss penalty grows substantially due to the increasing distance between devices and the serialization point, exacerbated by topological limitations. As a result, memory access latency rises up to 14 × that of local, noncoherent memory access [(Figure 1(c)].

Our in-depth analysis identifies four critical challenges in reducing the overhead of shared memory access in large-scale systems using emerging interconnects. First, device-side scaling proves inefficient, offering marginal benefits (15% higher hit ratio) at the cost of a 4× larger cache. Second, the complex standardization process, involving many vendors, hinders the incorporation of innovative ideas into interconnect specifications. Third, designing fail-safe mechanisms is challenging due to the inherent complexity of cc protocols, such as message reordering. Finally, performance varies significantly with workload behaviors, necessitating flexibility to adapt effectively.

To overcome the scalability challenges, we propose FS^2, a fast, scalable, and flexible switching system with a novel FS^2-switch architecture. Specifically, FS^2-switches provide shared caches that run on the unmodified emerging interconnects. As nearby

Digital Object Identifier 10.1109/MM.2025.3574732
Date of publication 30 May 2025; date of current version 24 December 2025.

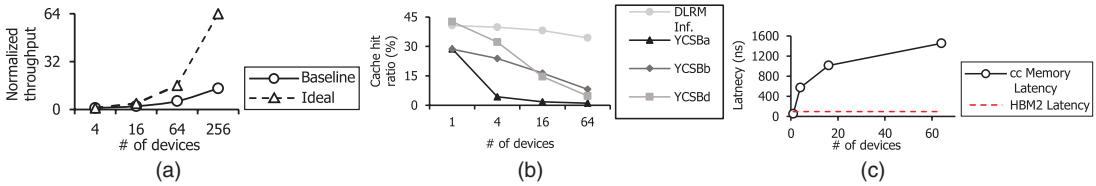


FIGURE 1. Analysis of limited scalability in cache-coherent nonuniform memory access (ccNUMA) interconnects. (a) Scalability. (b) Cache-hit ratio. (c) Memory latency. DLRM: deep learning recommendation model; Inf.: inference.

switch caches can serve the data to devices, we can both 1) exploit the locality scattered through the devices and 2) significantly shorten the average critical path of the cache-miss penalty. An FS^2-switch also enables topological flexibility (*FS^2-topology*) through configurable routing tables to 1) fully leverage the data reuse opportunity of our switch cache, which can be significantly differed by the routing path, and 2) strike a balance between per-device link bandwidth and the critical path of cache misses. Moreover, we design *FS^2-scheduling*, an FS^2-switch-aware request scheduling algorithm that assigns user requests based on their data affinity to maximize data reuse for device/switch caches.

We evaluate the validity and the performance of FS^2 through formal/empirical validation processes and our full-stack simulator. The validation results show that FS^2 is safe and deadlock-free while preserving the memory consistency model of the emerging interconnects. The performance evaluation with big data workloads proves FS^2's high scalability over the vanilla emerging interconnects: up to 2.64× and 4.44× over CXL- and CCIX-like baselines, respectively.

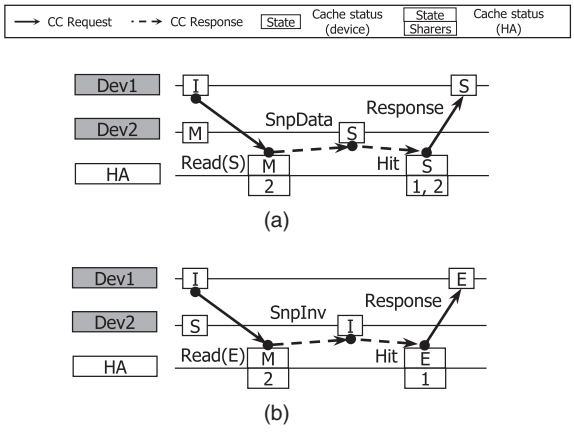


FIGURE 2. Timeline and messages of basic ccNUMA operations. (a) Read shared. (b) Write. cc: cache coherent.

BACKGROUND

The most distinguishing feature of cc nonuniform memory access (ccNUMA) interconnects (e.g., CXL[1] and CCIX)[2] is *cache coherence with a shared address space* for decentralized devices (e.g., GPUs), allowing devices to have their own private cache for a global address space.

ccNUMA interconnects support a directory-based MESI cache coherence protocol with the help of *home agents* (HAs)[a] and *cc messages*. Basically, HAs work as directories that record device lists (i.e., *sharers*) that have the latest value for each cache line. If HAs don't have an entry for the requested cache line, they broadcast the cc message to all devices, which may incur unnecessary traffics for unrelated devices.

ccNUMA interconnects handle two key operations: read and write, based on cache line exclusiveness (Figure 2). A read operation fetches the latest cache-line value without invalidating its sharers. The requesting device (Dev1) sends a read request to the HA. Then, the HA sends a cc message (SnpData) to the sharers (e.g., Dev2). Sharers respond with Hit or Miss and update the cache-line state to shared (S) if applicable. Once the responses are collected, the HA provides Dev1 with the latest data and the shared state. In a write operation, the cache line is granted exclusive ownership, invalidating it for other sharers. Dev1 sends a write request to the HA. Then, the HA broadcasts an SnpInv message to sharers, instructing them to invalidate the cache line. After acknowledgments (hit or miss) are received, the HA sends the latest data and updates the state to exclusive (E) for Dev1.

In the following sections, we analyze the scalability of ccNUMA systems using a deep learning recommendation model [(DLRM),[3] which can benefit from ccNUMA interconnects. It has three phases

[a]HAs can be implemented as either 1) a centralized directory or 2) a distributed setup across devices. Because both rely on a single synchronization point for specific addresses, we focus on 1) while presenting results for both approaches for simplicity.

in the Peripheral Component Interconnect Express (PCIe)-based conventional multi-GPU systems: 1) table lookup by each device (mostly memory access), 2) data gathering from all devices (sync, mostly memory access), and 3) postprocessing on the gathered data (multilayer perceptrons, mostly computation). On the other hand, ccNUMA systems don't require step 2 (each device can directly access other devices' memory) thanks to its shared address space.

MOTIVATION

We adopt PCIe's tree-based scaling methodology to design a baseline large-scale system with ccNUMA interconnects as they are built on top of PCIe to ensure compatibility. This approach scales effectively to rack-level deployments, aligning with modern systems used in industry and research.[4,5] Also, the CXL 3.1 specification[1] outlines a vision for rack-scale scalability, utilizing CXL switches. Expansion units, offered by several companies, integrate switches, device slots, and power without CPUs for modular scalability. These units are connected through high-port expansion switches and external cables that enable large-scale deployments.

However, we observe that the scaling methodologies for PCIe are not suitable for ccNUMA interconnects due to their unique feature: *cache coherence for a shared address space* [Figure 1(a)]. Specifically, we observe that the baseline's performance quickly saturates and is far from its ideal (e.g., 4.58× lower with 256 devices) with DLRM. Our latency breakdown reveals that poor scalability stems from *the memory bottleneck* caused by table lookups (accounting for 97.7% of the latency for 64 devices). Our in-depth analysis shows that 1) *loss of data reuse opportunity* and (2) *high cache-miss penalty* are the root causes of memory bottleneck.

With a large number of devices, the system suffers from a severe *loss of data reuse opportunities*. As memory accesses are spread across multiple devices, temporal locality is also distributed, reducing the likelihood of retaining specific memory addresses in its private cache. In Figure 1(b), our profiling with real-world workloads shows that the device cache-hit ratio significantly decreases as device counts increase (e.g., 42.7% → 4.8% in Yahoo! Cloud Serving Benchmark workload-d (YCSBd)). This loss stems from *the nature of the private cache* for a shared address space. This drop is due primarily to reduced cache visibility (= device cache size/system cache size) and the fragmented distribution of data across multiple caches.

The *high cache-miss penalty* is another reason for the bottleneck with a large number of devices [Figure 1(c)]. As we increase the device counts, *the cache-miss penalty also increases;* for instance, the cache-miss penalty is ~14× larger than local memory access with 64 devices. We identify that the high penalty comes from *long, critical paths between a device and HAs*. Because a device should request a cache coherence resolving to HAs (having the cache directory), each cache miss traverses the path between the HAs and the device "twice" (one for request, the other for response). This path comprises several interconnect switches whose number increases with more devices because of limited switch radix; each hop between two switches adds additional packet processing overheads. Moreover, with larger big data workloads and faster devices, this problem becomes more severe.

CHALLENGES

Here, we discuss three challenges to enable scalable ccNUMA interconnects.

Marginal Benefits From Device Scaling

As the loss of data reuse opportunities and high cache-miss penalty are the main causes, a naïve solution might be placing bigger device caches. However, this device-side scaling is impractical because devices already consume a large die space and have reached their maximum. Therefore, it is difficult to assign more die space to the caches without sacrificing computation performance due to physical limitations. In addition, this device scaling returns marginal benefits to the cache-hit ratios (private-1× versus private-4× in Figure 3).

With shared caches, we expect a dramatic increase in the cache-hit ratio because the system can utilize them much more efficiently. For example, adding the same amount of shared caches increases the hit ratio by more than 40% (private-1× +shared-1× versus private-2×).

Complex Standardization

We may consider adding new rules to the ccNUMA specification and changing hardware components

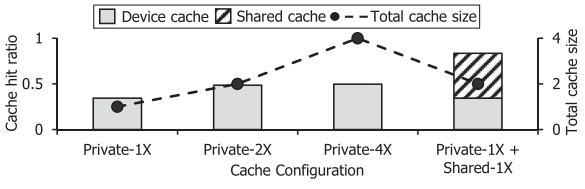


FIGURE 3. Impact of cache scaling. We use 64 devices, and the cache's size is normalized to the single device (1 × = 6 MB per device).

to support shared caches. However, unlike CPU-only cache-coherent interconnects, designing ccNUMA interconnects involves multiple vendors spanning the entire computer system. Therefore, even adding a single rule requires not only heavy development costs but also complex standardization procedures among different parties whose interests may conflict.

Workload-Dependent Performance

Cache-miss penalties depend on hop counts (the path between a device and HAs) and the overhead per hop, which is inversely proportional to per-device bandwidth. For example, comparing tree and all-to-all topologies with the same number of devices, the tree topology exhibits higher maximum hop counts but better worst-case bandwidth.

The tradeoff between hop counts and overhead depends on workload access patterns. Although tree topologies work well for legacy platforms with infrequent root access, they can be suboptimal for ccNUMA interconnects that frequently access memory. Therefore, a fixed topology cannot provide optimal performance for large-scale ccNUMA interconnects.

KEY SCHEMES OF FS^2

To scale ccNUMA interconnects, we propose FS^2, a novel switch-assisted cc system.

Scheme 1: FS^2-Switch

The first key scheme is replacing (naïve) switches with smart FS^2-switches that have shared caches. FS^2-switch provides intraswitch shared cache for its children, instead of expensive device-side static random-access memory (SRAM); the switch cache not only exploits the locality between devices but also reduces the cache-miss penalty by reducing the critical path. To preserve specification transparency, FS^2-switch only communicates with other components using native cc messages.

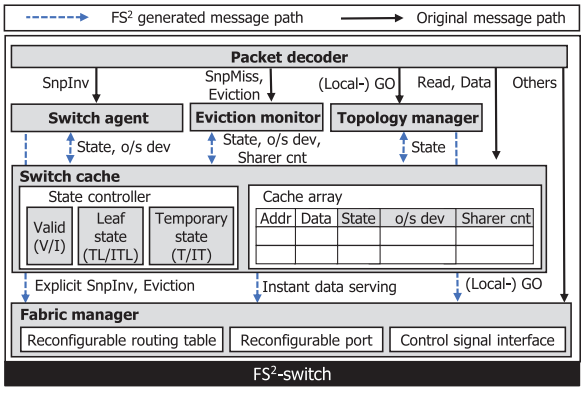


FIGURE 4. Overview of FS^2-switch. addr: address. o/s dev: outstanding device; GO: global observation; V/I: valid/invalid; TL/ITL: temporary for leaf/invalid temporary for leaf; T/IT: temporary/invalid temporary; cnt: count.

Upon receiving cc messages from a device (=device-cache miss), FS^2-switch looks up its cache for the requested address and forwards the cc messages to the HAs in parallel. In case of a hit, FS^2-switch instantly replies to the requesting device with data; then it internally follows FS^2-mechanism to resolve the cache coherence as if it is the requesting device to guarantee the coherent view of HAs, switches, and devices.

To enable FS^2-switch, we add four hardware modules on typical ccNUMA switches (Figure 4). A *packet decoder* interprets cc messages and activates a *shared agent* (for SnpInv) or an *eviction monitor* (for SnpMiss and Eviction) to maintain the cache coherence protocol unchanged. Then, the activated module accesses *the cache array* whose entry includes address, data, state, outstanding devices (o/s devs) to record devices in an inconsistent state, and share_cnt to store the number of sharers under this switch. We develop FS^2-mechanism to guarantee the safety of FS^2-switch.

Scheme 2: FS^2-Topology

To provide different optimal topologies for each workload, we extend a flexible switch from the PCIe domain.[7] Although typical PCIe switches only support a tree topology, such flexible switches can form an arbitrary topology through manual configuration through fabric manager (Figure 4) at the setup stage. To change the topology, the system operators require two simple tasks. They first choose a target topology and adjust 1) reconfigurable ports (=bandwidth per port) and 2) the physical connection among switches using external cables (=topology) so that the connection forms a target topology. Then, they provide new routing tables to the switches through an administration interface, which makes each switch know each position in the topology.

We ensure unique paths for cc messages between a device and the HAs through two rules. First, the configurable routing table and reconfigurable port enable port-based routing for specific cc messages, as adopted by CXL.[1] Second, caches follow the noninclusive and nonexclusive policy to simplify our architecture and enable various network topologies by removing dependencies between switches. Additionally, this design choice helps minimize cache space overhead in upper-level switches by balancing between inclusive and exclusive caching while leveraging data locality across devices.

Scheme 3: FS^2-Scheduling

The existing scheduling schemes are designed without considering caches in the switches. Therefore, there is room to more efficiently utilize the locality of the proposed FS^2-switch. To address this, we propose FS^2-scheduling, an example of a locality-aware software scheduling methodology, to leverage both device and switch caches. It can be generally applied to any workload because it does not require prior knowledge. Also, given that FS^2-scheduling is software assisted (orthogonal to FS^2-switch), it inherently offers flexibility, allowing it to be fine-tuned according to the characteristics of target workloads for better performance. At offline, FS^2-scheduling statically assigns an exclusive range of global address space to each device. At runtime, for each user request, the scheduler examines the addresses of the request and makes a score for each device. Specifically, it increases the score of the devices whose address overlaps with the addresses of requests. Next, for all switches located between the highest-scored device and the HA, we iteratively give an additional score to all successor devices under the switch to exploit locality through switch caches. Finally, FS^2-scheduling decides the target device based on its score and load by checking the current load of the highest-scored device. If the selected device is already fully utilized, FS^2-scheduling selects the next highest-score device as a candidate.

FS^2-MECHANISM

FS^2-mechanism (Table 1) serves as the core of FS^2, providing safe protocol translation layer between unmodified ccNUMA devices/HAs and FS^2-switches. It resolves three primary causes of inconsistent views between HAs and switches during switch-cache hits.

Switch Agent

FS^2-mechanism addresses the inconsistencies in write-after-read scenarios (Figure 5). Here, Dev1 and Dev2 are connected to the same switch (S1) and S1 caches data of address A, while Dev2 stores data from A. When S1 processes a read request from Dev1 using cached data, it acts as the HA for Dev1 and manages subsequent actions as if it were the device itself. Before the forwarded read request from S1 reaches the HA, another device under a different switch (Dev3) may issue a write request to A. This write request is forwarded to the HA without passing S1. Then, the HA broadcasts cache invalidation to the sharers (Dev2) through S1. Here, an inconsistency arises: the memory orders differ between S1 and HAs. At S1, the sequence is read → write, whereas at HAs, it is write → read. This discrepancy leads HAs to skip sending an invalidation message to Dev1 as it is not marked as a sharer at the time of the write. Consequently, Dev1 retains stale data, resulting in functional failures and noncoherent memory states.

The SA of FS^2-switch addresses the above violation by introducing a new state: temporary (T). After S1 serves a read to Dev1 using its cache, the state of the cache line moves to T and `outstanding (o/s) dev` field of the cache line records Dev1. When S1 gets a response for the read from the HAs, the state transits from T to valid (V). Here, we define a temporary section as a period where a cache line is in T state and mark the served cache line as invisible from the perspective of the HAs. If S1 receives a write for a cache line in a temporary section, the SA in S1 generates explicit invalidation messages to devices recorded in `o/s dev` to invalidate the dirty data in Dev1.

Eviction Monitor

Eviction monitor (Figure 4) addresses inconsistencies caused by read-evict-write scenarios. To prevent issues stemming from eviction messages that may reorder requests due to ccNUMA interconnect properties, the SA maintains a `sharer_cnt` field for each cache line to track device sharers under the switch. During the temporary section, the counter is decremented for each eviction message. When it reaches zero, the switch generates explicit invalidation messages to devices recorded in `o/s dev`, ensuring coherence. This mechanism also resolves inconsistencies caused by silent evictions in a similar manner, allowing the switch to act preemptively to maintain functional correctness.

Topology Manager

FS^2-mechanism uses a topology manager to prevent functional failures in multilevel switches caused by duplicate global observation (GO) responses. Note that GO is a response message, generated by the HA, indicating when data are guaranteed to be observed consistently by all agents, ensuring cache coherence for both read and write operations. In multilevel switch systems, devices may receive multiple GOs from upper-level switches and HAs, even if a switch (S1) serves the read request.

