Overcoming the Memory Wall with CXL-Enabled SSDs

Shao-Peng Yang, *Syracuse University;* Minjae Kim, *DGIST;* Sanghyun Nam, *Soongsil University;* Juhyung Park, *DGIST;* Jin-yong Choi and Eyee Hyun Nam, *FADU Inc.;* Eunji Lee, *Soongsil University;* Sungjin Lee, *DGIST;* Bryan S. Kim, *Syracuse University*

https://www.usenix.org/conference/atc23/presentation/yang-shao-peng

This paper is included in the Proceedings of the 2023 USENIX Annual Technical Conference.

July 10–12, 2023 • Boston, MA, USA

978-1-939133-35-9

Overcoming the Memory Wall with CXL-Enabled SSDs

Shao-Peng Yang *Syracuse University*
Minjae Kim *DGIST*
Sanghyun Nam *Soongsil University*
Juhyung Park *DGIST*
Jin-yong Choi *FADU Inc.*
Eyee Hyun Nam *FADU Inc.*
Eunji Lee *Soongsil University*
Sungjin Lee *DGIST*
Bryan S. Kim *Syracuse University*

Abstract

This paper investigates the feasibility of using inexpensive flash memory on new interconnect technologies such as CXL (Compute Express Link) to overcome the memory wall. We explore the design space of a CXL-enabled flash device and show that techniques such as caching and prefetching can help mitigate the concerns regarding flash memory's performance and lifetime. We demonstrate using real-world application traces that these techniques enable the CXL device to have an estimated lifetime of at least 3.1 years and serve 68–91% of the memory requests under a microsecond. We analyze the limitations of existing techniques and suggest system-level changes to achieve a DRAM-level performance using flash.

1 Introduction

The growing imbalance between computing power and memory capacity requirement in computing systems has developed into a challenge known as the memory wall [23, 34, 52]. Figure 1, based on the data from Gholami et al. [34] and expanded with more recent data [11, 30, 43], illustrates the rapid growth in NLP (natural language processing) models (14.1× per year), which far outpaces that of memory capacity (1.3× per year). The memory wall forces modern data-intensive applications such as databases [8, 10, 14, 20], data analytics [1, 35], and machine learning (ML) [45, 48, 66] to either be aware of their memory usage [61] or implement user-level memory management [66] to avoid expensive page swaps [37,53]. As a result, overcoming the memory wall in an application-transparent manner is an active research avenue; approaches such as creating an ML-centric system [45,48,61], building a memory disaggregation framework [36,37,52,69], and designing new memory architecture [23,42] are actively pursued.

We question whether it is possible to overcome the memory wall using flash memory — a memory technology that is typically used in storage due to its high density and capacity scaling [59]. While DRAM can only scale to gigabytes in capacity, a flash memory-based solid-state drive (SSD) is

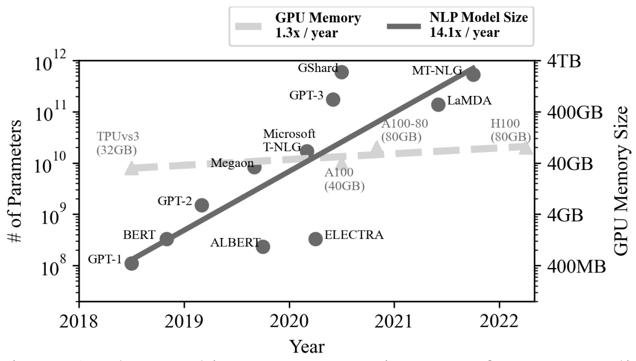


Figure 1: The trend in memory requirements for NLP applications [11, 30, 34, 43]. The number of parameters increases by a factor of 14.1× per year, while the memory capacity in GPUs only grows by a factor of 1.3× every year.

in the terabyte scale [23], a sufficiently large capacity to address the memory wall challenge. The use of flash memory as main memory is enabled by the recent emergence of interconnect technologies such as CXL [3], Gen-Z [7], CCIX [2], and OpenCAPI [12], which allow PCIe (Peripheral Component Interconnect Express) devices to be accessed directly by the CPU through load/store instructions. Furthermore, these technologies promise excellent scalability as more PCIe devices can be attached across switches [13] unlike DIMM (Dual Inline Memory Module) used for DRAM.

However, there are three main challenges to using flash memory as CPU-accessible main memory. First, there is a granularity mismatch between memory requests and flash memory. This results in a significant traffic amplification on top of the existing need for indirection in flash [23, 33]: for example, a 64B cache line flush to the CXL-enabled flash would result in 16KiB flash memory page read, 64B update, and 16KiB flash program to a different location (assuming a 16KiB page-level mapping). Second, flash memory is still orders of magnitude slower than DRAM (tens of microseconds vs. tens of nanoseconds) [5, 24]. As a consequence, while the peak data transfer rate between the two technologies is similar [4, 15], the long flash memory latency hinders sustained performance as data-intensive applications can only endure latency within the microsecond range at most [53]. Lastly, flash memory has limited endurance and wears out after repeated writes [24, 44]. This limits the usability of the memory technology as flash memory blocks beyond their endurance limit exhibit unreliable behavior and high levels of errors [44].

We address the above flash memory challenges by exploring design options, particularly those related to caching and prefetching, so that a CXL-enabled flash device (or CXL-flash) can be used to overcome the memory wall. Even though prior works have explored the scalability aspects of multiple CXL devices [36, 42] and have proven the feasibility of CXL-flash [9, 42], to the best of our knowledge, we are the first open-sourced in-depth study on the design choices of a CXL-flash device and on the effectiveness of existing optimization techniques. Due to the large design space, we first explore the CXL-flash hardware design in § 4 and then evaluate and analyze detailed policies and algorithms in § 5. We discover that it is possible to design a CXL-flash with 68–91% of its requests achieving less than a microsecond latency and an estimated lifetime of at least 3.1 years using memory traces of real applications. While exploring various designs and policies, we make seven observations which collectively indicate that modern prefetching algorithms are ill-suited to predict memory access patterns for the CXL-flash. More specifically, the virtual to physical address translation obfuscates access patterns for existing prefetchers to perform adequately. To counter this, we explore passing memory access hints from the kernel to the CXL-flash to further improve performance.

