Context-aware Prefetching for Near-Storage Accelerators

Jian Zhang (Rutgers), Marie Nguyen (Samsung), Sanidhya Kashyap (EPFL)
Sudarsun Kannan (Rutgers)

Abstract

We present ContextPrefetcher, a host-guided high-performant prefetching framework for near-storage accelerators that prefetches data blocks from storage (e.g., NAND) to device-level RAM. Efficiently prefetching data blocks to device-level RAM reduces storage access costs and improves I/O performance. We introduce a novel abstraction, *Cross-layered Context (CLC)*, a virtual entity that spans across the host and the device and is used for identifying, managing, and tracking active and inactive data such as files, objects (within object stores), or a range of blocks. To support efficient prefetching of actively used CLCs to device memory without incurring near-device resource (memory and compute) bottlenecks, ContextPrefetcher delegates prefetching management to the host, guiding near-device compute to prefetch blocks of active CLC. Finally, ContextPrefetcher facilitates the swift reclamation of blocks associated with inactive CLC. Preliminary evaluation against state-of-the-art near-storage accelerator designs demonstrates performance gains of up to 1.34×.

ACM Reference Format:
Jian Zhang (Rutgers), Marie Nguyen (Samsung), Sanidhya Kashyap (EPFL), Sudarsun Kannan (Rutgers). 2024. Context-aware Prefetching for Near-Storage Accelerators. In *16th ACM Workshop on Hot Topics in Storage and File Systems (HOTSTORAGE '24), July 8–9, 2024, Santa Clara, CA, USA*. ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3655038.3665956

1 Introduction

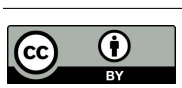
To reduce the cost of data movement from storage devices to compute, hardware companies have started integrating computational capabilities directly into storage devices, creating Computational Storage Devices (CSDs). These devices, such as Newport, SmartSSD, and ScaleFlux [5, 14, 15], perform tasks near where data is stored, enhancing system efficiency and performance. Furthermore, the emergence of high-speed communication protocols like Compute Express Link (CXL) is expected to enhance near-storage processing capabilities by enabling applications and the host compute to access near-storage memory directly. Despite their in-storage processing capabilities, for cost ($/GB) and power reasons, these devices are expected to host low-power wimpy compute cores and relatively smaller in-storage DRAM.

Despite these advancements in storage technology, optimizing data access remains challenging, particularly considering the substantial performance differences between the main memory and storage. Caching and prefetching techniques have been extensively explored to reduce data access costs in traditional storage, and the techniques could be applied to accelerate data access for near-storage accelerators.

More broadly, caching and prefetching techniques can be classified into systems focusing on (1) host-level caching by prefetching data from the device to the host and (2) device-level caching by prefetching data from NAND flash to the device memory. Host-level caching systems, such as the OS page cache, prefetch data from the device to the host to reduce application stalls in case of cache misses but suffer from high kernel software overheads and do not utilize device-level memory resources.

Recent state-of-the-art near-storage caching solutions (e.g., OmniCache [19]) have explored the benefits of utilizing device-level RAM as a parallel cache to speed up data access and data processing. However, these designs assume a large device-level DRAM (henceforth referred to as device-DRAM) for caching and employing compute-heavy caching policies (e.g., LRU-based caching), which is often impractical in real-world near-storage technologies that must also use the device-DRAM for storing FTL's logical to physical block mapping, wear-leveling, and as request transit buffer. Further, these approaches **lack support for near-storage data prefetching**, which is critical for expediting host-level I/O (e.g., `read, write`) and prefetching (e.g., `readahead`) operations for minimizing disk access. Replicating prefetching techniques using host OS caches [8] or application/runtime caches [2, 10] is infeasible due to restrictive memory and compute limits. Orthongally, prefetching has also been explored for memory expansion technologies like CXL SSDs [16] with near-device memory that prefetches next N-lines or blocks or techniques that use computationally-heavy ML training models (ExPAND [9]).