To resolve this, the topology manager introduces the temporary-at-leaf (TL) state for leaf switches (directly connected to devices). After serving GO and data to a device, leaf switch transitions the cache-line state from V to TL, blocking further GOs to that device.

For coherence, internal switches use a new Local-GO message, leveraging reserved bits in response packets. Internal switches generate Local-GOs for cache hits

or the first Local-GO from upper switches, while leaf switches convert the first Local-GO into a standard GO for devices. This ensures compatibility with unmodified devices and coherence across the hierarchy.

Validation

We validate FS^2-mechanism's functionality through formal and theoretical analyses. We used the Murphi model checker to formally validate FS^2. Because the scale of the system is limited due to the exponential growth of states with the number of devices, we extensively validate our scheme using a representative configuration of eight devices, incorporating all key elements of our design (e.g., hierarchical switch and various topologies).

Theoretically, we confirm that FS^2 adheres to the memory model defined in the CXL specification. This model, based on relaxed memory consistency, enforces two key constraints:[1] 1) ordering constraint: HAs are not required to preserve the order of device requests and 2) latest constraint: each read must retrieve the latest written data, guaranteed by the HA. FS^2 operates identically to standard CXL except in a specific scenario: 1) device 1 is instantly served by a switch and 2) before HAs are notified, they process a request from Device 2. In this case, device 1 and the HA observe reversed memory orders (device 1 → device 2 versus device 2 → device 1). Nonetheless, FS^2 remains compliant with the memory model because 1) both orders

TABLE 1. State transition table of FS^2-mechanism (Internal for internal switches and Leaf for leaf switches).

State	Definition	Event	Action	Transition (Condition)
V	Coherent cache line	Read	Forward data and Local-GO (if internal) or GO (if leaf) to the requester, forward read to the HA	T (if internal) or TL (if leaf)
< >	< >	Eviction (or SnpMiss)	Sharer count− −	I (if sharer count == 0)
< >	< >	SnpInv (or Write)	—	I
I	Invalid cache line	Read	Reserve a cache line, forward read to the HA	IT (if internal) or ITL (if leaf)
T	Cache line in temporary section (switch cache hit)	Read (or ##)	Stall read (or write) until GO message arrives	—
< >	< >	Eviction (or SnpMiss)	Sharer count− −, notify *Eviction Monitor* for explicit invalidation if sharer count == 0	—
< >	< >	SnpInv	Notify *switch agent* for explicit invalidation	—
< >	< >	GO + data	Forward data and GO, sharer count++	V
< >	< >	Local-GO + data	Block data and Local-GO	—
TL	T-state for leaf	GO + data	Block data and GO, sharer count++	V
< >	< >	Other events	Same as T-state's actions	Same as T-state's transitions
IT	Cache line in temporary section (Switch cache miss)	Read	Stall read until GO message arrives	—
< >	< >	GO (or Local-GO) + data	Forward data and GO (or Local-GO), sharer count++ if GO has arrived	V (if GO) or T (if Local-GO)
ITL	IT-state for leaf	Local-GO + data	Convert Local-GO to GO, forward data and GO	TL
< >	< >	Other events	Same as IT-state's actions	Same as IT-state's transitions

ITL: invalid temporary for leaf.

are permissible under the ordering constraint, and 2) explicit invalidations ensure that device 1 holds the latest data, satisfying the latest constraint. Thus, FS^2 aligns with CXL's consistency model.

EVALUATION

Considering the early development stage of ccNUMA interconnects, simulation-based analysis is crucial. Originally, we extended the cycle-accurate MGPUSim[9] to support ccNUMA protocols and FS^2, leveraging its high accuracy. However, MGPUSim's extreme runtime (e.g., more than six years for a single workload on large-scale systems) made it impractical for comprehensive evaluations. To address this, we developed a full-stack, trace-driven simulator and validated the accuracy (i.e., performance improvement of the proposed schemes) against MGPUSim with an error rate of ≤13%, which is lower than other simulators (25% on average).[12]

Software

Our simulator models real big data workloads using an multiserver Markovian queueing model with Poisson arrivals and exponential service times. The requests are queued, and a scheduler (e.g., PyTorch or FS^2-scheduling) assigns them to devices for execution.

Interconnection

We implemented major ccNUMA interconnect protocols, supporting CXL[1] and CCIX[2] for single-point and distributed HAs, respectively. To simplify deployment, all switches use uniform designs, with parameters based on sensitivity tests: 24-MB switch cache, 150-ns switch latency, and 75-ns interconnect speed.

Devices

We modeled the system with 64 V100-based devices with L1/L2 caches, HBM2 memory, and parallel execution units. Each device provides a coherent 6-MB, 16-way cache with a 7.7-ns lookup and shares its local dynamic random-access memory in a coherent address space.

Configurations

Our simulator provides various configuration options to simulate FS^2 in various scenarios. Options include device/switch cache size, number of devices, topology, and interconnect protocols.

Workloads

We evaluated two big data workloads: recommendation system tasks (DLRM RMC2)[3] using Terabyte[10] datasets for online inference (batch size 32) and offline training (batch size 8192), and GPU-accelerated database operations[6] using Redis with YCSB.[8] Out of six YCSB scenarios, we focused on three representative cases due to similar behaviors: 50% read and 50% update, 95% read and 5% update, and 95% read-latest and 5% write. We generated traces on a real machine by matching CPU cores to devices using multithreaded YCSB clients and fed them into the simulator.

FS^2 boosts the performance of the big data workloads for both single-point and distributed HAs [Figure 6(a) and (b)]. On average, FS^2 achieves a 1.934× speedup for a single-point HA and a 3.66× speedup for distributed HAs over their vanilla designs. All schemes of FS^2 contribute to such improvement by exploiting locality and reducing the critical path between devices and HAs. For FS2-topology, we show its potential through the best topology-workload pair. Specifically, FS^2-switch, FS^2-topology, and FS^2-scheduling provide speedups of 1.36, 1.33, and 1.21× for single-point HAs and 1.57, 2.29 and 1.47× for distributed HAs, respectively. We also notice that FS^2 with distributed HAs provides a greater benefit than FS^2 with the single-point HA because distributed HAs show longer worst-case hop counts than single-point HAs.

The benefits of FS^2 become greater with more devices (Figure 7). We emphasize that FS^2-topology is effective for scalability; by alleviating the topological limitations of the tree using the best-performing topology for each workload, FS^2 shows scalability closer to linear scaling for high-locality and read-intensive workloads.

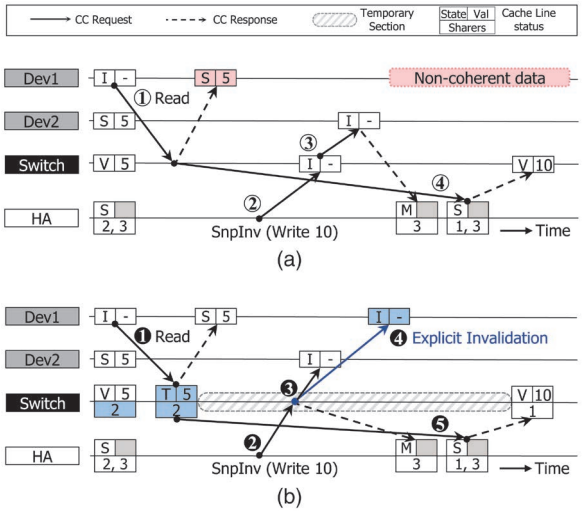


FIGURE 5. Explicit invalidation for write-read violation. (a) Timeline of read-write violation. (b) Timeline of explicit invalidation from the switch agent.

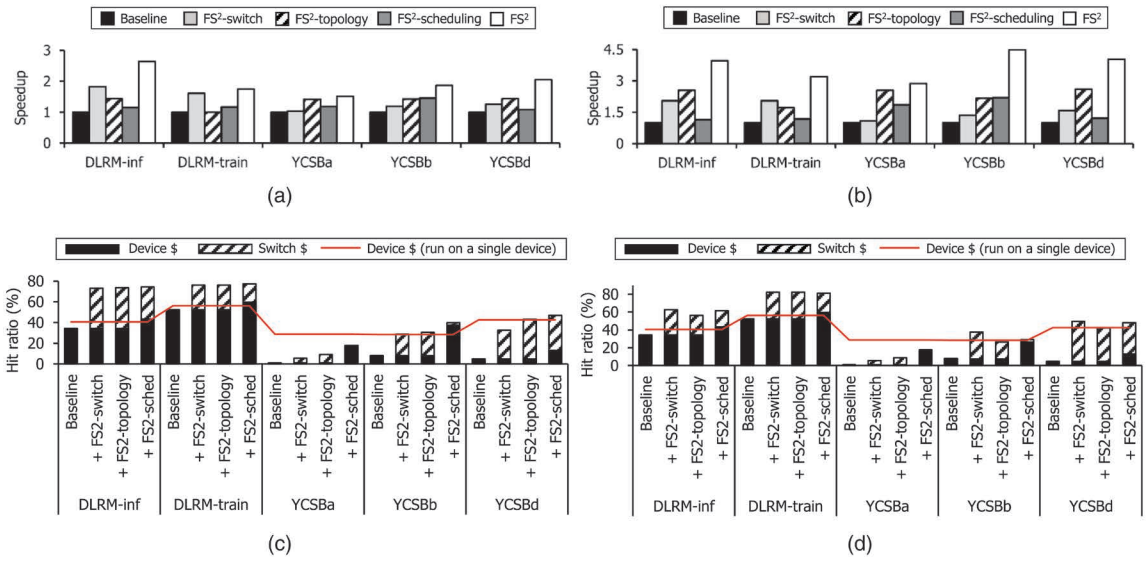


FIGURE 6. Speedup and cache-hit ratio of FS^2 for two HA designs. (a) Speedup (single-point HA). (b) Speedup (distributed HA). (c) Cache-hit ratio (single-point HA). (d) Cache-hit ratio (distributed HA).

We analyze the cache-hit ratio to elaborate on the performance improvements [Figure 6(c) and 6(d)]. The baseline fails to fully utilize workload locality because private cache is distributed across multiple devices, whereas FS^2 achieves significantly higher cache-hit ratios. Also, we observed the impact of how each scheme varies depending on workload patterns and HA designs. For example, FS^2-scheduling accelerates producer–consumer workloads like YCSBa/b by enhancing cache efficiency through closer placement of producers and consumers. By contrast, FS^2-switch boosts YCSBd, which follows the latest distribution pattern where data popularity decreases over time. This workload benefits more from a larger system cache due to its access to noncontiguous data. Additionally, FS^2-topology offers advantages for distributed HAs as it supports various topologies (e.g., 3 D-mesh and torus), unlike single-point HAs (limited to star or tree).

We demonstrate the feasibility of FS^2-switch using CACTI 6.0 for a switch's SRAM cache in 28-nm technology. The power and area overhead is 3.35 W and 89.48 mm^2, respectively, significantly smaller than off-the-shelf PCIe switches (18.6 W and 1225 mm^2, respectively)].[11]

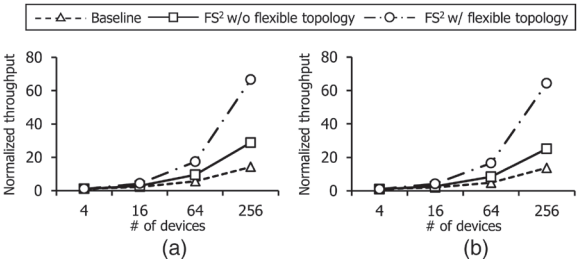


FIGURE 7. Scalability test (normalized to four devices). (a) DLRM-inf. (b) YCSBb.

CONCLUSION

We proposed FS^2, a scalable cache-coherent system by addressing the memory bottleneck of current ccNUMA interconnects. FS^2 adopts cache-augmented flexible switches with cache-aware scheduling while preserving all semantics of the ccNUMA interconnects, providing higher scalability. We believe that our ideas can be a useful guideline for next-version ccNUMA interconnects.

ACKNOWLEDGMENTS

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korean Government (MSIT) (RS-2024-00402898, RS-2025-02217106, RS-2024-00395134). We also appreciate the support from the Automation and Systems Research Institute (ASRI) and Interuniversity Semiconductor Research Center (ISRC) at Seoul National University. Heetaek Jeong and Kanghyun Choi contributed equally to this work.

REFERENCES

1. "CXL 3.1 specification." Compute Express Link. Accessed: Nov. 10, 2024. [Online]. Available: https://computeexpresslink.org/cxl-specification/
2. CCIX Admin, "CCIX base specification revision 2.0 version 1.0," *CCIX*, Oct. 18, 2022. [Online]. Available: https://computeexpresslink.org/resource/ccix-specification-archive/
3. U. Gupta et al., "The architectural implications of Facebook's DNN-based personalized recommendation," in *Proc. IEEE Int. Symp. High Perform. Comput. Archit. (HPCA)*, 2020, pp. 488–501, doi: 10.1109/HPCA47549.2020.00047.
4. P. Park, H. Jeong, and J. Kim, "TrainBox: An extreme-scale neural network training server architecture by systematically balancing operations," in *Proc. 53rd Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2020, pp. 825–838, doi: 10.1109/MICRO50266.2020.00072.
5. "Liqid SmartStack: Deliver unmatched GPU performance and utilization," Liqid, Broomfield, CO, USA, 2023. [Online]. Available: https://cdi.liqid.com/hubfs/Liqid%20SmartStack_090723.pdf
6. B. He and J. Xu Yu, "High-throughput transaction executions on graphics processors," *Proc. VLDB Endowment*, vol. 4, no. 5, pp. 314–325, 2011.
7. "PEX89000 series product brief." Broadcom. Accessed: Nov. 10, 2024. [Online]. Available: https://docs.broadcom.com/doc/PEX89000-Managed-PCI-Express-5.0-Switches
8. 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.
9. Y. Sun et al., "MGPUSim: Enabling multi-GPU performance modeling and optimization," in *Proc. 46th Int. Symp. Comput. Archit.*, 2019, pp. 197–209.
10. M. Brugidou, "Powering real-time decisions with a reduced footprint," *Criteo Tech Blog*, May 13, 2025. [Online]. Available: https://labs.criteo.com/2013/12/download-terabyte-click-logs/
11. "Product brief PEX8796." Broadcom. Accessed: Nov. 10, 2024. [Online]. Available: https://docs.broadcom.com/doc/12351860
12. M. Khairy, and Z. Shen, and T. M. Aamodt, and T. G. Rogers, "Accel-Sim: An extensible simulation framework for validated GPU modeling," in *Proc. ACM/IEEE 47th Int. Symp. Comput. Archit. (ISCA)*, 2020, pp. 473–486, doi: 10.1109/ISCA45697.2020.00047.

HEETAEK JEONG is a system architect at MangoBoost, Seoul, 08806, South Korea, and a Ph.D. candidate in electrical and computer engineering (ECE) at Seoul National University, Seoul, 08826, South Korea. His research interest is computer architecture. Jeong received his bachelor's degree in electrical and computer engineering (ECE) from Seoul National University. Contact him at heetaek@snu.ac.kr.

KANGHYUN CHOI is a system architect at MangoBoost, Seoul, 08806, South Korea, and a Ph.D. candidate in electrical and computer engineering (ECE) at Seoul National University, Seoul, 08826, South Korea. His research interest is computer architecture. Choi received his bachelor's degree in electrical and computer engineering (ECE) from Seoul National University. Contact him at kanghyun_choi@snu.ac.kr.

HAMIN JANG is a system architect at MangoBoost, Seoul, 08806, South Korea, and a Ph.D. candidate in electrical and computer engineering (ECE) at Seoul National University, Seoul, 08826, South Korea. His research interest is computer architecture. Jang received his bachelor's degree in electrical and computer engineering (ECE) from Seoul National University. Contact him at hamin.jang@snu.ac.kr.

DONGUP KWON is the CTO at MangoBoost, Seoul, 08806, South Korea. Kwon received his Ph.D. degree in electrical and computer engineering from Seoul National University, Seoul, 08826, South Korea. His research interest is computer architecture. Contact him at dongup.kwon@mangoboost.io.

EUNJIN BAEK is a system architect at MangoBoost, Seoul, 08806, South Korea. Baek received her Ph.D. degree in electrical and computer engineering from Seoul National University, Seoul, 08826, South Korea. Her research interest is computer architecture. Contact her at eunjin.baek@mangoboost.io.

PYEONGSU PARK is a research scientist at Meta, CA, 94089, USA. Park received his Ph.D. degree in electrical and computer engineering from Seoul National University, Seoul, 08826, South Korea. His research interest is system architecture for next-generation datacenter. Contact him at pyeongsu@meta.com.

JANGWOO KIM is the CEO at MangoBoost, Seoul, 08806, South Korea, and a professor of electrical and computer engineering at Seoul National University, Seoul, 08826, South Korea. Her research interest is computer architecture. Kim received his Ph.D. degree in computer architecture from Carnegie Mellon University. He is the corresponding author of this article. He is a Member of IEEE. Contact him at jangwoo@snu.ac.kr.

THEME ARTICLE: CACHE COHERENT INTERCONNECTS AND RESOURCE DISAGGREGATION TECHNIQUES

Compute-Enabled CXL Memory Expansion for Efficient Retrieval Augmented Generation

Derrick Quinn, Neel Patel, and Mohammad Alian, *Cornell University, Ithaca, NY, 14850, USA*

Conventional processing near-memory architectures often strike a tradeoff between memory capacity and memory bandwidth, leading to high initial data movement or high capital costs due to memory stranding. In this work, we introduce compute-enabled memory expansion through Compute Express Link as a solution for the widespread adoption of processing near-memory at scale. We discuss the Intelligent Knowledge Store (IKS), which is fundamentally a memory expander with lightweight near-memory accelerators that leverage high internal memory bandwidth to accelerate dense retrieval, a key component of retrieval-augmented generation. IKS disaggregates its internal memory capacity and supports both spatial and temporal multitenancy. It significantly accelerates high-quality dense retrieval while enabling multitenancy with modest memory access interference.