We make the following contributions with this work.

- We develop a novel tool that collects physical memory traces of an application, and we simulate the behavior of a CXL-flash with these traces. Both the memory tracing tool and the CXL-flash simulator (§ 3) are available at `https://github.com/spypaul/MQSim_CXL.git`
- Through synthetic workloads, we demonstrate the potential to effectively reduce the latency of a CXL-flash by integrating various system design techniques such as caching and prefetching, highlighting optimization opportunities. (§ 4)
- Using real-world workloads, we analyze the limitations of the current prefetchers and suggest system-level changes for future CXL-flash to achieve near-DRAM performance, specifically sub-μs latencies for the device. (§ 5)

2 Background

In this section, we first describe the opportunities presented by CXL (Compute Express Link) [3] as a representation of PCIe-based memory coherent interconnect technologies (which also include Gen-Z [7], CCIX [2], and OpenCAPI [12]). We then discuss the challenges of using flash memory with CXL.

2.1 Opportunities presented by CXL

CXL is a new interconnect protocol built on top of PCIe that integrates CPUs, accelerators, and memory devices into a single computing domain [42]. The main benefits of this integration are twofold. First, it allows coherent memory access between CPUs and PCIe devices. This reduces the synchronization overheads that are typically required for data transfers between the CPU and the device. Second, it is easy to scale the number of CXL devices: through a CXL switch, another set of CXL devices can be connected to the CPU.

Among the three types of devices that CXL supports, the Type 3 device for memory expansion is of interest to this work. Type 3 devices expose host-managed device memory (HDM), and the CXL protocol allows the host CPU to directly manipulate the device memory via load/store instructions [3]. While CXL currently only considers DRAM and PMEM as the primary memory expansion devices, it is possible to use SSDs, thanks to CXL's coherent memory access [42]. Moreover, the high capacity and better scaling of flash-based SSDs, enabled by stacking in 3D [59] and storing multiple bits in a cell [24], can effectively address the memory wall that modern data-intensive applications face. Inspired by previous works on CXL [36, 42], this paper studies the feasibility of using flash memory as a CXL memory expansion device.

2.2 Challenges with flash memory

We discuss the following three peculiarities of flash that make it challenging to use it as the system's main memory.

Granularity mismatch. Flash memory is not randomly accessible: its data are written and read at page granularity whose size is in the order of kilobytes [33], resulting in a large traffic amplification. Furthermore, a page cannot be overwritten. Instead, a block, which consists of hundreds of pages, must be erased first, and data can be written to only erased pages [33]. This restrictive interface causes any 64B cache line flush to incur a large write amplification through read-modify-writes. An SSD, as a block device whose access granularity is much larger (4KiB), has far less overhead.

Microsecond-level latency. Flash memory is orders of magnitude slower than DRAM, whose latencies are in the range of tens to hundreds of nanoseconds. The relatively faster flash memory read is still in the tens of microseconds, while the slower program and erase operations are in the hundreds and thousands of microseconds. Moreover, the flash memory latency also depends on their cell technology [24]. As outlined in Table 1 as an example, as more bits are stored per cell, the latency increases, from SLC (single-level cell) to TLC (triple-level cell). The ultra-low latency (ULL) flash is a variant of SLC that improves performance at the expense of density [46, 76]. Even the ULL technology, however, is orders of magnitude slower than DRAM. As a block device, microsecond-level latencies are tolerable due to the software

Table 1: Overview of memory technology characteristics.

Technology	Read latency	Program latency	Erase latency	Endurance limit
DRAM [50]	46ns	46ns	N/A	N/A
ULL [46, 76]	3μs	100μs	1000μs	100K
SLC [24]	25μs	200μs	1500μs	100K
MLC [24]	50μs	600μs	3000μs	10K
TLC [24]	75μs	900μs	4500μs	3K

overhead in the storage stack. However, for a memory device that is directly accessed using load/store instructions, microsecond-level latencies become a challenge.

Limited endurance. The high voltages applied to flash memory during the program and erase operations slowly wear out the cells, making them unusable over time [44, 72]. The memory manufacturers specify an endurance limit as a guide to how many times the flash memory block can be erased. This limit also depends on the flash technology, as shown in Table 1. While this is nevertheless a soft limit and flash memory can still be used beyond the limit [72], worn-out blocks exhibit unreliable behavior and are not guaranteed to correctly store data [44]. Due to application-level and kernel-level caching and buffering, the amount of writes to a block-interfaced SSD is reduced, and the current endurance limit is often sufficient during the SSD's lifetime. As a memory device, however, flash memory will quickly become unusable with frequent memory writes.

We note that while these challenges for flash also exist in the storage domain, they are handled by the SSD-internal firmware. For CXL-flash, however, they should be addressed by hardware due to the much finer timescale, making it difficult to implement flexible and optimal algorithms. Thus, we expect these challenges to exacerbate when moving flash memory from the storage domain to the memory domain.

3 Tool and Methodology

To understand the behavior of physical memory accesses from the CPU to the CXL device, we build a physical memory tracing tool using page fault events (§ 3.1). We then demonstrate the necessity of this tool by comparing it with a set of virtual memory traces (§ 3.2). The tools and artifacts generated in this work are publicly available.

3.1 Tracing memory accesses

Main memory and CXL-flash are accessed via physical memory addresses. Unfortunately, to the best of our knowledge, no publicly available tool traces the physical memory transactions between the last level cache (LLC) and the memory controller without hardware modifications. Tracing the load/store instructions in the CPU is not sufficient as (1) it collects the

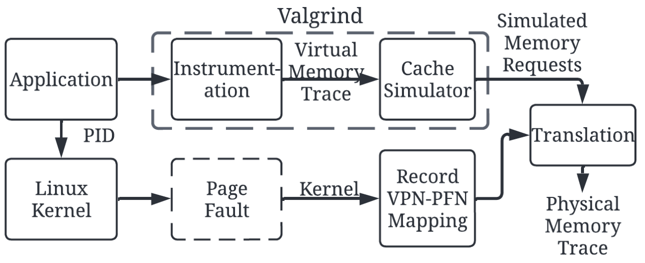


Figure 2: Workflow for collecting physical memory traces. We collect the virtual memory trace using Valgrind and simulate its behavior in the cache. Simultaneously, we capture page fault events to trace the updates to the page table, and use this to generate physical memory traces.