Unfortunately, across all the aforementioned prior work, besides resource overheads, these techniques fail to capture the context of which files, objects, or a range of blocks an application is accessing. This leads to imprecise prefetching and



HOTSTORAGE '24, July 8–9, 2024, Santa Clara, CA, USA

ACM ISBN 979-8-4007-0630-1/24/07.
https://doi.org/10.1145/3655038.3665956

higher cache pollution, leaving considerable performance benefits on the table.

Therefore, we propose **ContextPrefetcher,** a host-guided prefetching design for near-storage accelerators. The **key insight** of ContextPrefetcher is that the host (OS or runtime) creates a context using an actively-used file or object or block range for the device to prefetch to its RAM, termed as *Cross-layered Context (CLC)* using a prefetching logic (e.g., access pattern). The device uses CLC to prefetch without requiring substantial memory or compute resources to manage caching and prefetching. While we focus on ContextPrefetcher for exclusive caches maintained at the host and the device (e.g., OmniCache [19]), we envision ContextPrefetcher to support inclusive caches too (e.g., λ-IO [17]).

A CLC can represent various data entities, such as files, objects (in object stores), or fine-grained block ranges accessed by applications, aiding efficient prefetching. CLC facilitates the host in guiding the device on when and what to prefetch or evict without imposing substantial computational overhead. For example, a CLC enables the host to prioritize prefetching of active files used by applications or evicting cached blocks of inactive files.

Because applications can switch between files where an inactive file becomes active or vice versa, we take inspiration from CPU thread contexts to switch in and out the CLC of active and inactive files, respectively. Inactive CLC (s) are saved to a host data structure and restored when they become active again; if needed, their blocks are prefetched. This enables efficient use of device-DRAM to store only needed prefetch metadata and data and timely cache space reclamation.

Our preliminary evaluation of ContextPrefetcher on exclusive cache design, conducted by implementing our design on the prior state-of-the-art near-storage design [19] on a microbenchmark (Section 2.4), demonstrates up to 1.34x performance gains. In the remainder of the paper, we detail our preliminary design and discuss the challenges that must be addressed to fully realize ContextPrefetcher.

2 Background and Motivation

2.1 Host-level I/O Caching and Prefetching

Numerous prefetching techniques propose to enhance I/O performance at the host level. OSes like Linux offer page cache to expedite I/O performance according to access patterns. Linux also provides prefetching system calls like *readahead*, enabling applications to manage prefetching by specifying offsets and bytes to prefetch. Additionally, Leap [2] introduces an online, majority-based prefetching algorithm designed to boost the page cache hit rate. Lynx [10] presents a learning-based SSD prefetching mechanism that captures random access patterns utilizing Markov chains.

However, these approaches fail to utilize device memory resources and are not directly applicable to near-storage accelerators. Host-level prefetching operations, such as those in the Linux OS, result in high compute and memory overheads. For instance, the Linux OS requires maintaining a per-inode Xarray [7, 8], which could significantly impact performance when applied to CXL-connected storage with near-storage processing capabilities and strict memory constraints [16]. File system caches typically reside within the kernel, necessitating kernel involvement for prefetching and incurring associated software overheads. For example, each *readahead* call incurs system call and kernel trap overheads.

2.2 Near-storage Processing

While modern solid-state and nonvolatile memory storage devices have significantly improved I/O performance, software and hardware data access costs remain high. To tackle this challenge, hardware vendors have started integrating computational capabilities directly into storage devices, a concept known as Computational Storage Devices (CSDs). By executing tasks near the storage medium, CSDs enhance system efficiency, reduce data movement, and amplify performance. Several companies have introduced various CSD solutions, including Newport CSDs [5], FPGA-based solutions like SmartSSD [6], and ScaleFlux CSDs [15]. Additionally, the emergence of Compute Express Link (CXL) has notably advanced the capabilities of near-storage computing elements. If designed well, CXL can enable high-speed communication between CPUs and near-storage accelerators, facilitating direct access to the device-level RAM. However, for near-storage devices, hardware resources are limited. For instance, the processor frequency is significantly lower than the host CPU, typically around 1.2-1.8 GHz [3, 5]. Similarly, DRAM resources are constrained, ranging from 1-4GB [4–6]. Moreover, DRAM cannot be fully utilized for acceleration purposes as it needs to be reserved for internal tasks or software. For example, the BlueField SmartNIC [11] installs a Linux OS, which requires substantial memory resources.