Processing near memory (PNM) reduces data movement overhead by placing computation close to memory. Early PNM products have shown promise, but there is no consensus on how to balance tradeoffs in capacity, bandwidth, and complexity. Existing PNM architectures often sacrifice memory capacity to achieve higher bandwidth, causing large portions of dynamic random-access memory (DRAM) to remain underutilized and "stranded" when operating in acceleration mode. Given the capital cost and environmental impact of DRAM manufacturing, it is essential to efficiently share and utilize memory resources.

Memory disaggregation, where multiple components share pooled memory, can help address memory stranding. However, current PNM products lack the flexibility to share their internal DRAM between accelerators and CPUs. The emergence of Compute Express Link (CXL) presents an opportunity to unify PNM and memory disaggregation—two seemingly opposing paradigms.

Towards this vision, we recently proposed *Intelligent Knowledge Store* (IKS), a compute-enabled CXL memory expander that accelerates dense retrieval, crucial in retrieval-augmented generation (RAG) applications.[9] Unlike traditional PNM architectures, IKS shares its physical address space with the host CPU, allowing unused internal DRAM to serve other applications and address memory stranding. As illustrated in Figure 1, IKS extends the PNM design space, balancing high internal bandwidth for specialized tasks with efficient memory sharing. More specifically, as shown in Figure 2, IKS implements eight Low-Power Double Data Rate (LPDDR5X) packages, directly connected to and controlled by eight near-memory accelerator (NMA) chips, providing a total of 512 GB of DRAM capacity and an aggregate internal bandwidth of 1 TB/s. The NMAs function either as memory controllers for host CPU accesses or as search engines over the embedding vectors in the LPDDR5X packages. Leveraging both `CXL.cache` and `CXL.mem`, IKS implements a seamless interface between the CPU and NMAs for efficient offload.

Digital Object Identifier 10.1109/MM.2025.3575280
Date of publication 3 June 2025; date of current version 24 December 2025.

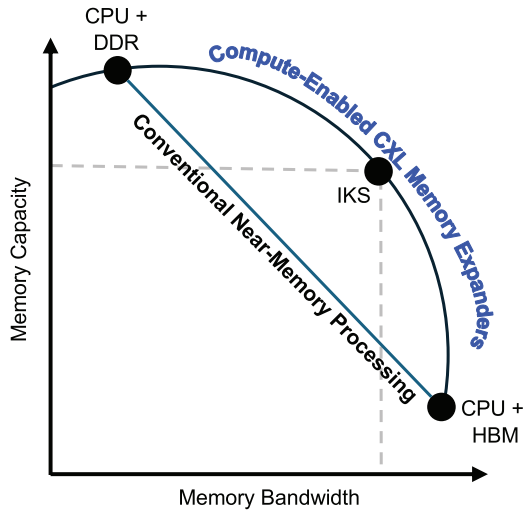


FIGURE 1. Spectrum of PNM.

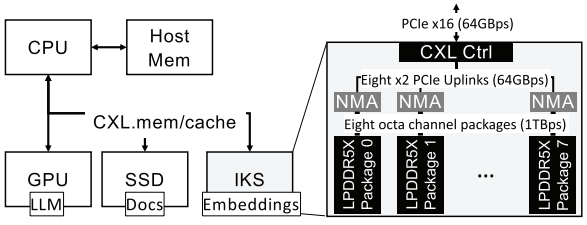


FIGURE 2. A server enhanced with IKS for RAG applications. IKS leverages internal bandwidth and memory parallelism of a CXL memory expander to accelerate dense retrieval.

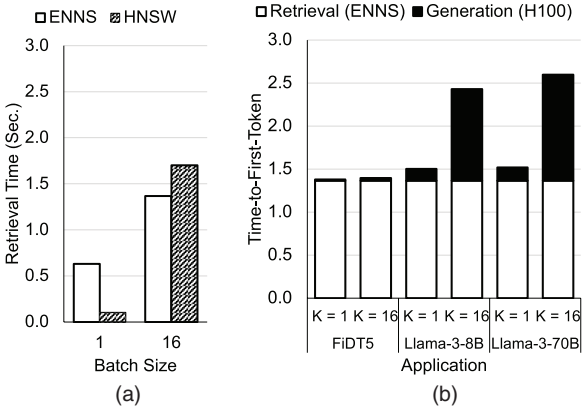


FIGURE 3. Comparison of ENNS and HNSW (Recall@32 = 0.95) retrieval, and TTFT breakdown for three representative RAG applications based on *Wikipedia* corpus. For generation, Llama-3-70B uses 8x NVIDIA H100 GPUs, while FiDT5 and Llama-3-8B use 1. (a) ENNS versus HNSW. (b) TTFT breakdown.

Our contributions are as follows:

- We present the opportunity that PNM offers for enabling algorithmically simple, accurate, and general exact nearest neighbor search (ENNS) for dense retrieval in RAG systems.
- We discuss IKS, a CXL-based memory expander that cost-effectively accelerates ENNS while minimizing memory stranding.
- We explore the interface options that CXL offers to tightly integrate near-memory accelerators into the rest of the system.
- We show that IKS effectively disaggregates its internal memory with minimal performance interference.
- We show that accelerating ENNS with IKS achieves up to 6.7× higher throughput for representative RAG applications compared to a CPU baseline using approximate nearest neighbor search.

RETRIEVAL IN FUTURE AI SYSTEMS

Modern artificial intelligence (AI) services increasingly integrate large language models (LLMs) with retrieval systems, enabling RAG to generate more accurate and factually grounded output. Underpinning these systems are dense retrieval methods that map retrievable items into high-dimensional vectors. Several vector search strategies exist, including ENNS as well as approximate nearest neighbor search (ANNS) methods.

ANNS algorithms accelerate retrieval by indexing or preclustering the document vectors to reduce the search space. However, this speed gain comes at a cost, as ANNS introduces algorithmic complexity and dataset-specific behavior, resulting in a nontrivial tradeoff between accuracy and retrieval latency. In line with recent works,[10] we observe that a RAG system with low retrieval accuracy can lead to an increase in the number of items (i.e., context) that need to be retrieved and sent to the downstream LLM for high-quality generation. Such an increase in context size can significantly increase the generation time and negate the end-to-end benefits of using a faster ANNS compared to a slower but exact ENNS. Moreover, in RAG serving systems in data centers, where batching is common due to many concurrent users sending inference requests, ANNS can lose its competitive advantage compared to ENNS. As shown in Figure 3(a), when multiple queries are processed together, ENNS can reuse corpus vectors for every query in the batch,

enabling it to bridge the efficiency gap with a high-quality Hierarchical Navigable Small World (HNSW).

However, using ENNS can significantly increase the time-to-first-token (TTFT), reducing the responsiveness of the RAG application. As shown in Figure 3(b), ENNS retrieval often dominates TTFT.

Combining these observations, there is a strong incentive to accelerate high-quality retrieval. ENNS's predictable, brute-force search pattern avoids the irregular memory accesses and complex indexing structures of graph-based ANNS, making it inherently easier to accelerate in hardware. As a result, accelerating ENNS can deliver significant gains in throughput and responsiveness for next-generation RAG pipelines through simple and flexible hardware. Nevertheless, both graph-based and clustering-based ANNS algorithms can benefit from an ENNS accelerator because similarity score evaluation is a core kernel shared between ENNS and all ANNS algorithms.

INTELLIGENT KNOWLEDGE STORE TO ACCELERATE ENNS

Unlike a monolithic accelerator, IKS implements eight lightweight NMAs, each paired with an LPDDR5X package. This *scale-out* approach reduces off-chip and on-chip data movement by using smaller NMA chips instead of a large monolithic chip that can provision enough chip shoreline to connect to eight LPDDR5X packages with 64 memory channels. Each LPDDR5X package provides 64 GB of DRAM capacity and eight memory channels, delivering a total of 136 GB/s of memory bandwidth. In contrast, achieving 512 GB of capacity using high-capacity ×4 DDR5 devices would require 32 devices and a large CXL device form factor while yielding only 89.6 GB/s of internal memory bandwidth. The high internal memory bandwidth is essential for IKS's near-memory acceleration of ENNS.

NMA Architecture

Within each NMA, 64 processing engines compute similarity scores between a batch of query vectors and embedding vectors. A column-major data layout maps each dimension of 68 embedding vectors contiguously, allowing efficient, output-stationary dot product operations. This arrangement simplifies data distribution to multiply-accumulate units and leverages batching by reusing embedding vectors for all query vectors within a batch, improving throughput and energy efficiency (Figure 4).

Every *vector dimension* number of clock cycles, each processing engine evaluates 68 similarity scores that need to be inserted into an ordered list maintained in the top-K unit. The insertion is overlapped with the similarity score evaluation of the next 68 vectors during the next *vector dimension* clock cycles. After all of the embedding vectors are fetched from memory and evaluated for similarity, the top-K unit copies the partial top-K list into the output scratchpad. The output scratchpad is what the CPU reads and aggregates (across eight NMAs and possibly across multiple IKS units in case of a multi-IKS setup) to construct the final top-K list.

Offload Model

The IKS address space is shared with the host CPU. The CPU runs the vector database application, which offloads the similarity calculations (i.e., dot-products between the query vectors and embedding vectors) using `iks_search(query)`, an application programming interface that does not require a system call or context switch.

When `iks_search(query)` is called, the CPU initiates a search by passing an *offload context* to IKS. As we discuss in the next section, IKS leverages `CXL.mem` and `CXL.cache` to enable CPU to seamlessly initiate offload and receive offload completion notifications with minimal overhead. The NMAs perform similarity calculations locally, returning top-K candidates that the CPU aggregates.

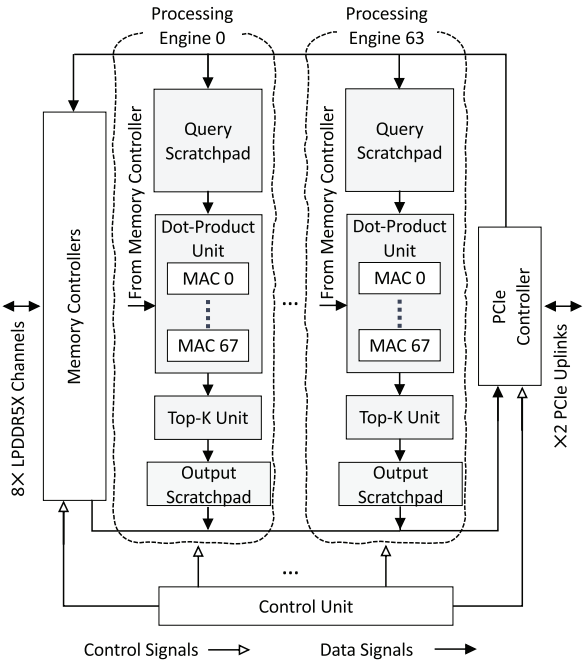


FIGURE 4. Architecture of a single NMA chip. IKS implements eight NMA chips, each next to an LPDDR5X package.

THE CXL-ENABLED INTERFACE

This section outlines current challenges in PNM architectures and describes how CXL-based solutions like IKS address them. We then detail the CPU-IKS interface enabled by CXL protocols.

Challenges in Current PNM Architectures

Cost

Traditional PNM architectures often require custom hardware and software modifications, increasing complexity and deployment costs.

Stranded Resources

NMAs and their dedicated DRAM can remain underutilized, wasting capacity and increasing the total cost of ownership.

Offload Tax

Fine-grain offloads are costly. Initiating and completing offloads involves multiple PCIe round trips, diminishing the potential performance gains.

Cache Coherency

Relying on software to flush CPU cache contents to memory before initiating an offload increases offload tax. This undermines the advantage of moving computation closer to data instead of transferring data closer to compute resources.

Address Translation

Similar to cache coherency, handling virtual addresses in accelerators can be expensive. Without an efficient translation scheme, page faults and translation lookaside buffer (TLB) misses become significant overheads.

Data Movement

Current PNM architectures often require explicit data transfers into NMA's memory, incurring additional offload tax.[3,7]

Opportunity with CXL

IKS leverages CXL features and makes several design decisions to address the challenges listed previously.

Reducing Cost

IKS uses the industry-standard CXL interface for interoperability with different devices and CPUs. In addition, IKS implements a scale-out PNM design to improve NMA chip manufacturing yield and maximize IKS internal memory bandwidth. Mapping IKS memory directly into the host address space allows the CPU to manage the vector database, minimizing expensive code changes. To this end, we ported Meta's Facebook AI Similarity Search (FAISS) to IKS by adding an `IKSIndexFlatIP` library similar to the existing `GpuIndexFlatIP` used for GPU offload.

Reducing Resource Stranding

IKS disaggregates its internal DRAM, allowing it to serve as both an accelerator and a memory expander. This prevents unused capacity from going to waste and reduces overall cost. The NMAs are lightweight, minimizing idle overhead.

Reducing Offload Tax

IKS utilizes the CXL protocol to implement a low-latency interface between the CPU and NMAs, reducing the offload tax. We provide details of this interface later in this section.

Cache Coherency

IKS leverages the fact that embedding vectors are rarely updated and implements a software-managed coherency protocol where the CPU flushes the caches after the rare updates. IKS utilizes the `CXL.cache` protocol only to keep NMA control registers and scratchpads coherent. These structures total less than 200 kB of storage, maintaining a low overhead for hardware cache coherency.

Virtual Memory Translation

With `CXL.mem`, the CPU uses `mmap` to map the entire IKS address space into a contiguous range of the CPU's virtual address space. The CXL controller translates these into device media addresses for NMAs,[2] which operate on a direct-segment model without page faults.[1] In IKS memory, the embedding vectors reside contiguously, simplifying address translation.

Minimizing Data Movement

CXL enables IKS to share the address space between NMA and the host CPU. Thus, NMAs directly access data in place without any data movement required before an offload.

IKS–CPU Interface

In this section, we explore the design space of the CPU-IKS interface and discuss the conventional

interfaces (`Ring Buffer` and `PIO`) as well as those enabled by CXL. Figure 5 illustrates and compares these interfaces.

Ring Buffer

Using a ring buffer that contains descriptors with pointers to the query vectors, an offload can be initialized by the CPU updating the descriptor at the head of the ring buffer and performing a memory-mapped input/output write to a doorbell register on the IKS. The IKS uses a direct memory access (DMA) engine to read the descriptor, retrieve the pointer to the query vectors, and issue another DMA access to read the query vectors from host memory to initiate the offload. Once the offload is completed, the IKS performs a DMA write to place the result into an output buffer and writes a completion descriptor to a doorbell, notifying the CPU of the offload's completion. The CPU must either poll the completion ring or rely on interrupts from the IKS to receive the completion notification. Figure 5(a) illustrates the ring buffer option.

This descriptor ring implementation has the advantage of being general-purpose, supporting multiple concurrent offloads and batching communication between the CPU and IKS for high throughput. However, it suffers from high latency due to multiple PCIe round trips, making it unsuitable for fine-grained offloads.

CXL.mem Interface

The host CPU can use the CXL.mem protocol to directly write into MMIO registers and scratchpad memory (SPM) without going through one level of indirection, which involves the ring buffer and then DMA. In this implementation, the host CPU directly writes the query vectors to the SPM and then writes them into a doorbell register to start the offload. `CXL.mem` protocol uses cacheline flit-based multiplexing at the PCIe physical layer, compared to packet-based multiplexing at the PCIe TLP layer in `CXL.io` and allows multiple in-flight memory accesses between the host and the device.[2]

Although this implementation delivers low offload initialization latency, it requires CPU cycles to perform long-latency uncacheable nontemporal writes into MMIO registers and SPM, or to use temporal writes followed by a cache flush. Once a completion is detected (through polling or interrupt), the CPU can directly load the output of the offload from IKS scratchpad memories. This option is illustrated in Figure 5(b).

CXL.cache Interface

This interface complements the `CXL.mem` interface by having the MMIO registers and SPM spaces coherent through `CXL.cache`, relying on a hardware-provided cache coherency protocol for CPU–NMA communications. This implementation not only simplifies the interface between the CPU and IKS but also reduces the overhead of initiating an offload, receiving notifications, and transferring the result to the CPU.

DISAGGREGATING IKS MEMORY

IKS provides 512 GB of memory accessible to the host over a ×16 PCIe Gen5 link, capable of up to 64 GB/s to the CPU. Internally, IKS interconnects eight LPDDR5X packages, each offering 136 GB/s, for a total internal bandwidth exceeding 1 TB/s—over 16× higher than the external bandwidth. This significant difference between the external and internal available memory bandwidth in IKS creates an opportunity to co-run ENNS and CPU applications with minimal interference. The memory controllers inside the NMAs implements a physical address mapping hash function that maps contiguous 4-kB OS pages to a single memory channel. In this way, each OS page is colored with a channel ID. IKS exposes the page color to the OS, enabling the memory allocator to map different applications to different channels.[6] This memory allocation effectively controls where the future accesses of applications/tenants will be physically routed and enables IKS to support two modes for multitenancy:

Spatial Multitenancy

IKS provides flexibility for fine-grain memory channel partitioning to ensure quality of service and effectively

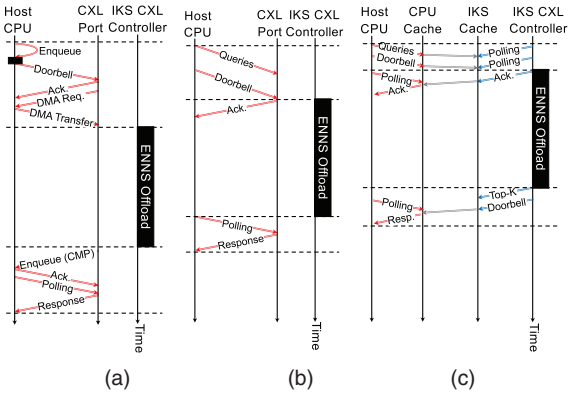


FIGURE 5. Options for IKS–CPU interface. (a) Ring buffer. (b) CXL.mem. (c) CXL.cache.

eliminate interference between different classes of applications. Each LPDDR5X package implements eight channels, each serving 8 GB of DRAM space at a theoretical bandwidth of 17 GB/s. With channel partitioning, CPU and ENNS traffic can be completely isolated (because ENNS lightly utilize the external bandwidth) by mapping them to different channels.