Table 2: Synthetic workload characteristics.

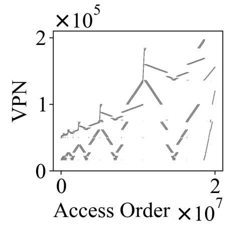
Workload	Inter-arrival time (ns)	Read-write ratio	Footprint (GiB)
Hash map	329	53:47	<1
Matrix multiply	38	55:45	<1
Min heap	72	50:50	1
Random	76	50:50	4
Stride	146	50:50	8

virtual address accesses, and (2) the eventual accesses to the CXL-flash are filtered by the cache hierarchy.

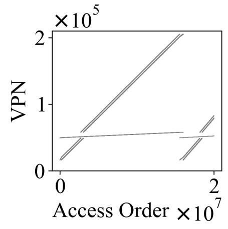
We trace physical memory accesses by combining memory tracing from Valgrind [19, 57] and information from page fault events. Figure 2 illustrates this workflow. As shown in the top path, we instrument the application with Valgrind for load/store instructions and use its cache simulator (Cachegrind) to filter accesses to memory. More specifically, we modify Cachegrind to collect memory accesses caused by LLC misses or evictions. However, these memory accesses from Cachegrind are still addressed virtually, and the virtual-to-physical (V2P) mapping information is needed to generate the physical memory trace. For this, as shown in the bottom path of Figure 2, we collect updates to the page table caused by page faults while the application is running. We modify kernel functions that install page table entries (`do_anonymous_page()` and `do_set_pte()`) and store the V2P translations for the target application's PID in the `/proc` file system. This captures the dynamic nature of page table updates during the execution of the application with minimal overhead. We combine the virtual accesses from Valgrind and the page table updates to generate the physical memory trace.

3.2 Virtual vs. physical memory accesses

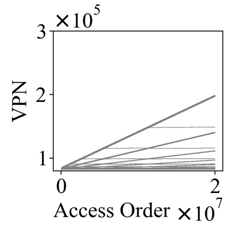
We demonstrate our physical memory tracing tool using five synthetic applications based on prior work on prefetching [25, 56]. The characteristics of collected traces are summarized in Table 2. We collect the first 20 million memory accesses:



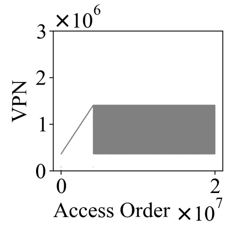
(a) Hash map (V)



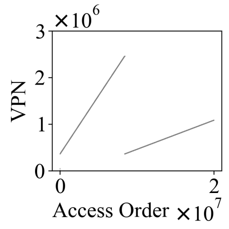
(b) Matrix mult. (V)



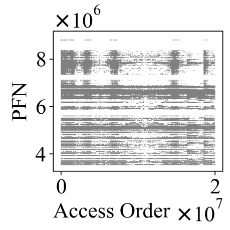
(c) Min heap (V)



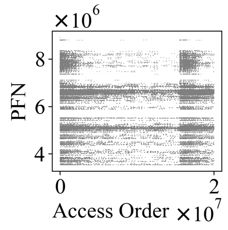
(d) Random (V)



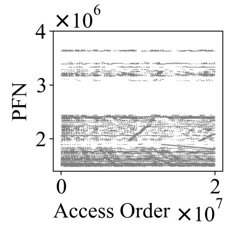
(e) Stride (V)



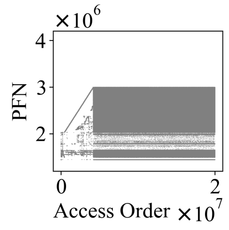
(f) Hash map (P)



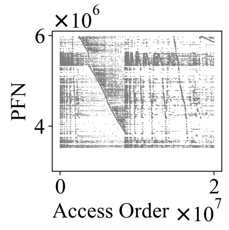
(g) Matrix mult. (P)



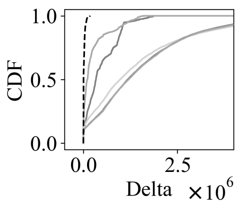
(h) Min heap (P)



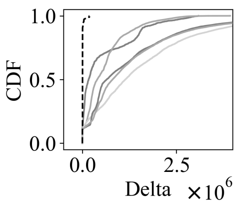
(i) Random (P)



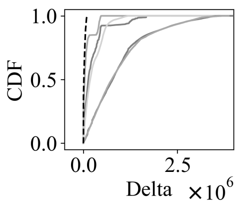
(j) Stride (P)



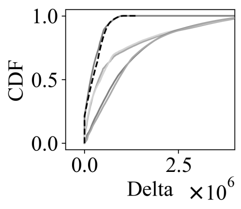
(k) Hash map (Δ)



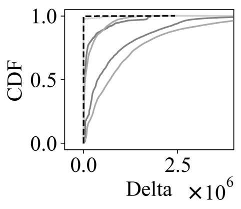
(l) Matrix mult. (Δ)



(m) Min heap (Δ)



(n) Random (Δ)



(o) Stride (Δ)

Figure 3: The scatter plots showing access patterns for five synthetic applications: hash map, matrix multiply, min heap, random, and stride. The top row (Figures 3a–3e) shows the virtual address accesses; the second row (Figures 3f–3j), the physical accesses. The last row (Figures 3k–3o) shows the CDF of the difference between consecutive accesses. We observe that the physical memory accesses appear different from the virtual ones due to address translation.

note these are not load/store instructions, but the memory transactions between the LLC and memory.