Near-storage Caching: For near-storage accelerators, device-memory caching has been explored recently. For instance, OmniCache [19] presents a horizontal (exclusive) caching across the host and the near-storage memory. However, OmniCache lacks support for prefetching and efficient eviction of the near-storage cache. Supporting efficient prefetching and eviction for near-storage devices presents several challenges due to their limited processor and memory resources. First, existing designs assume large internal device memory, which is impractical given that the device memory must be used for internal storage management and block allocations like FTL. Therefore, ContextPrefetcher optimizes device memory use by prefetching and only retaining data in active use through CLC. Second, adopting host-level caching and prefetching techniques for near-storage is not feasible. Host-level caching designs rely on high-performance CPUs with complex prefetching logic. For example, traversing the per-file Xarray is expensive for near-storage with limited

compute resources. ContextPrefetcher mitigates device compute overheads by delegating the expensive prefetching logic to the host. Third, traditional host-level caching designs employ FIFO/LRU or similar eviction strategies that are ineffective for small device memory caches and can lead to frequent cache evictions. Fourth, caching policies such as exclusive caching (e.g., OmniCache) and inclusive caching (e.g., λ-IO) are feasible with near-storage cache. ContextPrefetcher operates independently of caching design, compatible with both exclusive and inclusive cache designs.

Near-storage Prefetching: λ-IO [17] exploits host OS-level caches and prefetching by offloading processing for data not in the host cache. However, it overlooks the potential benefits of leveraging device-level caches and prefetching.

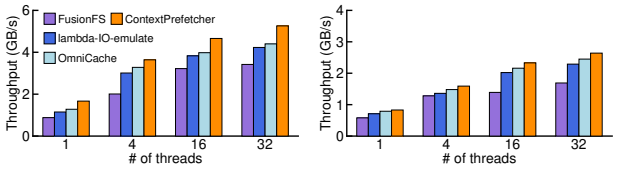
Shao-Peng et al.[16] propose utilizing device-DRAM for caching and prefetching 'next N cache lines' in CXL-based SSDs to reduce latency and increase endurance. Besides using LRU-based techniques for device memory management that incur high compute and memory overheads, these approaches lack host-level context. For example, an application thread could switch access to some other file(s) or block range(s), and a device without application or host context could continue to prefetch N lines and pollute the cache. ExPAND[9], an orthogonal approach aims to improve CPU prefetching for CXL-SSDs by employing ML-based prediction, which can demand a high computational cost for near-storage accelerators.

2.3 Limitations of state-of-the-art systems:

Next, we outline limitations observed in state-of-the-art systems, which serve as the motivation for our proposed prefetching techniques.

Lack of Prefetching for Device-DRAM: State-of-the-art systems lack prefetching for device-DRAM, impacting I/O performance. For instance, the OSes on the host file systems could either issue regular I/O or prefetch data from a file from the device to the host DRAM. For the data to be prefetched, the file's blocks must be first loaded to the device cache and then returned to the application. Besides being slow because the devices lack a context on what active file or object is accessed by an application, the device cache may cache currently inactive (not accessed) files, resulting in inefficient utilization of DRAM resources and failure to achieve peak I/O performance. An ideal approach would be to maintain a common context accessed by applications and accelerate prefetching around that context. We discuss the details of our proposed CLC in Section 3.2.