Temporal Multitenancy

Alternatively, IKS can share channels between ENNS and CPU applications. By allowing both applications to dynamically access all channels, IKS can fully leverage its internal bandwidth for ENNS when needed, maximizing ENNS performance. In this scenario, the host and ENNS may contend for bandwidth, but overall throughput and resource utilization are improved.

METHODOLOGY

We emulate IKS on a dual-socket Intel Xeon Gold 6554S server. Each socket has 512 GB of DRAM, and we use one socket to model the IKS device and NMAs, and the other as the host CPU. Although we do not use a real CXL device for our evaluation, we hypothesize that a well-optimized CXL implementation would exhibit a performance profile similar to that of a multisocket CPU. This methodology is widely used in prior works for evaluating the performance of CXL-based systems.[8]

Interfaces

We implement the `Ring Buffer`, `CXL.mem`, and `CXL.cache` interfaces described earlier. For `Ring Buffer`, we leverage the on-chip data streaming accelerator on the remote socket to emulate DMA transfers, descriptor handling, and completion notifications. This approach provides optimistic performance due to Ultra Path Interconnect latency/bandwidth differences compared to PCIe.

For `CXL.mem` and `CXL.cache`, we allocate buffers on the remote socket to emulate query and output scratchpads. We use `MOVDIR64B` instruction for direct writes and `CLDEMOTE` to model coherent cacheline operations. Completion notifications are emulated via remote writes, combined with polling (`CXL.mem`) or cache invalidations (`CXL.cache`).

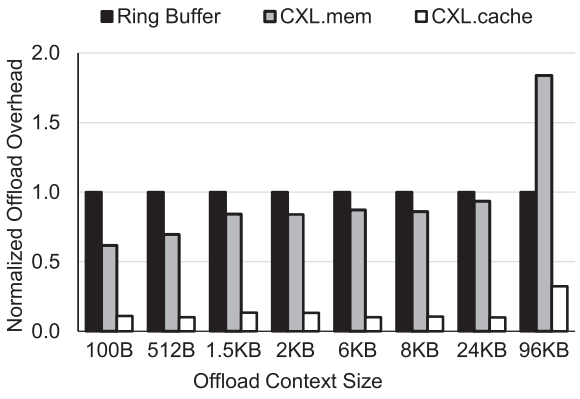


FIGURE 6. Comparing the offload tax (communicating context buffers) for ring buffer, CXL.mem, and CXL.cache

Multitenancy

To study spatial and temporal multitenancy, we use sub-nonuniform memory access clustering to partition channels and emulate different memory bandwidths. Each DDR5-4000 channel stands in for two LPDDR5X channels, matching the ratio of bandwidths and allowing fair comparisons. We throttle the local CPU application's memory bandwidth to not exceed the modeled 8-GB/s uplink between an NMA and CPU, aligning with the IKS design.

WE THROTTLE THE LOCAL CPU APPLICATION'S MEMORY BANDWIDTH TO NOT EXCEED THE MODELED 8-GB/S UPLINK BETWEEN AN NMA AND CPU, ALIGNING WITH THE IKS DESIGN.

RAG Pipeline Evaluation

We follow Karpukhin et al.[5] to construct a Wikipedia-based corpus and fine-tune query embedding model. To evaluate generative models, we use the open-weight 8- and 70-billion parameter and Llama-3 models. Additionally, we follow Izacard and Grave[4] to fine-tune a T5-based generative model based on Google's NQ "train" dataset. From these, we construct three RAG pipelines, labeled Llama-3-8B, Llama-3-70B, and FiDT5. Queries and ground-truth answers are taken from the NQ "dev" dataset. Each query is processed by generating a query embedding, performing retrieval (ENNS or HNSW), and then feeding top-K documents to the generative model. Accuracy for FiDT5 is exact match, while for Llama-3 models we use Rouge-L Recall.

For HNSW, we use FAISS with parameters chosen to maximize performance while maintaining a reasonable graph. The HNSW index is built with $M = 32$,

efConstruction = 128, and runtime efSearch values chosen to balance speed and accuracy.

EVALUATION

We evaluate the IKS–CPU interfaces, multitenancy modes, and overall IKS performance on RAG applications.

IKS–CPU Interface

Figure 6 shows offload overhead for Ring Buffer, CXL.mem, and CXL.cache. At small context sizes, CXL.cache and CXL.mem outperform Ring Buffer due to reduced initialization overhead. For large contexts (e.g., 96 kB), Ring Buffer can leverage DMA for bulk transfers, surpassing CXL.mem. CXL.cache consistently achieves the lowest overhead, improving offload efficiency by up to 85.5% and 86.2% over CXL.mem and Ring Buffer, respectively.

TABLE 1. Comparing Memcached and ENNS performance when running solo on IKS and with spatial multitenancy where half of the LPDDR channels are allocated to each application. ENNS corpus size is 256 GB.

App	Metric	Solo Run	Spatial MT
ENNS	Search Time	313 ms	612 ms
	QPS	51.0 q/s	26.1 q/s
Memcached	P50	111µs	111 µs
	P99	191µs	199 µs
	RPS	421.9k	421.8k

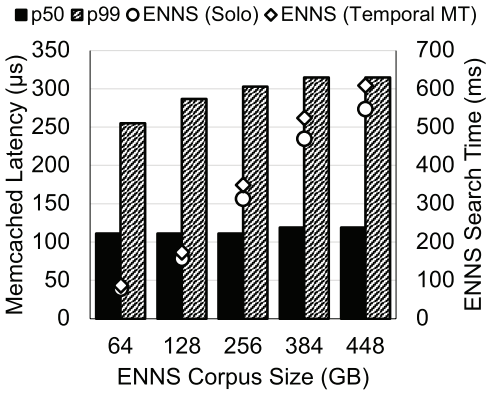


FIGURE 7. Latency of Memcached application and ENNS running on IKS with temporal memory disaggregation. A batch size of 16 is used. To show the impact of the impact of ENNS on Memcached, ENNS QPS is fixed to 32 with varying corpus sizes, affecting internal memory bandwidth utilization for ENNS configs. Solo is when Memcached is not co-running.

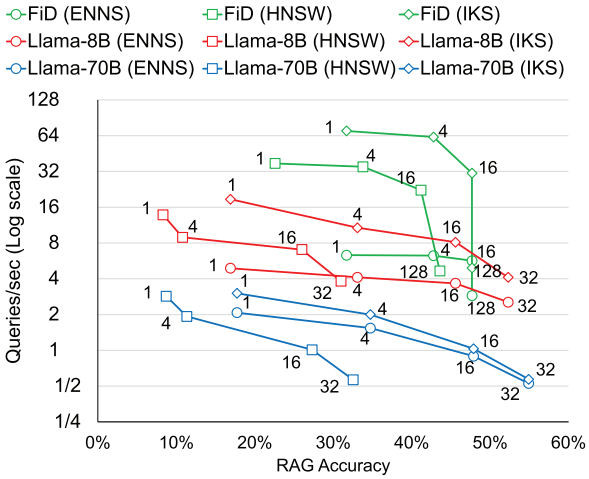


FIGURE 8. Comparison of accuracy and throughput of three RAG pipelines with different LLMs (FiDT5, Llama-3-8B, Llama-3-70B) with three different retrieval configurations: *ENNS* running on CPU, *HNSW* running on CPU, and *IKS* which offloads ENNS to IKS. The size of each data point represents document count (K value). HNSW is configured with *M*, *efConstruction*, and *efSearch* of 32, 128, and 4096, respectively. Batch size 16.

IKS Multitenancy and Memory Disaggregation

Spatial Multitenancy

Table 1 compares solo and co-run configurations of ENNS and Memcached. Allocating half the channels to ENNS nearly halves its queries per second (QPS), directly reflecting reduced memory bandwidth. Memcached latency and throughput remain essentially unaffected, demonstrating robust isolation of CPU traffic from ENNS.

Temporal Multitenancy

Figure 7 illustrates ENNS execution time and Memcached latency under temporal multitenancy. As corpus size grows, ENNS overhead rises modestly (by about 11%), and Memcached sees a small median latency increase but a more noticeable P99 increase. Temporal sharing ensures flexibility and high overall utilization, though it introduces some interference.

IKS Performance on RAG Applications

Figure 8 compares ENNS on CPU, HNSW on CPU, and ENNS accelerated by IKS. While HNSW offers higher

throughput than CPU-based ENNS, it sacrifices accuracy. With IKS acceleration, ENNS retrieval ceases to be a bottleneck, boosting throughput without diminishing accuracy. This enables high-quality RAG pipelines to achieve both high performance and accuracy, outperforming HNSW configurations.

CONCLUSION

In this work, we discussed IKS, a compute-enabled CXL memory expander that operates as a vector database accelerator while simultaneously disaggregating its internal memory to provide higher memory capacity. IKS leverages `CXL.mem` and `CXL.cache` to implement an optimized CPU-accelerator interface. IKS significantly improves the performance of RAG applications by offering accelerated, scalable, and accurate dense retrieval.

Processing near-memory enabled by CXL presents numerous opportunities for further research. We see IKS as an example of a PNM design that can leverage economies of scale to be successfully deployed, powering future compound AI systems.

ACKNOWLEDGMENTS

This work was supported in part by NSF grant 2239020 and in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation program sponsored by DARPA. Any opinions, findings, conclusions, and recommendations expressed in this material are those of the authors and do not necessarily reflect those of the sponsors.

REFERENCES

1. A. Basu, J. Gandhi, J. Chang, M. D. Hill, and M. M. Swift, "Efficient virtual memory for big memory servers," *ACM SIGARCH Comput. Archit. News*, vol. 41, no. 3, pp. 237–248, 2013.
2. D. Das Sharma, R. Blankenship, and D. Berger, "An introduction to the compute express link (CXL) interconnect," *ACM Comput. Surv.*, vol. 56, no. 11, pp. 1–37, Jul. 2024, doi: 10.1145/3669900.
3. F. Devaux, "The true processing in memory accelerator," in *Proc. IEEE Hot Chips 31 Symp. (HCS)*, 2019, pp. 1–24, doi: 10.1109/HOTCHIPS.2019.8875680.
4. G. Izacard and E. Grave, "Leveraging passage retrieval with generative models for open domain question answering," in *Proc. 16th Conf. Eur. Chapter Assoc. Comput. Linguistics*, P. Merlo, J. Tiedemann, and R. Tsarfaty, Eds., Apr. 2021, pp. 874–880.
5. V. Karpukhin et al., "Dense passage retrieval for open-domain question answering," 2020, *arXiv:2004.04906*.
6. A. K. Jones, R. E. Kessler and M. D. Hill, "Page placement algorithms for large real-indexed caches," *ACM Trans. Comput. Syst.*, vol. 10, no. 4, pp. 338–359, Nov. 1992, doi: 10.1145/138873.138876.
7. S. Lee et al., "Hardware architecture and software stack for PIM based on commercial DRAM technology: Industrial product," in *Proc. ACM/IEEE 48th Annu. Int. Symp. Comput. Archit. (ISCA)*, 2021, pp. 43–56, doi: 10.1109/ISCA52012.2021.00013.
8. H. Li et al., "Pond: CXL-based memory pooling systems for cloud platforms," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, New York, NY, USA: ACM, 2023, vol. 2, pp. 574–587.
9. D. Quinn et al., "Accelerating retrieval-augmented generation," in *Proc. 30th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, New York, NY, USA: ACM, 2025, pp. 15–32.
10. S.-Q. Yan, J.-C. Gu, Y. Zhu, and Z.-H. Ling, "Corrective retrieval augmented generation," 2024, *arXiv:2401.15884*.

DERRICK QUINN is a Ph.D. student in electrical and computer engineering at Cornell University, Ithaca, NY, 14850, NY. His research interests include computer architecture and systems at scale. Quinn received his bachelor of science degree in computer science and mathematics from the University of Kansas. Contact him at dq55@cornell.edu.

NEEL PATEL is a Ph.D. student in electrical and computer engineering at Cornell University, Ithaca, NY, 14850, NY. His research interests include computer architecture and systems at scale. Patel received his master of science in computer science from the University of Kansas. Contact him at nmp83@cornell.edu.

MOHAMMAD ALIAN is an assistant professor in electrical and computer engineering at Cornell University, Ithaca, NY, 14850, NY. His research interests include redefining the data-delivery hierarchy of future data centers. Alian received his Ph.D. degree from the University of Illinois Urbana Champaign. Contact him at malian@cornell.edu.

CXL–GPU: Pushing GPU Memory Boundaries with the Integration of CXL Technologies

Donghyun Gouk, *Panmnesia, Inc., Daejeon, 34136, South Korea*

Seungkwan Kang and Seungjun Lee, *KAIST, Daejeon, 34141, South Korea*

Jiseon Kim, Kyungkuk Nam, Eojin Ryu, Sangwon Lee, Dongpyung Kim, Junhyeok Jang, and Hanyeoreum Bae, *Panmnesia, Inc., Daejeon, 34136, South Korea*

Myoungsoo Jung, *Panmnesia, Inc., Daejeon, 34136, South Korea, KAIST, Daejeon, 34141, South Korea, and Panmnesia, Inc., Seoul, South Korea*

This work introduces a GPU storage expansion solution utilizing compute express link (CXL), featuring a novel GPU system design with multiple CXL root ports for integrating diverse storage media (dynamic random-access memory and/or solid-state drives). We developed and siliconized a custom CXL controller integrated at the hardware register transfer level (RTL), achieving two-digit nanosecond round-trip latency, the first in the field. This study also includes speculative read and deterministic store mechanisms to efficiently manage read and write operations to hide the endpoint's backend media latency variation. Performance evaluations reveal our approach significantly outperforms existing methods, marking a substantial advancement in GPU storage technology.

Large-scale deep learning models, such as large language models and mixtures of experts, have become prevalent across various computing domains. However, their memory demands often far exceed the capacity of accelerators like GPUs. For instance, while models with 1 billion parameters require approximately 16~24 GB of GPU memory for loading and training, models exceeding 100 billion parameters are increasingly commonplace.

To overcome these limitations, researchers have explored various approaches that leverage storage or external memory systems. One notable example is GPUDirect storage (*GPUDirect*[1]) which enables direct mapping of the GPU's PCIe base address register (BAR) to a target solid-state drive (SSD) by altering the storage stack. This approach facilitates the management of large-scale model parameters by utilizing the substantial capacity of SSDs. However, GPUDirect's adoption remains limited due to its complexity. Treating the SSD as a block device introduces challenges, including the management of the file system and mismatch in input–output (I/O) granularity. Furthermore, GPUDirect requires manual intervention for storage and memory operations, complicating the copy-then-execute programming model and deterring widespread usage.

In contrast, unified virtual memory (UVM[2]) provides a more straightforward solution by enabling shared virtual memory access for both CPU and GPU. This approach integrates GPU and host memory into a unified address space, allowing seamless data access. UVM automatically handles memory allocation and

Digital Object Identifier 10.1109/MM.2025.3582433
Date of publication 7 July 2025; date of current version 24 December 2025.

migration based on demand, simplifying the management of large datasets. As a result, UVM has become a widely adopted technique for deep learning frameworks like TensorFlow and DGL.

In this study, we present the compute express link (CXL) as a transformative technology for GPU storage expansion. Figure 1(c) illustrates the high-level framework for the proposed GPU storage expansion approach. CXL leverages the PCIe physical data link to map devices, referred to as endpoints (EPs), to a cacheable memory space accessible by the host. This architecture allows compute units to directly access EPs using standard memory requests. Unlike JEDEC's double data rate (DDR)-standard dynamic random-access memory (DRAM) interfaces, which rely on synchronous communication, CXL enables asynchronous communication with compute units, akin to block storage. This flexibility supports EP implementation using diverse storage media such as nonvolatile memory (NVM)-based SSDs[3] and DRAM.[4,5]

Despite its potential, integrating CXL for GPU storage expansion presents a significant challenge due to the absence of a native CXL logic fabric and subsystem in GPUs to support DRAM/SSD EPs as memory expansion devices. To address this limitation, we have developed a CXL hardware layer stack and implemented it through a custom CXL controller designed at the hardware register transfer level (RTL), integrated within the CXL logic fabric. Additionally, we propose a GPU architecture featuring multiple CXL root ports equipped with these controllers, each capable of interfacing with DRAM/SSD EPs via a host bridge. This work represents the first demonstration of a silicon-based CXL controller achieving sub-two-digit nanosecond round-trip latency, signifying a major advancement in high-speed memory expansion technology.

We further enhanced the design of our CXL controller to improve GPU storage expansion performance. While achieving the fastest CXL round-trip latency to date, the storage media in EPs remains slower than local GPU memory. To mitigate this latency gap, we propose two key strategies for GPU–EP interaction: speculative read (SR) and deterministic store (DS).