Figures 3a–3e (first row of Figure 3) plot the accessed virtual page number (VPN) for the five synthetic traces. We can observe that the virtual address access pattern matches our expectations for the application. However, as shown in Figures 3f–3j (second row of Figure 3), the corresponding physical frame number (PFN) does not resemble the VPN. We show the difference between consecutive accesses (Δ, delta) in Figures 3k–3o (last row of Figure 3). The black dashed line is the delta for the virtual address while the grey solid lines are the deltas for the physical addresses across five iterations, with the two of the iterations running while other applications are running to inflate the memory utilization. We make two observations. First, the virtual access patterns (black dashed lines) have much smaller delta values on average. However, the physical access patterns (grey solid lines) may have very large delta values due to virtual-to-physical address translation. Second, the grey solid lines rarely overlap with each other, highlighting that the physical memory pattern is dynamic and depends on various runtime factors that affect memory allocation. To this end, the observed mismatch between the physical and virtual addresses may be influenced by dynamic factors, such as memory utilization of the system.

To demonstrate that it is necessary to capture the physical memory trace, we measure the performance of a CXL-flash using the virtual and physical address traces as inputs. The CXL-flash is configured to have a flash memory backend of 8 channels and 8 ways per channel and a 512MiB DRAM cache, and implements a Next-N-line prefetcher [41] (more details in § 4). We measure the percentage of memory requests with less than a microsecond latency for the five synthetic applications and report the results in Table 3. Using virtual memory traces generates an *overly optimistic* result with far more requests completing under a microsecond compared to the result from running physical memory traces. The error between the running with virtual address and physical address is significantly high: all the matrix multiply experiments have errors over 25%. The random and stride access workload have low error rates, making it either too difficult or too easy to predict access patterns regardless of virtual or physical addressing.

One technique to mitigate the change of information during

Table 3: Percentage of sub-μs latencies for a CXL-flash using virtual and physical address traces for the five synthetic applications. We repeat the physical trace generation five times with iterations 4 and 5 having a higher system memory utilization (thus, a more fragmented memory layout). We compute the errors for the virtual trace performances in relation to those of the physical traces, and highlight errors over 10% in yellow (🟨), and over 25% in red (🟥).

Workload	% of sub-μs latency (virtual)	% of sub-μs latency (physical)					Error (%)				
		1	2	3	4	5	1	2	3	4	5
Hash map	96.9%	86.7%	88.3%	74.5%	63.8%	63.9%	10.2%	8.6%	22.4%	33.1%	33.0%
Matrix mult.	98.2%	72.7%	57.4%	59.2%	48.1%	47.9%	25.5%	40.8%	39.0%	50.1%	50.3%
Min heap	97.8%	92.1%	96.0%	75.6%	69.1%	69.4%	5.7%	1.8%	22.2%	28.7%	28.4%
Random	32.2%	26.4%	27.1%	28.0%	22.4%	21.8%	5.8%	5.1%	4.2%	9.8%	10.6%
Stride	64.7%	64.3%	59.4%	64.5%	51.9%	52.0%	0.4%	5.3%	0.2%	12.6%	12.7%

address translation is to utilize huge pages, which can significantly reduce the number of address translations [54, 58] to preserve memory access patterns. However, such a method can only reduce its impact on the system partially, and the address translation is inevitable. With the rapid growth of memory requirements of applications (14.1 × per year from Figure 1), within a few years, huge pages would exhibit the same challenges that the smaller pages face. Therefore, we decide to keep the configuration general to explore the design options for a CXL-flash.

4 Design Space for CXL-flash

We explore the design space for building a CXL-flash, specifically on the hardware modules inside it; we later evaluate algorithms and policies in § 5. To model the hardware, we build a CXL-flash simulator based on MQSim [68] and its extension MQSim-E [49], and use the physical memory traces of the five synthetic applications (Table 2) to evaluate the effects of design options. The overall architecture of our CXL-flash is depicted in Figure 4 with starting configuration in Table 4.

We answer the following research questions in this section.

- How effective is caching in improving performance? (§ 4.1)
- How can we effectively reduce flash memory traffic? (§ 4.2)
- How effective is prefetching in hiding the long flash memory latency? (§ 4.3)
- What are the appropriate flash memory technology and parallelism for CXL-flash? (§ 4.4)

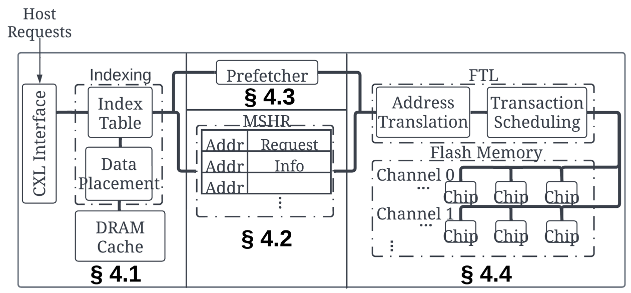


Figure 4: Architecture of the CXL-flash

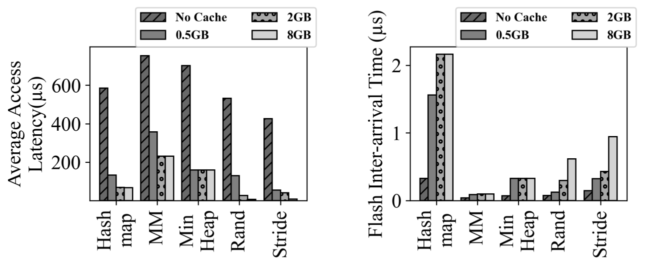
Table 4: Initial configuration for the CXL-flash in § 4.