Overheads of Intricate and Expensive Prefetch Operations: Attempting to directly apply host-level prefetching designs onto device prefetching encounters feasibility challenges. The resource-intensive nature of host prefetching leads to performance degradation on devices with limited CPU resources. In addition to the cost of detecting access patterns for prefetching, managing, and iterating data structures



(a) Seq. Read **(b) Rand. Read**

Figure 1. Analysis of prefetching benefits. The figure shows the throughput of near-storage designs w/o caching and prefetching for sequential and random reads. Only ContextPrefetcher (last bar) supports prefetching for near-storage devices.

like per-inode Xarray used in OS page caches could impose significant CPU overhead. Likewise, device-level prefetching strategies, such as those employed for other technologies like CXL-SSDs, could impose high computing demands. For instance, ExPAND [9] relies on machine learning-based prefetching, where training and inference consume higher computational needs.

Efficient and Timely Eviction of Near-Device Caches is Challenging: State-of-the-art near-storage caching designs employ cache eviction policies used for systems with large amounts of memory and compute resources [19]. For example, these designs frequently rely on conventional block eviction policies such as LRU, iterating over all the blocks in the cache to identify the least recently used blocks for eviction. Unfortunately, these techniques lack the information (i.e., context) to determine whether they are evicting blocks of a file or object that are currently active or inactive. This results in two types of overhead. Storing inactive files or objects in capacity-constrained caches leads to cache pollution. Understanding if they are inactive (or active) helps quickly evict cache blocks and reclaim memory space in a timely manner without the need to iterate across all blocks using block-level LRU, thus saving compute cycles and admitting cache blocks of actively used files or objects. Similarly, it also helps to retain blocks of active files longer and increase prefetching, techniques lacking in today's near-device caches [16].

We demonstrate that CLC, equipped with context switch capabilities, offers an efficient solution for managing near-device memory.

2.4 Analysis

We compare ContextPrefetcher with the following designs: (1) FusionFS [18] (a near-storage file system without caching and prefetching support); (2) emulated λ-I/O without FPGA but with host-level OS caching and prefetching but fails to utilize near-device memory for caching or prefetching; (3) OmniCache, a recently proposed unified caching design for near-storage accelerators but lacks prefetching support. Due to a lack of programmable near-storage accelerators, we carefully emulate near-storage accelerators similar to prior designs. We use a machine with 512 GB DC Optane

NVM for storage, 64 CPUs, and 32 GB DRAM. For near-device computational units, we dedicate 4 cores with their frequency set to 1.2 GHz. Further, as used in prior studies [19], we throttle the memory bandwidth and add PCIe latency. The host-to-device interface is a block interface using NVMe commands.

For host caching, we allocate a total cache size of 32 GB of memory across all workloads. However, for device-level cache (for OmniCache), we limit the device cache size to 1 GB but maintain the same total cache size (32GB) across the host and the device. It is important to note that while OmniCache employs a larger device cache (up to 8GB), such assumptions are not always practical or restricted to high-end resources, specifically in scenarios where the device memory is shared for other operations inside a device, which include maintaining a file translation table, wear-leveling, handling transit I/O requests, and near-storage data processing. Therefore, we deliberately demonstrate the effects of a smaller cache size. Likewise, ContextPrefetcher employs the same device cache size as OmniCache. We designed a microbenchmark that simulates real-world applications for this study; we only consider read access patterns. Each thread concurrently opens multiple database SST files, performs sequential/random reads, and closes them after finishing using [1].

As shown in Figure 1, FusionFS exhibits poor performance due to the lack of caching and prefetching support. While λ-I/O outperforms FusionFS with host caching and prefetching, it suffers from significant kernel overheads because its cache resides within the OS. More importantly, λ-I/O does not leverage near-device caching or prefetching, resulting in spending substantial cycles for cache misses to fetch the data from storage. OmniCache shows similar performance when the near-device cache size is small and in the absence of near-device prefetching capability. On the other hand, the preliminary ContextPrefetcher design demonstrates considerable performance gains. This improvement is due to several factors. First, ContextPrefetcher supports efficient prefetching for device-DRAM, which reduces I/O overheads. Second, and importantly, ContextPrefetcher can efficiently switch between active and inactive CLC when accessing multiple files, minimizing cache pollution and retaining the most needed data in the device cache. Our analysis indicates that existing techniques lack the context of files that applications are currently accessing and, therefore, could retain inactive files in the cache (i.e., cache pollution), increasing cache misses. We will discuss the design details of ContextPrefetcher shortly.