The SR mechanism leverages a module in our controller to manage load instructions, utilizing features from CXL 2.0. It anticipates target addresses by presharing them with DRAM/SSD EPs, enabling them to prefetch target pages before the actual requests. This approach significantly reduces the latency impact of the storage media. To avoid overloading EPs with SR requests, we integrate the CXL quality of service (QoS) telemetry feature to monitor the EP's status and regulate SR traffic, ensuring a balance between performance and system load. For scenarios involving slower EP write performance, such as with NVM-based SSDs, the DS mechanism addresses the variability in write speeds. It performs concurrent writes to both GPU memory and the SSD EP, immediately completing store requests. Since GPU workloads are predominantly read-oriented, this "fire-and-forget" strategy is proven effective. However, in cases with intensive write activity or SSD tail latencies, the DS mechanism identifies the SSD EP as a bottleneck. In such instances, our controller temporarily buffers incoming data in GPU memory, deferring their transfer to the SSD EP. This allows the GPU to sustain the fire-and-forget approach for the majority of store operations, delivering improved performance across diverse workloads.

To evaluate the performance of our CXL hardware framework, we fabricated the CXL controller using state-of-the-art silicon technology. Integration with the GPU was achieved using Vortex,[6] a RISC-V-based general-purpose GPU. The effectiveness of our GPU storage expansion approach was tested with custom-designed add-in-card (AIC) devices, as shown in Figure 1(d). Comprehensive testing was conducted using various storage media and GPU storage expansion techniques through RTL-level hardware simulations. Our GPU storage expansion approach significantly outperforms UVM and GPUDirect storage,[1] achieving 50× and 35.4× higher performance, respectively.

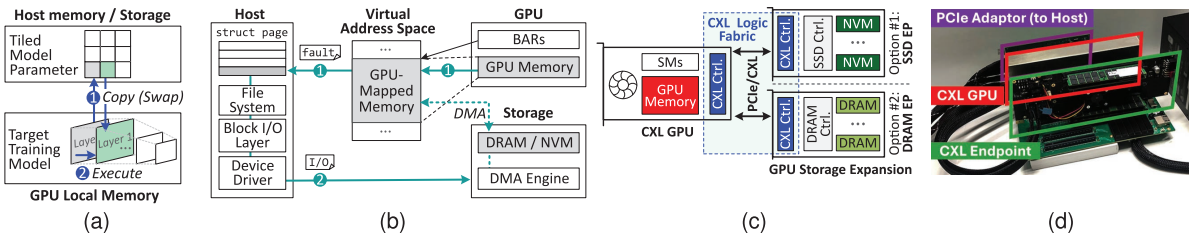


FIGURE 1. High-level view of GPU storage expansion. (a) Copy-then-execute. (b) GPU/Storage direct DMA. (c) Proposed CXL-based expansion. (d) AIC-level prototype.

MEMORY MANAGEMENT IN GPU AND CXL TECHNOLOGIES

Data Movement Management in GPUs

Copy-Then-Execute Model

GPU memory is crucial for storing not only the model parameters but also additional data such as gradients and intermediate computation buffers. The resulting memory requirement can be eight times larger than the parameters, exceeding GPU capacity. To address this limitation, GPUs often rely on host memory and storage, necessitating frequent data transfers between the CPU and GPU. Figure 1(a) illustrates the "copy-then-execute" model, commonly used for large-scale deep learning. In this model, parameters are divided into smaller segments or tiles, such as the model layers. This enables layer swapping ❶, allowing subsequent layers to process ❷. However, frequent data transfers create performance overhead and complicate GPU usage. Users must explicitly manage (tiled) data structures and coordinate data migrations, which is cumbersome and inefficient, ultimately posing a barrier to GPU adoption.

Unified Virtual Memory

NVIDIA and AMD support UVM[2] allowing both CPU and GPU to access data in a unified virtual address space. When the GPU requests data not in memory, a page fault occurs, triggering a PCIe interrupt to the host. The host-side software then allocates a new page in GPU memory and transfers the required data. UVM also replaces older pages with new data, ensuring efficient memory use. These features have made UVM widely adopted in frameworks like TensorFlow. However, UVM can introduce performance bottlenecks due to the substantial latency of the host runtime resolving GPU's page faults.

Direct Memory Access (DMA) for GPU and Storage

UVM faces limitations in memory capacity, as it relies on host-side DRAM. Alternative approaches expand GPU memory using block storage, significantly increasing memory capacity. Techniques such as GPUDirect[1] grant the storage's DMA engine to access the GPU's memory space, enabling direct data transfers between GPU and storage. As shown in Figure 1(b), they achieve this by mapping the GPU's BAR to the virtual address space and creating a `struct page` for GPU memory within the host kernel. This configuration allows the host's filesystem to share the GPU-mapped memory with storage, enabling DMA operations directly to GPU memory. However, these approaches are limited as they rely solely on the storage DMA engine. Consequently, when the GPU encounters on-demand page faults ❶, the host runtime must translate them into storage I/O requests ❷, resulting in overheads comparable to UVM.

CXL Memory Expander

End-to-End Latency Analysis

Many industry vendors have developed prototypes or proofs-of-concept for CXL memory expanders that leverage CXL 2.0, but commercial products or in-depth latency information from an end-user perspective remains scarce.

CXL's latency differs from JEDEC/DDR standards, which have precise timing specifications. Instead, CXL latency encompasses the full round-trip path across multiple layers. A memory request from the host CPU is converted into a CXL flit at the transaction layer, which supports the functionalities and interfaces required by each subprotocol. This flit progresses to the link layer, which manages communications such as buffering and acknowledgments. The flit is then passed to the flex bus physical layer, which sends it to the target EP based on link speed and lane configuration. For accurate management of CXL subprotocols, the EP must integrate these layers as well. After the EP's layer stack, the flit returns to the system bus, reaching either a memory or an SSD controller. The data are then retrieved from the storage medium and returned to the host CPU's system bus, traversing the hardware layers. The structure and function of this hardware layer stack play a critical role in determining roundtrip latency. Although market products are yet to be released, studies conducted by Samsung[4] and Meta[5] have reported latency of 250 ns for their prototypes.

CXL With an SSD Integration

The asynchronous nature of CXL.mem enables various memory types, including SSDs and different DRAM versions, providing flexibility in configurations. Industry proposals have explored integrating SSDs with CXL, including Intel's planned release of a CXL-attached Optane SSD and Samsung's introduction of the CXL-hybrid SSD (CMM-H). Although not yet commercialized, these SSDs are expected to use DRAM as a cache to mitigate slow storage media. Consequently, the performance will depend heavily on effective DRAM management. In addition, write operations on such devices are anticipated to be slower than reads and may occasionally encounter tail latency. For instance, flash memory necessitates garbage collection to reconcile

the mismatch between write and erase unit sizes. Therefore, SSD-CXL integration must carefully manage writes to ensure performance and reliability.

DESIGN OF CXL-INTEGRATED GPU

CXL Hardware Layer Stack

In response to the lack of a publicly accessible CXL hardware stack, we developed a series of hardware layers that support the CXL subprotocols, consolidating these into a unified controller. The controller is designed for CXL 3.1 while offering backward compatibility with earlier versions of CXL (2.0/1.1). The flex bus physical layer is integrated with our PCIe physical coding sublayer, supporting both PCIe and CXL layer stacks seamlessly over elastic buffers.[7]

To address the dual requirements of PCIe and CXL, particularly the power management and administrative operations, an arbitrator state machine has been incorporated into the controller. This state machine optimizes efficient resource allocation between PCIe and CXL tasks, ensuring performance and stability. The controller has undergone extensive testing, achieving a roundtrip latency in tens of nanoseconds, including the overhead of protocol conversion between standard memory operations and CXL flits. The controller has been integrated into hardware RTL implementations of GPU/CPU prototypes and memory expanders. Overall, our controller reduces latency by over three times compared to SMT4 and TPP.[5]

GPU Architecture Design and Integration

For the EP devices, we have integrated the CXL controller with memory and SSD controllers. This allows extending its back-end storage to the host system as host-managed device memory (HDM). The host is informed of this functionality through a feature analogous to PCIe capabilities, specifically adapted for HDM usage. Integrating with GPU architectures, however, poses challenges because the EP device must interface with the GPU cache. To address this, we designed a specialized CXL root complex with a host bridge that includes multiple root ports. This configuration, shown in Figure 2(a), connects the host bridge to the system bus port and several CXL root ports. The core component, the HDM decoder, manages the allocation of system memory address ranges, referred to as host physical addresses (HPA), for each root port. When a GPU issues a memory request ❶, the corresponding root port translates the request into a CXL flit ❷ and forwards it to the CXL controller ❸. This design ensures seamless PU-storage communication, facilitating efficient data access.

System Configuration and Memory Space

Figure 2(a) shows the CXL root complex placement within our GPU architecture, implemented using the Vortex GPU framework and its hardware RTL.[6] In the Vortex architecture, the GPU's computational units, known as streaming multiprocessors (SMs), connect to the system bus through the last-level cache. The system bus also interfaces with a GPU memory controller and a PCIe EP, which supports host communication and kernel execution. In our design, the CXL root complex is integrated into the system bus alongside a simplified core responsible for initializing the connected EPs, the host bridge's HDM decoder, and the HPAs of each root port. During initialization, firmware identifies CXL EPs by examining their configuration space and PCIe BARs and aggregates each EP's memory address space by analyzing the HDM capability registers. The firmware then records this information in the HDM decoder of the host bridge, specifying the base address and size of the HDM for each root port [cf. Figure 2(a)].

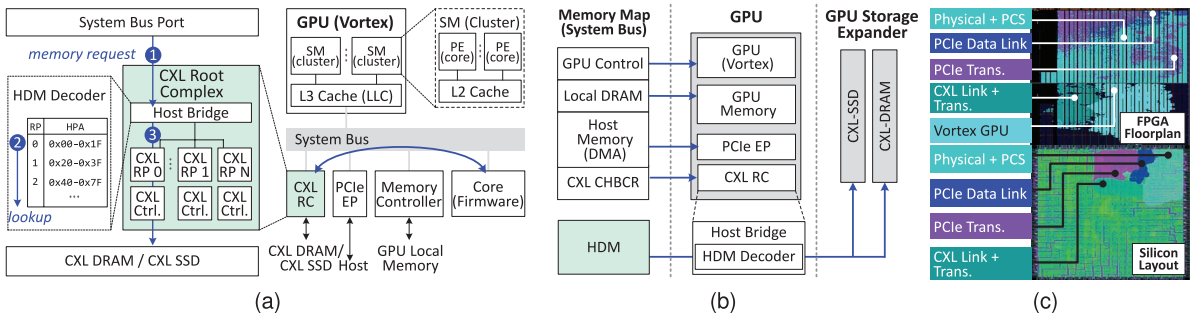


FIGURE 2. Design of CXL-integrated GPU. (a) GPU architecture. (b) GPU memory space. (c) Hardware prototypes.

After initialization, the GPU's system bus memory map is structured as shown in Figure 2(b). This map defines the allocation of address space across devices, segmented by function. For instance, when an SM generates a request targeting host memory, it interacts with the PCIe EP, which routes the request to the host. Similarly, GPU storage expanders under the CXL root complex are integrated into the system's memory architecture. Upon receiving a memory request for this system memory segment, the CXL root complex consults the HDM decoder, converts the request into a CXL flit, and forwards it to the corresponding root port and controller.

The systems employing our CXL controllers, including the Vortex GPU and EP devices, are realized on a 7-nm field-programmable gate array-based custom AIC. The associated RTL designs are detailed in Figure 2(c).

OPTIMIZATION OF CXL CONTROLLER FOR GPUs

The CXL round-trip latency depends on the back-end media type, such as DRAM or SSD. To address the latency challenges, we propose two optional strategies: 1) SR and 2) DS. SR aims to minimize read latency, while DS mitigates tail latencies during writes.

Accelerating Reads With Speculation

To enhance read performance, we employ the speculative read feature (`MemSpecRd`) introduced in CXL 2.0. This feature operates similarly to standard memory requests but provides hints about the addresses to be accessed shortly. Figure 3(a) shows our queue logic beneath the root port, consisting of two queues: the SR queue and the memory queue, each with 32 entries. Load requests are first added to the SR queue and processed into a `MemSpecRd` operation.

Implementation

To adapt `MemSpecRd`, we modify its address format. While originally designed for 64-B granularity, we repurpose the two least significant bits to indicate the length of the request, with the remaining bits specifying a 256-B memory offset, allowing requests to be coalesced. The requests in the SR queue are forwarded to the memory queue if available space exists. Once in the memory queue, requests are forwarded to the transaction layer. The SR reader module records the address and length of each issued request in a ring buffer. If a new request matches a previously issued SR request, it is directly forwarded to the memory queue. This allows the EP-side controller to preload data before the actual read requests.

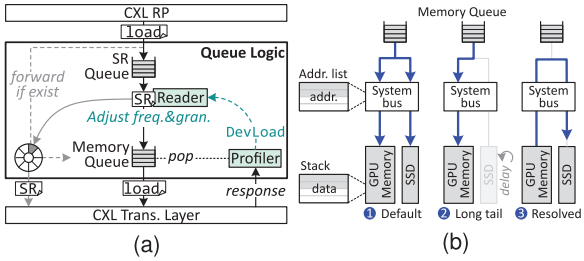


FIGURE 3. Optimization of CXL controller. (a) Speculative read. (b) Deterministic store.

When the EP's response arrives, the queue logic's profiler removes the completed request from the memory queue. Because SR requests also contribute to the back-end media bandwidth, the profiler keeps track of the EP's status by analyzing the QoS telemetry data of the response, specifically the `DevLoad` field.[7] The `DevLoad` field, which reflects the workload of the EP, is shared with the SR reader to dynamically adjust the granularity of SR requests, optimizing system performance. The details are described next.

Load Control for Speculative Reads

The `DevLoad` field, defined as two bits, classifies the device's current load into four states: light load (`ll`), optimal load (`ol`), moderate overload (`mo`), and severe overload (`so`). When the device is in a `ll` state, the back-end media has sufficient bandwidth. In this case, the queue logic increases the granularity of SR requests by utilizing the modified `MemSpecRd` format, enabling efficient data preloading. In the `ol` state, the device operates at full bandwidth without being overwhelmed, so the current granularity is maintained. In contrast, a `mo` state indicates that the back-end media is experiencing increased traffic. To alleviate this, the granularity of SR requests is reduced to fetch only essential data. In a `so` state, the memory queue is completely saturated. The queue logic temporarily halts SR requests until the EP recovers and returns the `ll` state. This adaptive load control mechanism dynamically balances efficiency and stability, ensuring optimal performance across varying conditions.

Address Window Control

SR can improve the read performance of SSD-based EP devices by preloading data into the internal DRAM. However, because actual CXL requests operate at a 64-B granularity, preloading wrong data can cause

internal DRAM pollution and waste back-end media bandwidth. To mitigate this, the queue logic optimizes the address window for each SR request by analyzing the memory and SR queue. An address window consists of a starting and ending address. The queue logic begins with an initial address window starting and ending at the incoming SR request's address minus and plus its granularity, respectively. Since requests in the memory queue and SR queue each represent prior and future requests, the queue logic adjusts the window accordingly. Specifically, for each request in the memory and SR queue, the queue logic shifts the start and endpoint of the address window upwards and downwards by the same granularity, respectively. Finally, the computed address window is rounded to the nearest 256-B boundary. This ensures that SR requests target the most relevant data range, minimizing unnecessary data movement and optimizing DRAM utilization.

Deterministic Store for Precise Writes

The proposed DS technique conceals the slow write latencies by leveraging reserved space in the GPU's local memory.

Design and Implementation

As shown in Figure 3(b), when a write operation to an SSD is initiated, the request is concurrently sent to both the GPU memory and the SSD, allowing immediate release of the request ❶. If a delay of the SSD's response is observed, the queue logic temporarily stores the data from subsequent write request in the GPU memory's reserved address ❷. This temporary data storage is organized using a stack structure, designed to fill up during slow writes events and updated in place elsewise. To maintain a record of each stack entry, an address list is stored within the system bus's internal SRAM. The stack is methodically emptied by flushing the data in the background ❸. Through the implementation of DS, the CXL root complex effectively shields the GPU from variations in write latency.

Fine Control for Internal Tasks

While the proposed hybrid memory management reduces write latency variability, internal SSD tasks such as garbage collection require finer control to mitigate their long tail latency. These tasks can temporarily make the SSD unavailable and cause congestion at the ingress port, degrading the performance of even read operations. To address this, we utilize the `DevLoad` field as well. Specifically, when an internal task is expected to reduce throughput temporarily, the back-end media reports this through the `DevLoad` field beforehand. By monitoring `DevLoad`, our controller dynamically throttles write requests to the affected port and forwards them to GPU memory, ensuring that the throughput reduction does not lead to severe performance degradation.

EVALUATION

Methodology

Our hardware prototype provides detailed latency characteristics that are not accessible to end users but lacks flexibility to explore the design space for storage expansion. To address this, we developed a simulator that accurately models the behavior of our prototype. We performed an RTL behavior simulation of Vortex using Verilator[8] and extracted data based on its performance counters and waveform dump. In addition, to add a back-end media, we connected DRAMSim[39] to the memory interface in the verilated RTL. To account for latency from PCIe/CXL/host runtime, we created a custom DRAMSim configuration that includes this additional latency. For PCIe and CXL bus latencies, we used real values measured from our application-specified integrated circuit. For UVM and GPUDirect's host runtime overhead, we accounted 45 µs.[10]

Configurations

For the evaluation, we consider five different GPU configurations: 1) `UVM`, 2) `GDS`, 3) `CXL`, 4) `CXL-SR`, and 5) `CXL-DS`. `UVM` employs NVIDIA's unified virtual memory,[2] extending GPU memory using host DRAM. `GDS` uses GPUDirect storage,[1] enabling DMA between the GPU and storage. In both cases, data are transferred to the GPU per page through the host runtime. `CXL` represents an expander implemented with proposed controller. `CXL-SR` and `CXL-DS` build upon `CXL` by incorporating SR and DS techniques, respectively. We also evaluate an ideal configuration, `GPU-DRAM`, which assumes infinite on-device GPU memory. For SR and DS evaluations, we simulate various storage media as in Table 1(a).