Parameters	Value
DRAM latency	46ns
Cache replacement	FIFO
Flash parallelism	32 × 32
Flash technology	SLC (Table 1)

4.1 Caching for performance

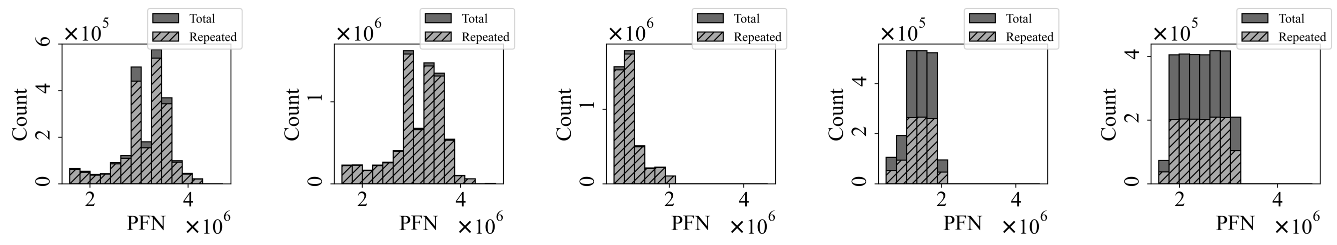
We first explore the effect of adding a DRAM cache in front of flash memory. The cache mainly serves two purposes. First, it improves the performance of the CXL-flash by serving frequently accessed data from the faster DRAM. Second, it reduces the overall traffic to flash memory on a cache hit.

Figure 5 quantitatively shows the benefit of using a cache. We vary the cache size from 0 to 8GiB and measure the average latency for the physical memory accesses (Figure 5a) and the inter-arrival time of flash memory requests to the backend. (Figure 5b). When there is no cache, the average latency is much higher than the flash memory read and program latencies because of queuing delays. This is even though the flash memory backend is configured to have an ample amount of



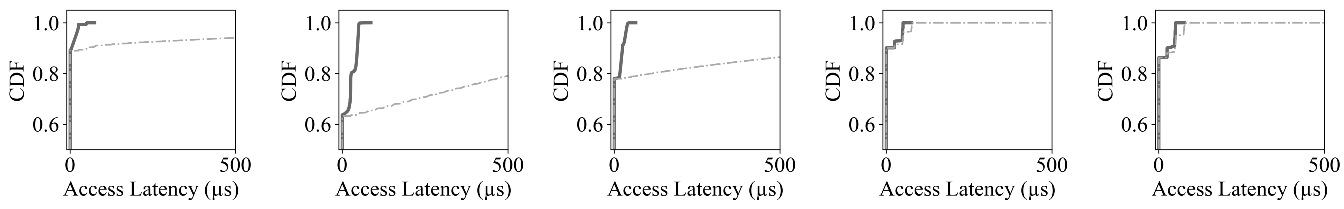
(a) Average access latency (b) Flash memory traffic

Figure 5: Average access latency (Figure 5a) for both 64B read and write requests and flash memory traffic (Figure 5b) as the DRAM cache size varies. In general, the cache improves performance and reduces the amount of traffic to flash. However, even with a sufficiently large cache, the average latencies are still much higher than that of DRAM due to the high intensity of memory accesses.



(a) Hash map (b) Matrix mult. (c) Min heap (d) Random (e) Stride

Figure 6: Flash memory read count for physical memory frames. The solid bar represents the total number of reads, while the shaded bar, the number of repeated reads. A repeated read is a read request to an outstanding read request.



(a) Hash map (b) Matrix mult. (c) Min heap (d) Random (e) Stride

Figure 7: The latency distribution with (solid lines) and without (dashed lines) MSHR.

parallelism at 32 channels and 32 ways per channel, it is insufficient to process the memory requests with short inter-arrival times. Adding a cache significantly reduces the traffic to the flash memory backend and improves the overall performance. However, we observe in Figure 5a that the average latencies for matrix multiply and min heap are still much higher than the DRAM latency, even though the memory footprints for these workloads are smaller than the cache. This is due to the short inter-arrival time of requests that overwhelm the flash memory backend for fetching data (Figure 5b). This experiment shows that caching alone is insufficient in reducing the latency of a CXL-flash, and we need additional auxiliary structures to reduce the traffic to flash memory.

4.2 Reducing flash memory traffic

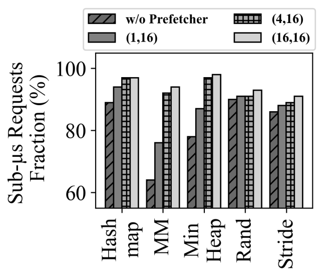
Memory accesses are at 64B granularity while the flash memory backend is addressed at 4KiB units. Thus, upon a cache miss, 4KiB of data will be fetched from flash, and subsequent 64B cache misses that belong to the same 4KiB will generate additional flash memory read requests even when the flash memory read is in progress. This scenario is very likely for memory accesses with high spatial locality, and exacerbated by the much longer flash memory latency. We call these *repeated reads*, and Figure 6 illustrates the severity of repeated reads in the hash map, matrix multiply, and heap workloads: over 90% of flash memory reads are repeats!

Inspired by CPU caches, we add a set of MSHRs (miss status holding registers) [29, 47] to the CXL-flash, as shown in Figure 4. MSHR tracks the current outstanding flash memory requests and services multiple 64B memory accesses from a single flash memory read. We note that MSHRs are uncommon in SSDs: in the storage domain, the software stack merges block I/Os with overlapping addresses so there is no need for the underlying device to implement MSHRs. However, for CXL-flash, there is no software layer to perform this duty as it receives memory transactions directly from the LLC. We observe in Figure 7 that MSHRs significantly reduce long tail latencies, particularly for the three workloads with a significant number of repeated reads. We also observe small improvements to the other two workloads, random and stride, by adding MSHR. However, MSHRs only reduce flash memory traffic, and it does not actively improve the cache hit rate by bringing data into the cache before they are needed.

4.3 Prefetching data from flash

Prefetching is an effective technique for hiding the long latency of a slower technology. Typically prefetchers fetch additional data upon a demand miss or a prefetch hit. To understand the effectiveness of this technique, we implement a simple Next-N-line prefetcher [41] in our CXL-flash, as shown in Figure 4. This prefetcher has two configurable parameters: degree and offset. The degree controls the amount of additional data to fetch while the offset determines the prefetch address from the triggering one. In other words, the degree parameter represents the aggressiveness of the prefetcher, and the offset controls how far ahead the prefetcher is fetching.