3 Design

3.1 Design Goals

1. Host-guided Prefetching at the Near-storage Memory. Our primary objective is to establish a host-guided prefetching at the near-storage RAM to increase I/O prefetching efficiency by proactively prefetching data for actively used files or objects (context) to device-DRAM.

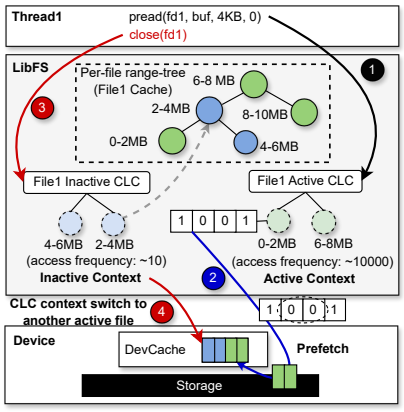


Figure 2. High-level Design. The figure shows ContextPrefetcher handling CLC context switch. ❶ The thread issues `pread` to read the data residing in the active CLC. ❷ ContextPrefetcher issues prefetching requests along with the block range metadata and a bitmap to indicate which blocks to fetch to the device cache or have already been fetched. ❸ The thread closes the current file. ❹ ContextPrefetcher switches to another active CLC for prefetching and reclaims cache space from inactive CLC in the device-DRAM due to space constraints.

2. Prioritize efficient device memory utilization by only maintaining data and metadata of active context. We introduce a novel prefetching abstraction: *Cross-layered Context*. This abstraction allows us to prioritize efficient device-DRAM usage by prefetching and retaining metadata and data for actively accessed files. We demonstrate the efficacy of our approach in achieving efficient prefetching and timely memory reclamation.

3. Avoid computing-intensive prefetching logic inside the device by delegating it to the host. We delegate the prefetching logic, such as access pattern detection as well as an expensive index or data structure, to track what and when to prefetch to the host, with the device simply performing the prefetch operations for missing blocks. This approach helps reduce near-device computational overheads and the metadata (e.g., index) memory overheads.

3.2 Preliminary Design

3.2.1 Cross-layered Context (CLC)

At its core, CLC is a virtual entity for grouping and managing active and inactive data, as well as prefetching status, spanning across both the host and the device. CLC abstraction facilitates seamless connectivity in prefetching activities between devices. Through CLC, we introduce *ContextPrefetcher*, a host-guided prefetching framework tailored for near-storage accelerators operating across the host and the device.

The rationale behind this design is simple yet powerful. By capturing accesses for active or inactive context right down to the device layer, caching and prefetching can be optimized for actively used CLC instances. Conversely, CLC identified as inactive (e.g., an open but inactively used) unused (e.g., a file is closed) file can be evicted or context-switched out based on the available device-DRAM or a specific policy.

As shown in Figure 2, we show an example of using CLC and the context-switching between active and inactive files. We draw inspiration from the CPU thread contexts to implement CLC context switch mechanisms. Much like how a CPU context switch efficiently manages multiple tasks or processes on a single physical CPU core, CLC context switch ensures optimal caching and prefetching for multiple files on limited device memory. It achieves this by selectively preserving prefetching metadata and data pertinent to actively used files on the device.

In essence, during a CLC context switch, only the blocks or a designated range of blocks associated with the current active CLC are retained in memory. Inactive or less frequently accessed CLC instances are swiftly replaced, potentially freeing up valuable memory space to support prefetching for new files. While in this paper, we focus on file-level contexts, CLC can be generalized to other contexts, such as object contexts in Object Storage Devices (OSD) or memory-context to memory-mapped regions, which will be our future work.