Workloads

We evaluate eleven workloads from the Rodinia benchmark suite[11] and also evaluate two real-world workloads, `gnn` and `mri`. The `gnn` workload is composed of `bfs`, `vadd`, and `gemm`, while the `mri` workload is composed of `sort` and `conv3`. To ensure fair comparison across configurations, the input data sizes are adjusted to fit within the 10× bigger capacity of the GPU's local memory. For clarity, the workloads are categorized into

compute and memory-intensive applications, with the latter further divided into load and store-intensive subsets. All workloads are arranged in ascending order based on their memory access ratios. The important characteristics of the workloads are summarized in Table 1(b).

Performance Analysis

Figure 4(a) and (b) presents the performance of various GPU configurations with CXL memory expanders using DRAM and Z-NAND as back-end media, respectively. All execution times are normalized to the performance of the GPU-DRAM configuration.

DRAM-Based Expanders

Figure 4(a) compares UVM, CXL, and GPU-DRAM, as SR and DS mechanisms are only relevant for expanders with non-DRAM back-end media. As shown, UVM performs 52.7 × worse than GPU-DRAM, with pronounced latency increases for load-intensive workloads such as *gemm*, *vadd*, and *saxpy*. These workloads involve sequential operations like vector addition, where data are read once and seldom accessed again. This access pattern generates numerous page faults, causing host runtime intervention. In contrast, CXL improves performance by 50 × over UVM, by accessing the extended DRAM directly. The performance of CXL approaches that of GPU-DRAM, being slower by only 2.3%, 19.7%, and 6.8% for compute-intensive, load-intensive, and store-intensive applications, respectively. Note that the real-world applications show patterns similar to the operations that compose each application.

SSD-Based Expanders

Due to the significant performance variation, Figure 4(b) presents the results in a logarithmic scale. This analysis highlights SR and DS techniques' benefits. As shown, CXL-SR improves performance by 7.4 × compared to CXL. This is due to the SR requests preloading data into internal DRAM, allowing sequential reads to be served immediately. However, the effectiveness of SR depends on applications exhibiting data access localities and avoiding excessive jumps across SR requests. For instance, 1-D vector computation (e.g., *vadd*, *saxpy*) and 2-D array computation workloads (e.g., *gemm*, *conv3*) exhibit the highest performance gains (15.6 ×) by issuing SR requests for upcoming computations. In contrast, graph-based applications with irregular access patterns, such as *path* and *bfs*, show limited benefits (67.6%). The internal DRAM can become crowded, leading to data eviction and reducing the effectiveness of SR.

TABLE 1. Evaluation setup. (a) Configs. (b) Workloads.

Vortex Configuration	Cores/Threads 8/8
PCIe Configuration	PCIe 5.0 (32GT/s) x8
CXL Configuration	PCIe 5.0 (32GT/s) x8 Sync-header bypass
DRAM	DDR5-5600[9]
Optane	Intel P5800X
Z-NAND	Samsung 983 ZET
NAND	Samsung 980 Pro

(a)

Workload Type	Acronym	Compute Ratio	Load Ratio
Compute Intensive	rsum stencil sort	31.4% 37.5% 38.1%	53.3% 72.5% 98.7%
Load Intensive	gemm vadd saxpy conv3 path	11.6% 15.6% 16.2% 21.8% 27.0%	99.9% 69.1% 69.2% 78.6% 92.7%
Store Intensive	cfd gauss bfs	20.9% 23.5% 29.3%	42.6% 48.5% 43.2%
Real-World Workload	gnn mri	27.4% 29.2%	73.8% 53.3%

(b)

CXL-DS improves performance by 20.9%, 8.7%, and 62.8% over CXL-SR for each category. These benefits are particularly pronounced in store-intensive applications, as DS mitigates tail latency caused by internal SSD operations, preventing GPU threads from stalling.

Backend Media Latency Mitigation

Figure 4(c) illustrates the performance benefits of the SR and DS mechanisms across various back-end media, specifically Optane (O), Z-NAND (Z), and standard NAND (N). We analyzed workloads with a sequential access pattern (vadd), a random and load-intensive pattern (path), and a random and store-intensive pattern (bfs). For sequential workloads, SR can provide near-DRAM performance by hiding the latency of the back-end media. For instance, vadd achieves 21.5 × performance improvement compared to CXL when using SR. DS provides up to a 4× performance gain for store-intensive applications (bfs) by hiding the tail latency during media management (e.g., garbage collection). The detailed analysis of each mechanism is described next.

Speculative Read

Figure 4(d) illustrates the performance benefits of the proposed SR mechanism with Z-NAND-based memory expansion. CXL-NAIVE generates 64-B MemSpecRd requests for all incoming requests. In contrast, CXL-DYN coalesces multiple SR requests using our flexible address format. CXL-SR builds on CXL-DYN by analyzing the SR and memory queues to adjust the SR address window. For comparison, the workloads were categorized by memory access patterns. Seq and Rand represent sequential and random memory accesses, respectively. Around represents access patterns where the direction of the sequential access alters in runtime. For instance, Gaussian elimination (gauss) accesses the current or previous row based on the row's first element.

CXL-NAIVE increases performance 1.9 × compared to CXL. This benefit occurs because data can be read from NAND in advance by sending MemSpecRd requests for requests that are waiting in the GPU's memory queue. This approach prevents cold misses in the SSD's internal DRAM for all workloads. Specifically, under CXL, the Seq, Around, and Rand workloads reach DRAM hit rates of 47.4%, 31.2%, and 10%, respectively, while under CXL-NAIVE, they reach 88.4%, 56%, and 32.1%, respectively.

However, CXL-NAIVE remains 6.6 × slower than GPU-DRAM due to the limited coverage of the SR requests. CXL-DYN achieves an additional performance gain of 4.5 × for Seq workloads by providing the opportunity to preload data more data promptly. This is because the aggressive preloading raises the SSD DRAM hit rate for Seq workloads to over 99%. Around and Rand workloads gain less due to the probabilistic utilization of the preloaded data, resulting in 10.6% and 5.7% improvement, respectively. For Rand workloads, simply increasing the granularity of preloading can lead to SSD DRAM pollution and reduce performance. In practice, sending the largest possible SR requests at all times lowers the SSD DRAM hit rate for Rand from 32.1% to 30.7%. However, since CXL-DYN can dynamically adjust the granularity of MemSpecRd requests, it achieves a higher hit rate (34%) for Rand.

For Around workloads, which generally follow sequential access patterns, the SSD DRAM hit rate is higher than CXL-NAIVE, but it still remains at 57.4% because the next access may occur either before or after the current address. CXL-SR overcomes this limitation by dynamically adjusting the address window of the SR requests, yielding an additional 2.1 × performance improvement for Around. This is possible because CXL-SR can analyze the incoming addresses and decide whether to send MemSpecRd requests for addresses before or after the current one, raising the SSD DRAM hit rate for Around to 75.8%.

Deterministic Store

We evaluated two Z-NAND-based systems, each with (CXL-DS) and without (CXL-SR) our proposed DS mechanism, under long-tail cases due to garbage collection. Figure 4(e) shows the time series of load/store latencies and ingress queue occupancy. CXL-SR experiences higher load/store latencies as store requests saturate the ingress queue. The GC process even recurs shortly after completion, as the accumulated store requests flood into the media, depleting the free pages. In contrast, CXL-DS avoids sending store requests to the EP. This strategy prevents ingress queue congestion,

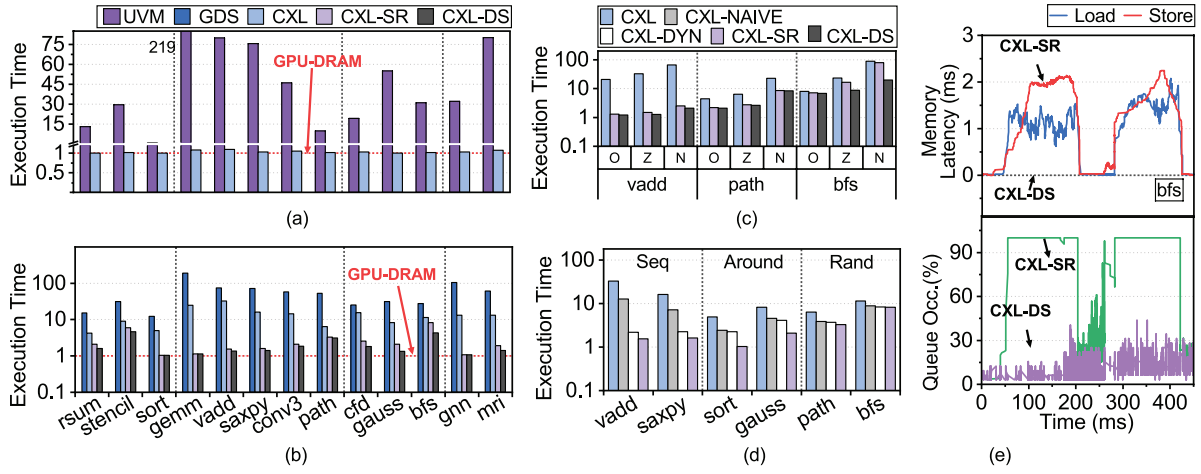


FIGURE 4. Performance analysis. (a) Overall (DRAM). (b) Overall (Z-NAND). (c) Backend media sensitivity. (d) Speculative read. (e) Deterministic store.

hiding tail latency of both load and store requests, and prevents subsequent GC.

CONCLUSION

We introduce a method to increase GPU storage using CXL technology, integrating DRAM and/or SSDs through multiple CXL root ports. Our custom-designed CXL controller, implemented directly in the hardware, achieves fast response times not seen before in this field. We further refine this system with features designed to improve both read and write operations to storage media. Our tests show that this approach improves upon existing solutions, offering a significant step forward in GPU storage capacity and efficiency.

ACKNOWLEDGMENTS

The work was supported in part by IITP under Grants RS-2023-00221040, RS-2024-00460762, RS-2025-02214652, RS-2025-02214654, and RS-2025-02263080, KIAT under Grants P0027923 and P0028225, and the MSS Technology Development Program under Grant RS-2023-00303967. The authors thank anonymous reviewers for their constructive feedback. Authors associated with KAIST contributed to introducing the concept of speculative read and deterministic store, with all remaining work and contributions brought by Panmnesia. This work is protected by one or more patents.

REFERENCES

1. "GPUdirect storage." NVIDIA. Accessed: Feb. 26, 2025. [Online]. Available: https://docs.nvidia.com/gpudirect-storage/
2. "CUDA toolkit." NVIDIA. Accessed: Feb. 26, 2025. [Online]. Available: https://docs.nvidia.com/cuda/cuda-driver-api/
3. M. Jung, "Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD)," in *Proc. 14th ACM Workshop Hot Topics Storage File Syst. (HotStorage)*, 2022, pp. 45–51, doi: 10.1145/3538643.3539745.
4. K. Kim et al., "SMT: Software-defined memory tiering for heterogeneous computing systems with CXL memory expander," *IEEE Micro*, vol. 43, no. 2, pp. 20–29, Mar./Apr. 2023, doi: 10.1109/MM.2023.3240774.
5. H. A. Maruf et al., "TPP: Transparent page placement for CXL-enabled tiered-memory," in *Proc. 28th ACM Int. Conf. Architectural Support Program. Lang. Operating Syst. (ASPLOS)*, 2023, vol. 3, pp. 742–755, doi: 10.1145/3582016.3582063.
6. B. Tine, K. P. Yalamarthy, F. Elsabbagh, and K. Hyesoon, "Vortex: Extending the RISC-V ISA for GPGPU and 3D-graphics," in *Proc. 54th Annu. IEEE/ACM Int. Symp. Microarchit. (MICRO)*, 2021, pp. 754–766, doi: 10.1145/3466752.3480128.
7. "CXL specification revision 3.1" Compute Express Link, Beaverton, OR, USA, Aug. 2023. Accessed: Feb. 26, 2025. [Online]. Available: https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf
8. W. Snyder, P. Wasson, and D. Galbi. "Verilator." Veripool. Accessed: Feb. 26, 2025. [Online]. Available: https://www.verilator.org
9. S. Li, Z. Yang, D. Reddy, A. Srivastava, and B. Jacob, "DRAMsim3: A cycle-accurate, thermal-capable DRAM simulator," *IEEE Comput. Archit. Lett.*, vol. 19, no. 2, pp. 106–109, Jul./Dec. 2020, doi: 10.1109/LCA.2020.2973991.
10. T. Allen and R. Ge, "Demystifying GPU UVM cost with deep runtime and workload analysis," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. (IPDPS)*, 2021, pp. 141–150, doi: 10.1109/IPDPS49936.2021.00023.
11. S. Che et al., "Rodinia: A benchmark suite for heterogeneous computing," in *Proc. IEEE Int. Symp. Workload Characterization (IISWC)*, Austin, TX, USA, 2009, pp. 44–54, doi: 10.1109/IISWC.2009.5306797.

DONGHYUN GOUK is the chief architect officer at Panmnesia, Inc, Daejeon, 34136, South Korea. His research interests include computing switch/networks, artificial intelligence infrastructures, RISC-V, and link solutions including CXL, UALink, and Ethernet. Contact him at donghyun@panmnesia.com.

SEUNGKWAN KANG is working toward his Ph.D. degree at KAIST, Daejeon, 34141, South Korea. His research interests include storage-integrated accelerators. Kang received his B.S. degree in electrical engineering from KAIST. Contact him at kangsk@camelab.org.

SEUNGJUN LEE is working toward his Ph.D. degree at KAIST, Daejeon, 34141, South Korea. His research interests include OS and storage stack optimization. Lee received his B.S. degree in electrical engineering from KAIST. Contact him at sjlee@camelab.org.

JISEON KIM is an assistant director at Panmnesia, Inc, Daejeon, 34136, South Korea. Her research interests include edge computing, on-device artificial intelligence, memory, and link solutions including CXL, UALink, and Ethernet. Contact her at jskim@panmnesia.com.

KYUNGKUK NAM is a senior professional at Panmnesia, Inc, Daejeon, 34136, South Korea. His research interests include ML acceleration, computing switch/networks, artificial intelligence infrastructures, and link solutions including CXL, UALink, and Ethernet. Contact him at kknam@panmnesia.com.

EOJIN RYU is an assistant director at Panmnesia, Inc, Daejeon, 34136, South Korea. His research interests include GPU/SSD integration, edge computing, and link solutions including CXL, UALink, and Ethernet. Contact him at neo21top@camelab.org.

SANGWON LEE is a director at Panmnesia, Inc, Daejeon, 34136, South Korea. His research interests include persistent memory, RISC-V, and link solutions including CXL, UALink, and Ethernet. Lee received his Ph.D. degree from KAIST. Contact him at swon@panmnesia.com.

DONGPYUNG KIM is an assistant director at Panmnesia, Inc, Daejeon, 34136, South Korea. His research interests include GPU/SSD integration, edge computing, and link solutions including CXL, UALink, and Ethernet. Contact him at dpkim@panmnesia.com.

JUNHYEOK JANG is a director at Panmnesia, Inc, Daejeon, 34136, South Korea. His research interests include machine learning, NVMe, reliability, security, and link solutions including CXL, UALink, and Ethernet. Contact him at jhjang@camelab.org.

HANYEOREUM BAE is a director at Panmnesia, Inc, Daejeon, 34136, South Korea. His research interests include new storage, ZNS, and link solutions including CXL, UALink, and Ethernet. Contact him at hyr.bae@camelab.org.

MYOUNGSOO JUNG is a full professor in the School of Electrical Engineering, KAIST, Daejeon, 34141, South Korea, and the CEO of Panmnesia, Inc, Daejeon, 34136, South Korea. His research interests include computer architecture, operating systems, storage systems, nonvolatile memory, parallel processing, heterogeneous computing, edge computing, artificial intelligence infrastructure, and link solutions including CXL, UALink, and Ethernet. He is the corresponding author of this article. Jung received his Ph.D. degree in computer science and engineering from The Pennsylvania State University. Contact him at mj@panmnesia.com.

IEEE COMPUTER SOCIETY

www.computer.org

PURPOSE: Engaging professionals from all areas of computing, the IEEE Computer Society sets the standard for education and engagement that fuels global technological advancement. Through conferences, publications, and programs, IEEE CS empowers, guides, and shapes the future of its members, and the greater industry, enabling new opportunities to better serve our world.

OMBUDSMAN: Contact ombudsman@computer.org.

CHAPTERS: Regular and student chapters worldwide provide the opportunity to interact with colleagues, hear technical experts, and serve the local professional community.

PUBLICATIONS AND ACTIVITIES

***Computer*:** The flagship publication of the IEEE Computer Society, *Computer*, publishes peer-reviewed technical content that covers all aspects of computer science, computer engineering, technology, and applications.

Periodicals: The IEEE CS publishes 12 magazines, 18 journals

Conference Proceedings & Books: Conference Publishing Services publishes more than 275 titles every year.

Standards Working Groups: More than 150 groups produce IEEE standards used throughout the world.

Technical Communities: TCs provide professional interaction in more than 30 technical areas and directly influence computer engineering conferences and publications.

Conferences/Education: The IEEE CS holds more than 215 conferences each year and sponsors many educational activities, including computing science accreditation.

Certifications: The IEEE CS offers three software developer credentials.

AVAILABLE INFORMATION

To check membership status, report an address change, or obtain information, contact help@computer.org.