Figure 8 shows the effect of varying the degree and offset for the prefetcher. For the (X, Y) notation in a unit of the


(a) Sensitivity to degree

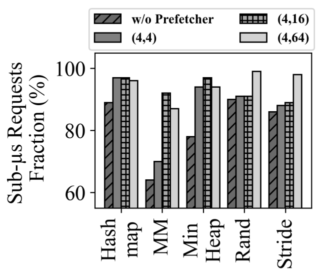
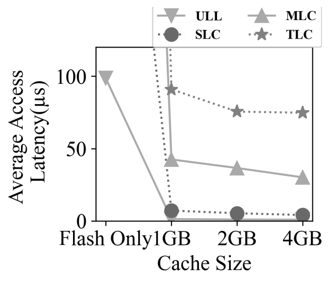

(b) Sensitivity to offset

Figure 8: The performance of the CXL-flash with different prefetcher configurations. (X, Y) represents the degree and offset for the Next-N-line prefetcher.


(a) Average access latency

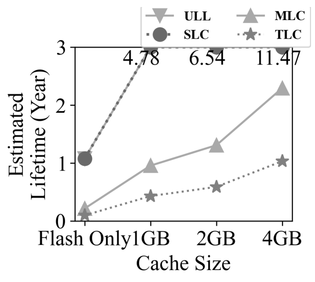
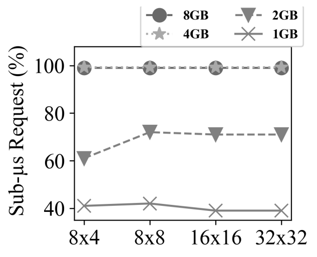

(b) Estimated lifetime

Figure 9: Sensitivity test to flash technology and cache size on the performance and lifetime of CXL-flash with stride.


(a) Random

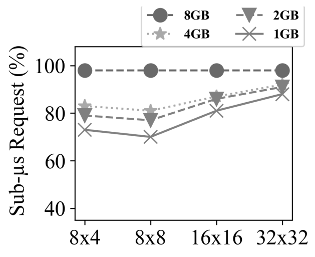

(b) Stride

Figure 10: Percentage of sub-μs requests when varying flash parallelism and cache size. The x-axis represents the flash memory parallelism (channels $\times$ ways). The lines represent values for different cache sizes.

number of 4KiB pages, X denotes the degree, and Y is the offset. As shown in Figure 8a, increasing the degree, or the aggressiveness of the prefetcher generally improves the performance. Even a small degree of 1 increases the portion of sub-microsecond requests from 64% to 76% for the matrix multiply workload, highlighting the necessity of prefetching for CXL-flash. However, the improvement plateaus and further increasing the degree may only pollute the cache. On the other hand, increasing the offset shows two different behaviors depending on the workload. For the hash map, matrix multiply, and min heap workloads, the performance first improves when increasing the offset from 4 to 16. However, an offset of 64 deteriorates the performance as it fetches data too far out. The random workload is insensitive to the offset unless it is large enough, while the stride workload shows gradual improvement as the offset increases.

4.4 Exploring flash technology and parallelism

In previous subsections, we examine the performance of the CXL-flash using SLC flash technology and ample flash parallelism of 32 channels and 32 ways per channel (32×32). In this section, we experiment with how sensitive technology (ULL, SLC, MLC, and TLC) and parallelism (8×4, 8×8, 16×16, and 32×32) are to the overall CXL-flash performance.

We first examine the effect of flash technology and cache size for the stride workload as shown in Figure 9. We use this workload as it performs well in the default configuration, thus we expect it to represent the workload with the lowest room for improvement. Figure 9a illustrates the average latency for the different memory technologies. We observe that even though ULL and SLC flash have a noticeable difference in latency (3μs vs. 25μs), the performance difference between the two is negligible with the existence of a cache. Only when there is no DRAM cache, ULL flash is outstandingly better. We also observe that using MLC and TLC technology degrades the performance significantly. Figure 9b shows the estimated lifetime of the CXL-flash based on the amount of flash write traffic. This estimation takes into consideration the endurance limit, the capacity, and the amount of data written; it optimistically assumes that the write amplification at the flash memory backend is negligible. We observe that with ULL and SLC technologies and some cache, the CXL-flash can achieve a lifetime of more than 4 years. Increasing the size of the cache further improves the lifetime due to reduced flash write traffic. For MLC- and TLC-based CXL-flash, it would only be viable with a sufficiently large cache: with only a 1GiB of cache, it would not last more than a year.

Next, we investigate the effect of varying flash parallelism and cache size on the overall performance using the random (Figure 10a) and stride (Figure 10b) workloads. We use these two as they have the largest memory footprint (8GiB and 4GiB, respectively). Flash technology here is SLC. We observe that with a sufficiently large cache, reducing parallelism to (8×4) does not adversely affect the performance. However, with smaller caches, the flash parallelism matters. Interestingly, the two workloads exhibit slightly different behaviors. The random workload shows high sensitivity to the cache size. On the other hand, the stride workload is less sensitive to the cache size but more to the parallelism. This is due to the prefetcher's effectiveness with stride workloads.

Table 5: Workload characteristics of real-world applications. The spatial and temporal locality values range between 0 and 1, and are computed using the stack and block affinity metrics [32]: a higher value indicates higher locality.

Workload	Category	Description	Inter-arrival time (ns)	# of accesses (M)	Read-write ratio	Footprint (GiB)	Spatial locality	Temporal locality
BERT [18]	NLP	Infers using a transformer model	297	41	73:27	0.5	0.64	0.66
Page rank [6]	Graph	Computes the page rank score	51	435	68:32	3.7	0.40	0.42
Radiosity [17]	HPC	Computes the distribution of light	1863	61	84:16	1.6	0.93	0.87
XZ [21]	SPEC	Compresses data in memory	237	71	55:45	0.9	0.31	0.38
YCSB F [22]	KVS	Read-modify-writes on Redis [14]	1137	110	65:35	1.8	0.56	0.55

4.5 Summary of findings

We briefly summarize our findings.

- Caching alone is not sufficient to hide the much longer flash memory latencies (§ 4.1), and we need auxiliary structures to reduce the flash memory traffic (§ 4.2).
- Prefetching data improves the CXL-flash's performance, but the best configuration (or even the algorithm) is workload-dependent (§ 4.3).
- The performance difference between using ULL and SLC is only marginal, and it is challenging to utilize MLC and TLC flash in terms of both performance and lifetime (§ 4.4).