3.2.2 ContextPrefetcher Components

Our preliminary system design comprises a user-level runtime (LibFS) and near-device component. LibFS in the host manages caching and prefetching for both the host and the device. Our current implementation of the host component is a user-level library for quick prototyping, but our ongoing work also includes building support for integrated OS-level and device-level solutions.

To enable concurrent data access and caching across the host and the device, LibFS maintains a per-file range tree, serving as a unified index for managing caches across both domains. Each node within the range tree corresponds to a specific range of a file (each range size is 2 MB by default but is configurable). This index is crucial in locating cached data in both the host and device caches.

Regarding the near-device component, in contrast to earlier near-storage file system designs [12, 13, 18, 19], which require complex integration between file systems and firmware, our proposed ContextPrefetcher operates without mandating a file system on the device itself. The near-device component is only responsible for prefetching data based on the context passed by the host.

3.2.3 Using and Updating CLC

As shown in Figure 2, when a file is first created or opened, we associate a new empty CLC with the corresponding file, and it is marked as inactive. As we discussed in Section 3.2.2, CLC maintains a per-file range tree for caching indexing (both host and device).

Initially, in the absence of prefetching information, as an application reads a range of blocks, the blocks are first read to the device cache based on a policy used by prior work [19] to reduce data movement to the host. To track the blocks cached on the device, LibFS maintains a per-file range tree where the range represents a block range on the device cache. Each range node also stores a bitmap with one bit for each block in the range, indicating if the block is in the device cache. As blocks of a file are accessed, the host-level LibFS uses its prefetching logic to predict the next set of accesses and issues a prefetching request to the device using a special NVMe-like command that contains the CLC structure with the blocks to be prefetched. The device uses the CLC, creates an equivalent bitmap, prefetches the blocks, sets the corresponding bits on the bitmap upon prefetching, and resets them upon eviction. This bitmap, for instance, on a large 1 TB file would necessitate, at most, a 32 MB bitmap compared to hundreds of MBs for other complex data structures.

For the actual prefetching logic in the host, our current prefetcher can detect sequential, random, and strided access patterns, but we envision designing a more sophisticated prefetcher to capture different access patterns.

In addition to facilitating efficient prefetching, CLC also offers swift reclamation for inactive data. As shown in Figure 2, each file and a file's range tree's node is equipped with a frequency counter. For a file or a range within the file, if the frequency exceeds a certain threshold (a configurable parameter), we mark the file and the range as an active CLC and instruct the device to prefetch. Similarly, any inactive file or a range in a file is marked as an inactive CLC, the device CLC (a bitmap) is context switched out, and the cache blocks are reclaimed if needed. Our future work will focus on efficiently tracking active and inactive contexts, reducing frequent context switches, and potentially leveraging ML-based techniques using host-level computational resources.

In this paper, we focus on exclusive caching and prefetching design paradigms and only on device-level prefetching. Our ongoing future work will analyze and design coordinated host-device prefetching.

4 Discussion and Conclusion

This paper introduces a novel context-aware prefetching approach for near-storage devices. Our work demonstrates that traditional prefetching and eviction methods are inadequate for near-storage accelerators due to constraints on their computational and memory resources. Therefore, facilitating prefetching across devices necessitates establishing a unified and coordinated prefetching context (CLC), which helps delegate prefetching complexities to the host. In this

preliminary work, we focus on the general principles of pushing down application context (i.e., a file) to the device. We will explore how the host and the device can collaboratively prefetch the contents of different files or even different blocks of the same file. Similarly, our approach can be generalized to objects and memory-mapped ranges, which is important for other devices such as CXL-SSDs and memory expanders, areas we will focus on in our future work.

Acknowledgements

We thank Michio Honda (our shepherd) for his insightful comments to improve the quality of this paper. We also thank the anonymous reviewers and the members of RSRL for their valuable feedback. This research was supported by funding from Samsung (Memory Solutions Lab), and NSF CNS grants 1910593 and 2231724. This work was partially carried out on the experimental platform funded by NSF II-EN grant 1730043.