IEEE COMPUTER SOCIETY OFFICES

WASHINGTON, D.C.:
2001 L St., Ste. 700,
Washington, D.C. 20036-4928
Phone: +1 202 371 0101
Fax: +1 202 728 9614
Email: help@computer.org

LOS ALAMITOS:
10662 Los Vaqueros Cir.,
Los Alamitos, CA 90720
Phone: +1 714 821 8380
Email: help@computer.org

IEEE CS EXECUTIVE STAFF

Interim Executive Director: Anne Marie Kelly
Director, Governance & Associate Executive Director: Anne Marie Kelly
Director, Conference Operations: Silvia Ceballos
Director, Information Technology & Services: Sumit Kacker
Director, Marketing & Sales: Michelle Tubb
Director, Membership Development: Eric Berkowitz
Director, Periodicals & Special Projects: Robin Baldwin

IEEE CS EXECUTIVE COMMITTEE

President: Hironori Washizaki
President-Elect: Grace A. Lewis
Past President: Jyotika Athavale
Vice President: Nils Aschenbruck
Secretary: Yoshiko Yasuda
Treasurer: Darren Galpin
VP, Member & Geographic Activities: Andrew Seely
VP, Professional & Educational Activities: Cyril Onwubiko
VP, Publications: Charles (Chuck) Hansen
VP, Standards Activities: Edward Au
VP, Technical & Conference Activities: Terry Benzel
2025–2026 IEEE Division VIII Director: Cecilia Metra
2024–2025 IEEE Division V Director: Christina M. Schober
2025 IEEE Division V Director-Elect: Leila De Floriani

IEEE CS BOARD OF GOVERNORS

Term Expiring 2025:
İlkay Altintaş, Joaquim Jorge, Rick Kazman, Carolyn McGregor, Andrew Seely

Term Expiring 2026:
Megha Ben, Terry Benzel, Mrinal Karvir, Andreas Reinhardt, Deborah Silver, Yoshiko Yasuda

Term Expiring 2027:
Sven Dickinson, Alfredo Goldman, Daniel S. Katz, Yuhong Liu, Ladan Tahvildari, Damla Turgut

IEEE EXECUTIVE STAFF

Executive Director and COO: Sophia Muirhead
General Counsel and Chief Compliance Officer: Ahsaki Benion
Chief Human Resources Officer: Cheri N. Collins Wideman
Managing Director, IEEE-USA: Russell Harrison
Chief Marketing Officer: Jayne O'Brien
Chief Publication Officer and Managing Director: Steven Heffner
Staff Executive, Corporate Activities: Donna Hourican
Managing Director, Member and Geographic Activities: Cecelia Jankowski
Chief of Staff to the Executive Director: TBA
Managing Director, Educational Activities: Jamie Moesch
IEEE Standards Association Managing Director: Alpesh Shah
Chief Financial Officer: Kelly Armstrong
Chief Information Digital Officer: Jeff Strohschein
Managing Director, Conferences, Events, and Experiences: Marie Hunter
Managing Director, Technical Activities: Mojdeh Bahar

IEEE OFFICERS

President & CEO: Kathleen A. Kramer
President-Elect: Mary Ellen Randall
Past President: Thomas M. Coughlin
Director & Secretary: Forrest D. Wright
Director & Treasurer: Gerardo Barbosa
Director & VP, Publication Services & Products: W. Clem Karl
Director & VP, Educational Activities: Timothy P. Kurzweg
Director & VP, Membership and Geographic Activities: Antonio Luque
Director & President, Standards Association: Gary R. Hoffman
Director & VP, Technical Activities: Dalma Novak
Director & President, IEEE-USA: Timothy T. Lee

IEEE

revised 2 September 2025

EDITOR: Joshua J. Yi, josh@joshuayipatentlaw.com

DEPARTMENT: MICRO LAW

A Review of *Wisconsin Alumni Research Foundation v. Apple*—Part VII

Joshua J. Yi, *The Law Office of Joshua J. Yi, PLLC, Austin, TX, 78750, USA*

Part I of this series introduced the *Wisconsin Alumni Research Foundation v. Apple* cases and described the asserted patent (U.S. Patent No. 5,781,752). That article also summarized some recent large verdicts for patents asserted by academic institutions and provided several reasons why this series may be of interest to the readership of *IEEE Micro*, most notably because the inventors are well-known and several well-known computer architects worked as experts on this case. Part II described the complaints; namely, it described the plaintiff, Wisconsin Alumni Research Foundation ("WARF"), the inventors, and WARF's allegations as to how the Apple's products infringed WARF's patent. Part III described Apple's answer to the allegations in WARF's complaint, Apple's counterclaims, and WARF's response to those counterclaims. Part IV examined Apple's allegation of inequitable conduct by the inventors, a technical analysis of that allegation, and Judge Conley's legal analysis of the sufficiency of Apple's allegations. Parts V and VI reviewed WARF's motion to compel Apple to produce executable versions of the simulators for its A6, A7, and A8 processors (hereinafter "Accused Products") and the hearing that Magistrate Judge Crocker (hereinafter "Judge Crocker") had on that motion.

MOTION TO COMPEL INFORMATION REGARDING FUTURE PROCESSORS

In addition to WARF's motion to compel Apple to produce executable versions of the simulators of the accused processors, WARF filed three more motions to compel. This article covers the first and second of those motions; the third one will not be covered in this article series as it likely is of less interest to the readership of *IEEE Micro*.

Digital Object Identifier 10.1109/MM.2025.3638484
Date of current version 24 December 2025.

WARF filed the first of these motions to compel on 10 October 2014, just one week after it filed the simulator-related motion to compel (see *Wis. Alumni Rsch. Found. v. Apple, Inc.*[1]). In this motion WARF requested that the Court order Apple to produce documents, source code, etc. for additional accused processors (A8 and two future, unreleased processors) that likely use the patented invention (see pages 1–2 *Wis. Alumni Rsch. Found. v. Apple, Inc.*[1]).

With respect to the A8 processor, Apple responded that "[o]nce products incorporating the A8 processor were released for sale to the public, and WARF actually accused the A8 processor of infringement on September 16, 2014, Apple informed WARF that it would produce discovery for the A8" (see page 5 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[2]). With respect to unreleased processors, Apple argued that they were "still undergoing major design changes" and were "not finalized" (see page 2 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[2]). Apple argued that the on-going design changes and the accompanying documentation would: 1) require "repeated collections and productions of documents and source code through the end of fact discovery," which will impose a significant burden on Apple, and 2) result in WARF constantly changing its infringement theories and expert reports of those infringement theories to track Apple's design changes (see page 3 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[2]).

Judge Crocker held a telephonic hearing on 31 October 2014 for this motion and the motion to compel regarding production of an executable version of the simulator (see *Wis. Alumni Rsch. Found. v. Apple, Inc.*[3]). Based on the redacted transcript, it appears that the parties agreed that the A8 and A8X processors should be included in this case and that Apple would produce documents, source code, etc. for those processors (see page 21 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[4]). But Judge Crocker denied WARF's motion regarding the two future processors for apparently three reasons. First, Judge Crocker agreed with Apple that the fact that the future processors were not released meant "it's a moving target. Every day something new

and different happens" (see page 21 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[4]). Second, the future processors may not be released prior to trial (see page 22 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[4]). Third, including the future processors in this lawsuit might delay the trial date (see page 21 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[4]). Judge Crocker said that while he did not prefer to have "piecemeal" cases and that it would be more efficient to have all accused processors in a single trial, he had to "draw the line somewhere" and WARF could include the future processors in a second lawsuit, which WARF ultimately did (see pages 32, 20–21, 22, 32–33 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[4]).

With respect to the future processors, I agree with Judge Crocker's second and third reasons. While it may be more efficient to have a single trial for all accused processors, including the future processors at this stage in the case might delay the trial. But I disagree with his first reason that the design of future processors is a "moving target," which necessarily requires that Apple will need to repeatedly collect documents and could require that WARF change its infringement theories and expert reports to track Apple's on-going design changes. The reason for this is because an architectural-level feature, such as tracking potential dependences across memory accesses, is very unlikely to meaningfully change—let alone change enough to affect the patent infringement analysis—after the architectural design stage. Furthermore, the further into the design process the future processor is, e.g., register transfer level, verification, synthesis, physical design, design for test, tape-out, silicon testing, etc., the less likely that there will be *any* architectural-level changes, let alone patent infringement-relevant changes. For example, during layout, the designers may realize that a buffer may need to have one fewer (or can have one more) entry or that the access time might be a little higher. But none of those kinds of lower level changes would affect whether Claim 1 of the WARF's patent would be infringed:

1) In a processor capable of executing program instructions in an execution order differing from their program order, the processor further having a data speculation circuit for detecting data dependence between instructions and detecting a mis-speculation where a data consuming instruction dependent for its data on a data producing instruction of earlier program order, is in fact executed before the data producing instruction, a data speculation decision circuit comprising:
   - a predictor receiving a mis-speculation indication from the data speculation circuit to produce a prediction associated with the particular data consuming instruction and based on the mis-speculation indication; and
   - a prediction threshold detector preventing data speculation for instructions having a prediction within a predetermined range.

—U.S. Patent No. 5,781,752, Claim 1.

Rather, Claim 1 requires that there is a predictor and a prediction threshold detector that each operate in a particular way. Given that Claim 1 focuses on high-level components that are very unlikely to change after the architectural design stage, even if the future processor is still being designed, it is very unlikely that there will be any design changes that would require that Apple will need to repeatedly collect documents or could require that WARF change its infringement theories or expert reports to track Apple's on-going design changes. Rather, it is fairly likely that the only documents Apple would need to collect and produce are those that existed at the end of the architectural design stage. Similarly, it is also fairly likely that WARF would not need to change its infringement theory or its expert reports, as it is unlikely that there will be any meaningful changes at an architectural level that would change WARF's infringement analysis.

Finally, with respect to Judge Crocker's comment that the design is always changing, to the extent that future processors, e.g., an A9 processor, also use the data-dependency predictor,[a] that data-dependency predictor is likely to be the same—at least in terms of the components required by WARF's patent—as what the A8 processor uses. For example, while the A9 processor may use a more sophisticated predictor than the A8 processor, all that Claim 1 requires is "a predictor receiving a mis-speculation indication from the data speculation circuit to produce a prediction... based on the mis-speculation indication[.]" As such, as long as the predictor in the A9 processor receives a misspeculation indication from the data speculation circuit and bases its prediction on the misspeculation indication, for the purposes of infringement, the predictor in the A9 processor would operate in the same infringing manner as the predictor in the A8 processor.

Similarly, while the A9 processor may use a more sophisticated prediction threshold, e.g., one that dynamically varies, the fact that both the A8 and A9 processors might have a prediction threshold means that, for the purposes of infringement, the prediction

[a]This article uses the term "data-dependency predictor" to refer to the accused feature in the accused products.

threshold in the A9 processor would operate in the same infringing manner as the prediction threshold in the A8 processor.

As such, rather than the design constantly changing as Judge Crocker believed, the high-level designs may be effectively identical, only differing by their implementation. Given that the asserted patent is not limited to a particular implementation, the implementation differences of the data-dependency predictor in the A8 and A9 are irrelevant. In that case, including the future A9 processors into WARF's original lawsuit probably will not delay the trial.

MOTION TO COMPEL TO PRODUCE SPECIFIC VERSIONS OF ACCUSED PROCESSORS

As described in Part VI of this series, WARF filed a motion to compel to require Apple to produce an executable version of the simulator for the Accused Products (A6, A7, and A8 processors). The reason WARF did this is so that they could measure the performance benefit of the asserted patent in those processors. With respect to this issue, Part VI of this series commented that "Apple never appeared to suggest that, as an alternative to producing an executable version of the simulator (which it also argued was not relevant and was an undue burden), that WARF should run the workloads on actual silicon of the Accused Products. This tends to indicate that there was no way to turn off the data-dependency prediction feature in the Accused Products, and thus there would be no way to measure the speedup of the patent."[5] But that speculation was incorrect.

On 20 February 2015, WARF filed its next motion to compel (second of the three remaining motions to compel). In this motion, WARF requested that the Court order Apple to produce specific versions of the Accused Processors in order to measure the performance benefit of the Accused Products (see page 1 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[6]). More specifically, WARF stated that it tried to get the performance information, first by asking, via an "interrogatory" (i.e., a written question where the answer is under oath), whether there was a way to disable to the data-dependency predictor (see page 1 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[6]). WARF then stated that "[f]our months later, Apple identified only a technical manual describing a technique it knew WARF would be unable to use due to Apple's security controls (but neglected to disclose this to WARF)" (see page 1 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[6]). Rather, according to WARF, "***only Apple*** has the security keys to permit the [data-dependency predictor] to be flipped" (see page 1 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[6]) (emphasis in WARF's brief).

When WARF asked Apple to produce sample devices that had the data-dependency predictor disabled, Apple stated that such devices "do not presently exist at Apple" (see page 1 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[6]). But after deposing Apple's engineers, WARF learned that Apple had several such devices that its engineers "routinely used" to "perform exactly the kinds of experiments WARF seeks to run" (see page 2 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[6]). WARF argues that, despite those facts, "Apple *still* refused to let WARF inspect and test the devices on the familiar ground that it would be too burdensome" (see page 2 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[6]) (emphasis in WARF's brief).

To support its motion, WARF retained an expert, Prof. Glenn Reinman of the University of California, Los Angeles, to provide an expert declaration (see page 1 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[7]). While portions of his declaration are redacted, Prof. Reinman appears to have tried to disable the data-dependency predictor on a commercial iPhone, but was unable to do so given that the particular variable "resides in a protected portion of memory controlled by the operating system" (see paragraph 9 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[7]).

WARF requested that the Court order Apple to do one of three[b] alternatives (see page 24 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[6]). Unfortunately, due to the redactions in WARF's brief, Prof. Reinman's declaration, Apple's brief, and Apple employee Peter Bannon's declaration, it is very difficult to determine what the three alternatives are. That said, reading between the lines, the first alternative may be that Apple provides their typical test setup, that includes a JTAG interface, so Prof. Reinman can "plug a cable directly into the board/device in which the chip is installed and interact directly with the processor" (see paragraph 17 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[7], and page 24 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[6]). The second alternative may be that Apple provides the iOS source code and its compiler so that Prof. Reinman (or alternatively Apple itself) can manually change iOS in order to create a "side build" of iOS that disables the data-dependency predictor (see paragraph 12 of *Wis. Alumni Rsch.*

[b]Apple describes WARF's requests as only being two alternatives, which could mean the first and third alternatives are either similar or overlap, or are otherwise combined (see page 1 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]).

Found. v. Apple, Inc.[7], and page 24 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[6]). The third alternative may be that Apple provides devices where the data-dependency predictor is somehow disabled and possibly where iOS has not been installed (see paragraph 10 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[7], and page 24 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[6]).

In its response, Apple argues that it provided WARF with "substantial performance-related discovery," including "all results that it could locate of the simulations or experiments created by Apple engineers during the development of the accused A7, A8, and A8X processors," "emails or other documents" discussing those results, source code for those processors, depositions of the Apple engineers who designed the accused feature, etc. (see pages 1, 6–8 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]).

Apple makes three arguments against WARF's requests that the Court should order Apple to undertake one of the alternatives. First, Apple argued that WARF's requests would require Apple to create documents or things (see pages 1, 6–8 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]). In other words, Apple argued that because none of the things that WARF requests exists, Apple would have to create them, which is beyond the scope of what is required under the Federal Rules (see page 10 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]). Furthermore, Apple said that WARF's requests are significantly beyond what courts have granted in other cases (see page 13 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]).

Second, Apple asserted that creating and producing what WARF requests would impose an "undue burden" on it (see page 14 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]). In particular, Apple stated that it would be "deprived [of] valuable engineering time" in order to meet WARF's requests and to monitor "WARF's use of the equipment due to security risks presented by their production" (see pages 14–17 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]).

Third, Apple said that WARF "greatly overstates the relevance of additional performance data" (see page 17 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]). In particular, as described above, Apple argued that it already produced "all simulations or experiments created by Apple engineers during the development of the accused A7, A8, and A8X it could locate" (see page 18 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]). Apple also argued that WARF already questioned Apple's engineers about the performance aspects of the accused feature in depositions (see page 18 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]).

In addition, Apple argued that "testing performed on the requested devices would be of limited relevance" (see page 19 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]). More specifically, Apple argued that the modifications that WARF products may have "unknown and unintended effects" that may affect the test results (see page 19 of *Wis. Alumni Rsch. Found. v. Apple, Inc.*[8]).

On 11 March 2015, Judge Crocker held a telephonic hearing for this motion (see *Wis. Alumni Rsch. Found. v. Apple, Inc.*[9]). The meeting minutes that were published on the docket state that Judge Crocker "granted the motion in the manner and for the reasons stated during the hearing" (see *Wis. Alumni Rsch. Found. v. Apple, Inc.*[9]). Unfortunately, the redacted version of the hearing transcript is not publicly available, so Judge Crocker's reasons are unknown.

In general, I agree with Judge Crocker's decision. Measuring the exact performance impact of the data-dependency predictor in the Accused Products is critically important to determining the damages due to Apple's alleged infringement of the asserted patent. Obviously, if the data-dependency predictor has a speedup of 20%, then it is more valuable than if it provides a speedup of 2%. While the damages may not be 10× higher with the former as compared to the latter, the former should be significantly more valuable. Therefore, in order to provide the jury with accurate data upon which they can make an educated decision regarding the damages for Apple's alleged infringement of the asserted patent, it is necessary to measure the performance of the data-dependency predictor in the accused products.

With respect to Apple's argument that the modifications WARF requested may affect the test results, I do not believe that the modifications that WARF requested are likely to affect the test results at all, let alone significantly. More specifically, WARF's requested modifications only changes one parameter: whether the data-dependency predictor is on or off. As such, given that there is a single change, the likelihood that the test results may be affected is low. Furthermore, WARF and Apple can compare WARF's test results to the results that Apple generated while designing the accused processor to "sanity check" whether WARF's test results are reasonable. Finally, this kind of test is exactly what engineers typically do to measure the performance of a particular feature.