5 Evaluation of Policies

Building on top of our design space exploration for the CXL-flash architecture from § 4, we evaluate advanced caching and prefetching policies in this section. We use five different real-world applications that are memory-intensive from a wide variety of domains: natural language processing [18, 70], graph processing [6, 27], high-performance computing [17, 62], SPEC CPU [16, 21], and key-value store [22, 31]. We collect the physical memory traces using our tool (§ 3) and summarize the workload characteristics in Table 5.

However, the memory footprints of the real applications are smaller than we had anticipated, even though they are collected on a machine with 64GiB of memory. Thus, we intentionally configure the cache to be small (64MiB) so that experimental results would scale up for larger workloads. We also scale down the flash parallelism to a more realistic setting and use ULL flash. The default parameters for the CXL-flash in this section are summarized in Table 6.

Table 6: Default parameters for the CXL-flash in § 5.

Parameters	Value
DRAM size	64MiB
DRAM latency	46ns
Flash parallelism	8×8
Flash technology	ULL (Table 1)

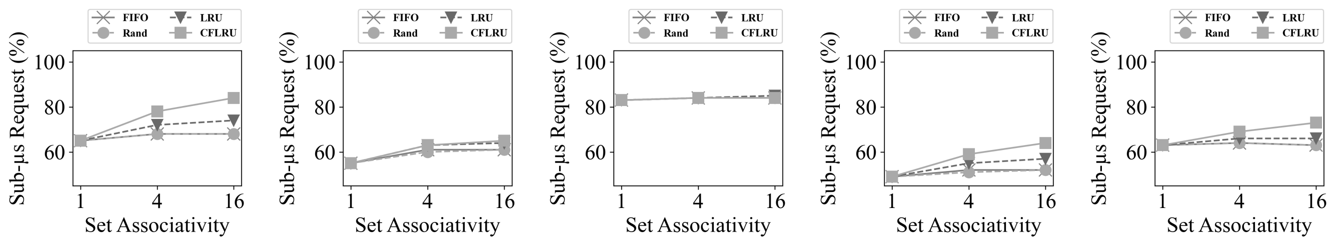
5.1 Cache replacement policy

Unlike the previous examination of cache size on performance (§ 4.1), here we fix the cache size and evaluate the effects of different cache replacement policies across different set associativities. In particular, we implement the following four.

FIFO evicts the oldest data.
Random selects data arbitrarily to evict.
LRU kicks out the least recently used data.
CFLRU [60] prefers to evict clean data over modified ones.

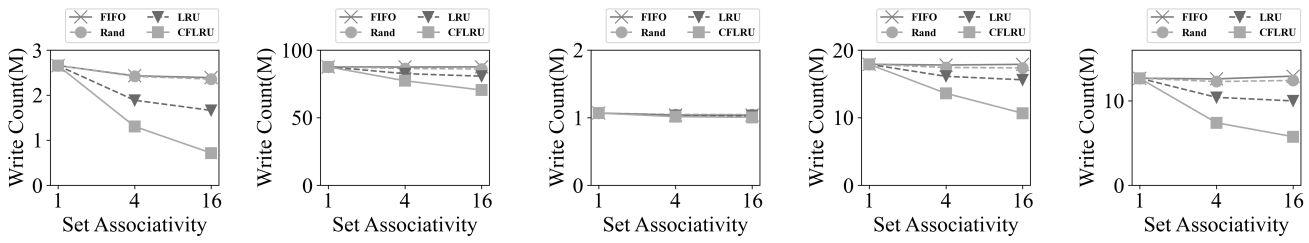
We select Random as our baseline, and FIFO and LRU are two standard CPU cache policies implementable in hardware. To further reduce traffic and extend the device's lifetime, we implement CFLRU to explore the benefits of prioritizing evicting clean cache lines to reduce flash write activities.

Figure 11 measures the percentage of memory requests to the CXL-flash with less than a microsecond latency, and Figure 12 shows the number of flash memory writes. We make five observations from these figures. First, increasing associativity improves performance as it increases the cache hit rate. For a caching system whose miss penalty is high, increasing the hit rate has a greater impact than reducing hit time. Second, CFLRU outperforms the other replacement policies, particularly in BERT, XZ, and YCSB (Figures 11a, 11d, and 11e). This is supported by the significant reduction in write traffic as shown in Figures 12a, 12d, and 12e. Third, workloads with high localities such as Radiosity are insensitive to cache replacement policies (Figures 11c and 12c): at least 83% of the request have sub-microsecond latency regardless of the policy. Four, read-dominant workloads generally perform better than write-heavy ones as the flash memory program latency is disproportionately larger than that of read. BERT and Radiosity only generate as low as 0.7M and 1.0M flash writes, respectively (Figures 12a and 12c), and in turn, their portion of sub-microsecond latencies are as high as 84% and 85%, respectively (Figures 11a and 11c). Lastly, workloads with low localities not only perform poorly but also are less sensitive to the cache policies. In particular, as shown in Figure 11b, only at most 65% of the requests achieve a sub-microsecond latency for the page rank workload due to its low localities and large footprint. The XZ trace in Figure 11d



(a) BERT (b) Page Rank (c) Radiosity (d) XZ (e) YCSB

Figure 11: Percentage of CXL-flash latencies smaller than a microsecond with respect to cache replacement policies and set associativity. In general, increasing associativity reduces the latency and CFLRU performs better than the others.



(a) BERT (b) Page Rank (c) Radiosity (d) XZ (e) YCSB

Figure 12: Number of flash memory write requests with respect to cache replacement policies and set associativity. CFLRU noticeably reduces the number of writes as the associativity increases.

also exhibits low localities but is more sensitive to CFLRU than page rank as the workload has a higher write ratio.

In the storage domain, reducing the amount of data written to the SSD is achieved by various software-level techniques, including the OS-level page cache. Cache management for CXL-flash, however, behaves similarly to CPU caches, and there may be limitations to how close it approaches optimality.