References

[1] [n.d.]. RocksDB. http://rocksdb.org/.

[2] Hasan Al Maruf and Mosharaf Chowdhury. 2020. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 843–857.

[3] ARM. [n.d.]. ARM CSD. https://www.arm.com/markets/storage/computational-storage.

[4] ARM. [n.d.]. Cosmos+ OpenSSD. http://www.openssd-project.org/cosmospl/overview/.

[5] Jaeyoung Do, Victor C. Ferreira, Hossein Bobarshad, Mahdi Torabzadehkashi, Siavash Rezaei, Ali Heydarigorji, Diego Souza, Brunno F. Goldstein, Leandro Santiago, Min Soo Kim, Priscila M. V. Lima, Felipe M. G. França, and Vladimir Alves. 2020. Cost-Effective, Energy-Efficient, and Scalable Storage Computing for Large-Scale AI Applications. *ACM Trans. Storage* 16, 4, Article 21 (Oct. 2020), 37 pages. https://doi.org/10.1145/3415580

[6] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. ACM, New York, NY, USA, 1221–1230. https://doi.org/10.1145/2463676.2465295

[7] Shaleen Garg, Jian Zhang, Rekha Pitchumani, Manish Parashar, Bing Xie, and Sudarsun Kannan. 2024. CrossPrefetch: Accelerating I/O Prefetching for Modern Storage. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 102–116.

[8] https://docs.kernel.org/core api/xarray.html. [n.d.]. Xarray Documentation. https://docs.kernel.org/core-api/xarray.html

[9] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. 2023. Cache in Hand: Expander-Driven CXL Prefetcher for Next Generation CXL-SSD. In *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*. 24–30.

[10] Arezki Laga, Jalil Boukhobza, Michel Koskas, and Frank Singhoff. 2016. Lynx: A learning linux prefetching mechanism for ssd performance model. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. IEEE, 1–6.

[11] Nvidia. [n.d.]. NVIDIA Mellanox BlueField SmartNIC. https://network.nvidia.com/files/doc-2020/pb-bluefield-smart-nic.pdf.

[12] Yujie Ren, Changwoo Min, and Sudarsun Kannan. 2020. CrossFS: A Cross-layered Direct-Access File System. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. USENIX Association, 137–154. https://www.usenix.org/conference/osdi20/presentation/ren

[13] Yujie Ren, Jian Zhang, and Sudarsun Kannan. 2020. {CompoundFS}: Compounding {I/O} Operations in Firmware File Systems. In *12th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 20)*.

[14] Samsung. [n.d.]. Samsung Key Value SSD. https://www.samsung.com/semiconductor/global.semi.staticSamsung_Key_Value_SSD_enables_High_Performance_Scaling-0.pdf.

[15] ScaleFlux. [n.d.]. https://scaleflux.com/.

[16] Shao-Peng Yang, Minjae Kim, Sanghyun Nam, Juhyung Park, Jin-yong Choi, Eyee Hyun Nam, Eunji Lee, Sungjin Lee, and Bryan S Kim. 2023. Overcoming the Memory Wall with {CXL-Enabled}{SSDs}. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. 601–617.

[17] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. 2023. {λ-IO}: A Unified {IO} Stack for Computational Storage. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. 347–362.

[18] Jian Zhang, Yujie Ren, and Sudarsun Kannan. 2022. FusionFS: Fusing I/O Operations using CISCOps in Firmware File Systems. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*. USENIX Association, Santa Clara, CA, 297–312. https://www.usenix.org/conference/fast22/presentation/zhang-jian

[19] Jian Zhang, Yujie Ren, Marie Nguyen, Changwoo Min, and Sudarsun Kannan. 2024. {OmniCache}: Collaborative Caching for Near-storage Accelerators. In *22nd USENIX Conference on File and Storage Technologies (FAST 24)*. 35–50.