With respect to Apple's arguments that creating and producing what WARF requests is an undue burden, I disagree for at least two reasons. First, creating a side version of iOS—one where the only difference from public releases of iOS versions is that it disables the data-dependency predictor—seems to be incredibly easy for Apple's iOS software engineers to do. Second, Apple's arguments regarding security appear to

be exaggerated, as WARF's testing would likely occur in the same place that WARF reviewed Apple's source code. Given that Apple does not appear to have significant issues with WARF reviewing its source code in that location, Apple likewise should not have any significant issues with WARF testing specific devices with the data-dependency predictor turned off or on in the same location.

The next article in this series will continue to examine what happened in this case.

REFERENCES

1. Wis. Alumni Res. Found. v. Apple Inc, No. 3:14-cv-00062, ECF No. 61 (W.D. Wis. Oct. 10, 2014).
2. Wis. Alumni Res. Found. v. Apple Inc, No. 3:14-cv-00062, ECF No. 66 (W.D. Wis. Oct. 10, 2014).
3. Wis. Alumni Res. Found. v. Apple Inc, No. 3:14-cv-00062, ECF No. 71 (W.D. Wis. Oct. 10, 2014).
4. Wis. Alumni Res. Found. v. Apple Inc, No. 3:14-cv-00062, ECF No. 832 (W.D. Wis. Aug. 11, 2025).
5. J. J. Yi, "A review of Wisconsin Alumni Research Foundation v. Apple—Part VI," *IEEE Micro*, vol. 45, no. 5, pp. 134–137, Sep./Oct. 2025, doi: 10.1109/MM.2025.3619665.
6. Wis. Alumni Res. Found. v. Apple Inc, No. 3:14-cv-00062, ECF No. 82 (W.D. Wis. Feb. 20, 2015).
7. Wis. Alumni Res. Found. v. Apple Inc, No. 3:14-cv-00062, ECF No. 84 (W.D. Wis. Feb. 20, 2015).
8. Wis. Alumni Res. Found. v. Apple Inc, No. 3:14-cv-00062, ECF No. 94 (W.D. Wis. Mar. 4, 2015).
9. Wis. Alumni Res. Found. v. Apple Inc, No. 3:14-cv-00062, ECF No. 97 (W.D. Wis. Mar. 11, 2015).

JOSHUA J. YI is a solo practitioner at The Law Office of Joshua J. Yi, PLLC, Austin, TX, 78750, USA. Contact him at josh@joshuayipatentlaw.com.

EDITOR: R. Iris Bahar, ribahar@mines.edu

DEPARTMENT: MICRO ECONOMICS

Private Returns on Technology Adoption

Shane Greenstein, *Harvard Business School, Boston, MA, 02163, USA*

From a firm's perspective, the emergence of a new technology wave is a new opportunity to generate a financial return. The question is precisely how. That topic remains as salient today in the era of artificial intelligence as it was in the era when firms first encountered smartphones, the commercial Internet, and personal computers.

Before we fully embrace this new era, let me suggest that we review lessons from the most recent era of technology adoption. In particular, let's focus on consumer computer technologies (CCTs)—the mix of the mobile ecosystem and the widely used Internet, enhanced by Web 2.0.

Some research partners and I recently analyzed returns from co-invention in CCTs. By co-invention, we mean the invention of new applications by firms that utilize CCTs in their business. That investment typically involves developing business processes and practices to complement the adoption of CCTs and support the introduction of new services and business models.

This column provides a somewhat brief overview of high-level points from the study. Some of the implications arising from these points might seem obvious, but it was a surprise to us that all the consequences would come from the same framework. For more information, please see the reference at the end of the column.

FRAMEWORK

Consider a straightforward model of the two types of projects undertaken by firms, where one utilizes the new technology as an intermediate input and involves *incremental* co-invention. The other also uses the technology as an intermediate input, but requires something more ambitious and innovative for the firm. Referred to by many names in everyday speech, for brevity, we will label it as *novel* co-invention.

Incremental co-invention in CCTs was exceedingly common. That is because, when faced with a low-cost expansion of existing services, most firms choose to invest in it. For example, should a firm build an app? This was a comparatively straightforward economic decision, especially when the benefits were directly measurable in terms of web traffic that enabled a marketing funnel into sales leads or ad revenue. Accordingly, most firms did build apps.

Novel co-invention differed. Success was difficult, costly, and rare, but you knew a successful novel co-invention effort when you saw it. Many users found the service compelling. It established a new category of uses or altered leadership in an existing end-market. Think of Netflix in its early years, or the elaborate efforts required by your pharmacy to send you text reminders to refill a prescription.

For a firm involved in novel co-invention, adopting CCTs was usually less straightforward, often because the payoffs were several years out from the initiation of investment. For example, when Netflix first began transitioning from mail-order DVD rentals to streaming, they had to confront considerable uncertainty about how to do so at scale. It took them more than seven years to find a predictable and reliable process.

SOME OF THE IMPLICATIONS ARISING FROM THESE POINTS MIGHT SEEM OBVIOUS, BUT IT WAS A SURPRISE TO US THAT ALL THE CONSEQUENCES WOULD COME FROM THE SAME FRAMEWORK.

In an entrepreneurial setting, the economic constraints bind in different ways. For example, should an entrepreneur build an app for a use case that no other firm covers? Consider Snap, who introduced ephemeral messaging with Snapchat. This novel social experience drove rapid adoption among young users, creating network effects that fueled explosive early growth before competitors caught up with similar features. Indeed, Snap reached a place that few entrepreneurs ever attain. Alas for them, their time at the top was brief.

Digital Object Identifier 10.1109/MM.2025.3615287
Date of current version 24 December 2025.

Here is my point: Novel co-invention was risky in the CCT era. Private returns were uncertain. Yet, as we all know, many firms did build these. Some succeeded, and most failed, sometimes spectacularly.

IMPLICATIONS

What does all that imply? For one, comparisons between investors in incremental co-invention should reveal a specific pattern of returns, one proportional to the preexisting customer revenue.

Here is what that means. If one observes a single industry as it evolves into an online market over time, and if every firm made incremental changes, then everybody would have received a 2%, 3%, or 4% return, and nobody should miraculously gain a 50% return on their website.

We tested that prediction. We rolled up our sleeves and collected information from the Internet Archive about the online experiences of terrestrial radio and newspapers, in each case in 2013. We compared that with their listenership in 1993, 20 years earlier. In radio, we were able to track the experiences of over 2000 stations. In newspapers, we examined approximately 100.

The framework predicts that radio stations with more listeners in 1993 would have a larger online audience in 2013 than stations that started with a smaller number of listeners in 1993. That is what we see. In other words, radio appears to be an industry without any novel successful co-invention as of 2013. We make a similar prediction for the audience size for newspapers, but see a slightly different outcome. Newspapers with larger circulations gained a more than proportionate online readership compared to those with smaller circulations.

COMPARE INCREMENTAL AND NOVEL

Here is another implication: Incremental co-invention should be as widespread as the businesses that use it. In contrast, the returns on novel co-invention should be skewed with only a few big winners and many losers. Novel co-inventions should also be tied to the labor market for technical talent or industry specialties.

Testing this contrast is no small task. It requires an examination of all industries. We rolled up our sleeves again and found a way to collect data on private returns for 2400 leading online properties in 2013, roughly split between smartphone apps and websites. Through extensive (and sometimes tedious) work, we classified all of these into incremental and novel co-invention efforts that led to their production. Some of these industries are so new that we also had to build a new product classification to capture the main categories in the data.

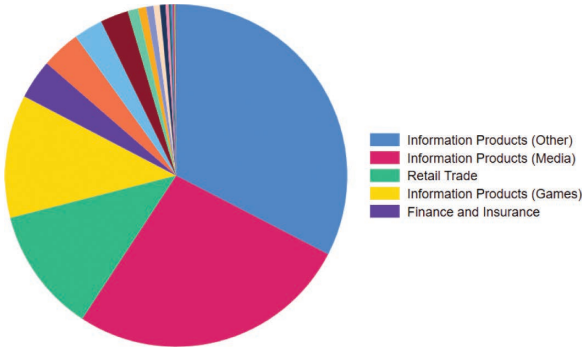


FIGURE 1. Incremental co-invention distribution across industries.

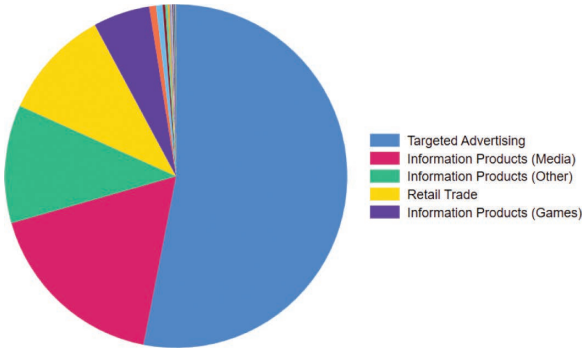


FIGURE 2. Novel co-invention distribution across industries.

Figures 1 and 2 illustrate the distribution of value creation across firms. These should differ between incremental and novel innovations. Indeed, we find that it does. Incremental co-invention grows from firm assets that support existing business, while novel co-invention grows from an entirely different origin. Local labor markets with appropriate technical and commercial domain knowledge supported investment in novel uses of CCTs.

We then examined a related, open question. There is no theoretical reason why the total value created by novel co-inventors should be bigger or smaller than that made by incremental co-inventors. A small number of big hits could exceed the total of a large volume of small contributions, or vice versa. Only data can settle the question.

As it turned out, the aggregate private value created by incremental co-invention is smaller than that made by novel co-invention, and by a considerable amount. Empirically, we estimate that incremental co-invention accounts for approximately 6% of the total observed value among the top 2400 firms. If every

bit of value we do not observe in the long tail of small firms is attributed to some incremental investment, it still accounts for less than 18% of the total value.

In other words, novel co-invention drove the vast majority of new private value creation in CCTs, and in a small set of industries where novel co-invention thrived; like it or not, Google, Facebook, Netflix, Amazon, and other top hundred CCT-based companies owned a significant portion of the profits from adapting CCTs.

To be clear, sometimes these companies came up with the novel co-inventions themselves (e.g., the Google Search Engine and Ad network), and sometimes they bought parts of it (e.g., Google bought YouTube after they tried and failed to develop something similar), absorbing it in the big company. Sometimes these received enormous investments, and sometimes these purchases went nowhere.

That result begs questions about the origins of why novel co-inventions sometimes originate in entrepreneurial firms and sometimes within leading firms. That is a bigger topic for another day. Today's column focuses on the lessons from characterizing these fact patterns.

LIKE IT OR NOT, GOOGLE, FACEBOOK, NETFLIX, AMAZON, AND OTHER TOP HUNDRED CCT-BASED COMPANIES OWNED A SIGNIFICANT PORTION OF THE PROFITS FROM ADAPTING CCTS.

CONCENTRATION

There is one more subtle prediction. If all co-inventions were incremental, there would be little impact on competition and market structure. Similarly, the industrial and geographical distribution of economic activity would change little, as they are based on the existing choices of firms and their existing assets. However, novel co-inventions should not be widespread. It should concentrate on a few industries and in a few places.

As expected, four related media and entertainment industries, as well as retail trade, collectively account for 97% of the value from novel co-invention, with targeted advertising being the most significant. Most of the manufacturing and several additional significant sectors, such as health care, are absent.

To be sure, incremental co-invention is more widespread industrially. However, it is also less impactful in creating value.

Novel co-invention should also concentrate geographically. To check this, we undertook another forensic investigation into the location of the co-inventing firm's headquarters for each product.

Again, as expected, we find that the location of incremental and novel co-invention differs. Because incremental co-invention builds on existing business, it is geographically distributed widely, following the existing location patterns of the co-inventing firms. In contrast, the geographic outcome of novel co-invention differs sharply, with four prominent regions emerging for novel innovation: the greater San Francisco region, New York, Seattle, and Los Angeles.

San Francisco and Seattle are no surprise, as they have long been centers for technical talent, and many firms there zig-zagged their way into media and entertainment markets. New York and Los Angeles firms had the opportunity to join them during this technology wave because their cities possessed the entertainment industry's human capital, and many firms there had to zig-zag their way into technology.

CONCLUSION

We are not the first commentators to notice the bifurcation of results in the commercial Internet. However, most commentators emphasize the winner-take-all competitive dynamics that result in a small number of massive providers.

We took another approach. You might call it reductionist for eschewing elaborate explanations about why things happened, but simple has advantages too. It boils results down to either/or, incremental/novel, which makes it easy to observe. If that trend continues into the new AI era, expect to be able to track project outcomes this way.

REFERENCE

1. T. Bresnahan, S. Greenstein, and P.-L. Yin, "New economic forces behind the value distribution of innovation," Nat. Bur. of Econ. Res., Cambridge, MA, USA, Working Paper 34090, 2025. [Online]. Available: https://www.nber.org/papers/w34090

SHANE GREENSTEIN is a professor with Harvard Business School, Boston, MA, 02163, USA. Contact him at sgreenstein@hbs.edu.

IEEE COMPUTER SOCIETY CAREER CENTER

Career Accelerating Opportunities

Explore new options—upload your resume today

careers.computer.org

Changes in the marketplace shift demands for vital skills and talent. The **IEEE Computer Society Career Center** is a valuable resource tool to keep job seekers up to date on the dynamic career opportunities offered by employers.

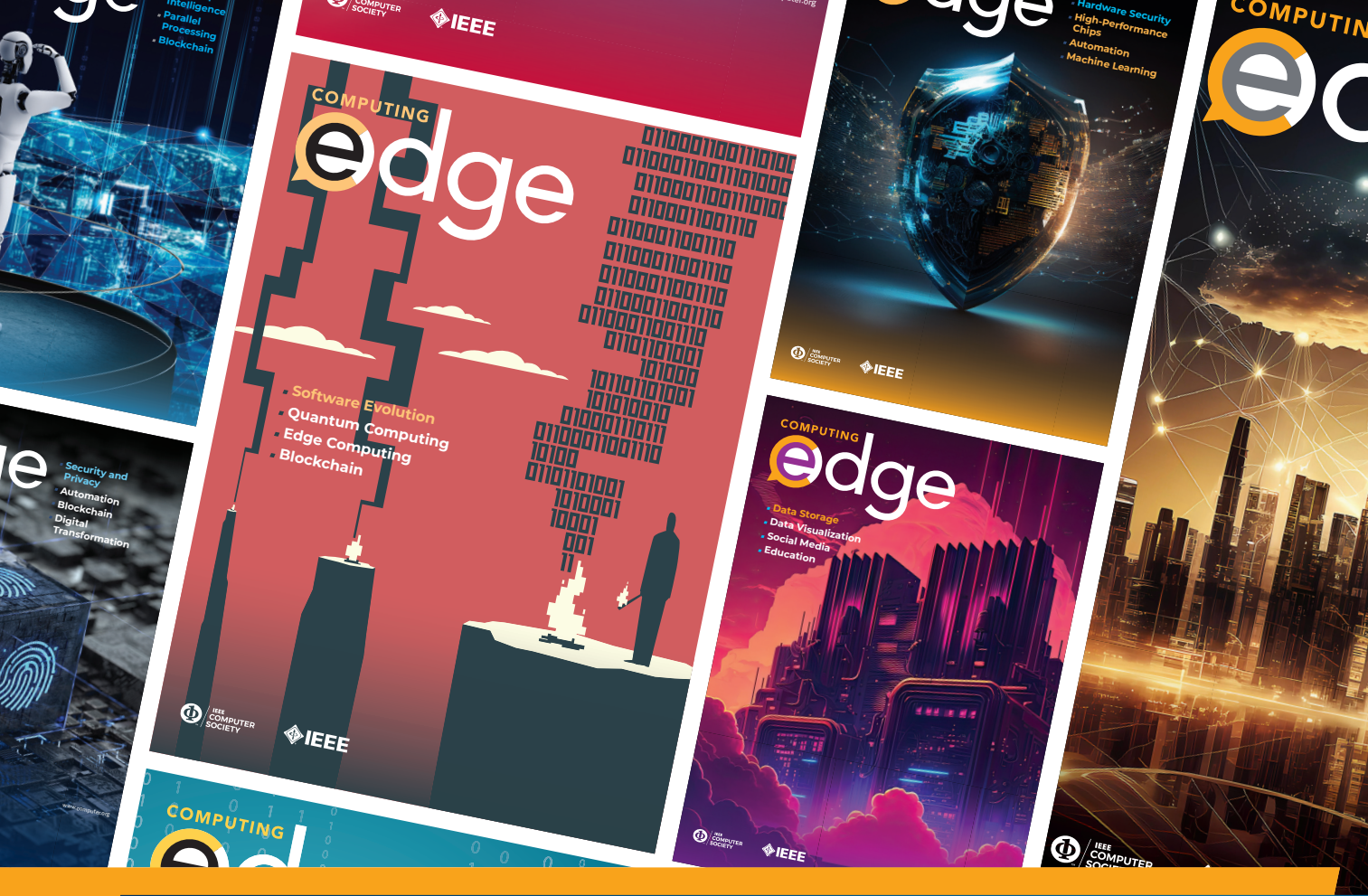
Take advantage of these special resources for job seekers:

- JOB ALERTS
- TEMPLATES
- WEBINARS
- CAREER ADVICE
- RESUMES VIEWED BY TOP EMPLOYERS

No matter what your career level, the IEEE Computer Society Career Center keeps you connected to workplace trends and exciting career prospects.

IEEE COMPUTER SOCIETY

IEEE



ComputingEdge

Your one-stop resource for industry hot topics, technical overviews, and in-depth articles.

- Cutting-edge articles from the IEEE Computer Society's portfolio of 12 magazines.
- Unique original content by computing thought leaders, innovators, and experts.
- Keeps you up to date on what you need to know across the technology spectrum.

Subscribe for free

www.computer.org/computingedge

IEEE COMPUTER SOCIETY

IEEE