5.2 Prefetching policy

Previously in § 4.3, we measured the effectiveness of a simple Next-N-line prefetcher with a large 8GiB cache. In this section, we scale down the cache to 64MiB, set its associativity to 16, and manage it using the CFLRU algorithm, and measure the performance of the following five prefetcher settings.

NP (No prefetch) does not prefetch any data.

NL (Next-N-line) [41] brings in the next *N* data upon a demand miss or prefetch hit.

FD (Feedback-directed) [65] dynamically adjusts the aggressiveness of the prefetcher by tracking prefetcher accuracy, timeliness, and pollution.

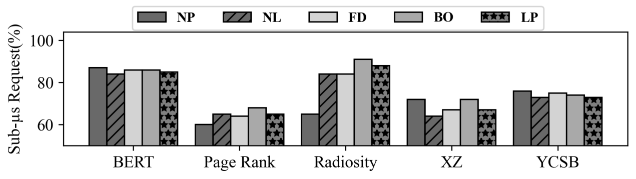
BO (Best-offset) [55] learns the deltas between consecutive accesses by tracking the history of recent requests. It also has a notion of confidence to disable prefetching.

LP (Leap) [53] implements a majority-based prefetching with dynamic window size adjustment. It also gradually adjusts aggressiveness based on the prefetcher accuracy.

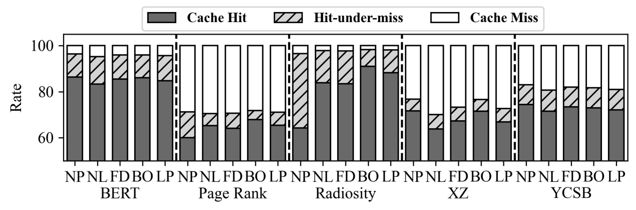
We select these algorithms as they are proven to be effective, implementable in hardware, and fit into the design space of a CXL-flash. In particular, NL, FD, and BO are prefetchers for CPU cache, where BO is an enhancement of NL, and FD utilizes performance metrics we will later discuss. LP is primarily for prefetching data from remote memory, where the latency difference between a cache hit and a cache miss can be similar to that in our design space.

Observation #1: ***Although 68–91% of requests have a latency of under a microsecond, using a prefetcher can be detrimental to the performance of real-world applications.*** As shown in Figure 13a, the state-of-the-art prefetchers degrade the performance for three workloads, BERT, XZ, and YCSB workloads, and are only helpful for the other two workloads. Radiosity, in particular, shows a 36% increase in sub-microsecond latencies when using the best-offset prefetcher. To understand why, we examine the cache hit, hit-under-miss (HUM), and miss rate in Figure 13b. A cache hit-under-miss refers to a hit in the MSHR where while the data is not in the cache yet, it is being fetched due to a previous miss. We observe that BO on Radiosity converts 25% of hit-under-miss into hits, indicating high effectiveness of prefetching on workloads with a high spatial locality factor (*cf.* Table 5). Our observation indicates that the performance of prefetchers depends on the characteristics of workloads, and they can have detrimental effects on applications.

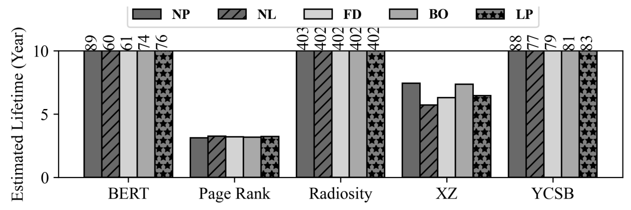
Observation #2: ***Even under high-intensity workloads, a CXL-flash has a lifetime of at least 3.1 years.*** We estimate the lifetime of the CXL-flash under real workloads based on the amount of data written to flash, endurance limit, and 1TiB capacity, as shown in Figure 13c. We observe, in the worst case, the device would last 3.1 years under Page Rank, but



(a) Percentage of sub-μs requests



(b) Cache hit, cache hit-under-miss (HUM), and cache miss



(c) Device lifetime

Figure 13: CXL-flash's performance and lifetime with different prefetchers. Figure 13a shows the portion of requests with a latency of less than a microsecond. Figure 13b shows the hit, hit-under-miss, and miss rate of the 64MiB cache inside the CXL-flash. Figure 13c plots the estimated lifetime.

under workloads such as Radiosity, it would be as much as 403 years. Three factors contribute to the lifetime: workload intensity, read-write ratio, and locality; Page rank has the highest workload intensity, a high ratio of writes, and a low locality. Even under this adverse condition, the CXL-flash provides a reasonable lifetime; hence, the durability of the CXL-flash can sustain the intensity of memory requests.

Observation #3: ***A CXL-flash has a better performance per cost than a DRAM-only device.*** While a CXL-flash falls slightly short of achieving a DRAM-like performance for sub-µs requests, our analysis reveals its potential to provide benefits for memory-intensive applications. As a CXL-flash can serve 68–91% of the memory requests under a microsecond, and the recent price point of DRAM is 17 - 100× higher than that of NAND flash [39, 64, 77], we expect an 11 - 91× performance-per-cost benefit from a CXL-flash over a DRAM-only device, as depicted in Figure 14. Although some cases may still prefer a DRAM-only device when achieving the best performance is essential, a CXL-flash can be a cost-effective memory expansion option depending on the workload.

Interestingly, we observe that while prefetchers are useful for Page Rank, their performance is overall the worst, with only at most 68% of requests completing under a microsecond. To further understand the performance of prefetchers, we measure the following four metrics.

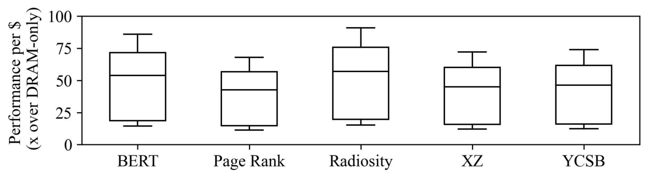


Figure 14: Performance-per-cost benefits of a CXL-flash with BO prefetcher over a DRAM-only device. The estimation is derived from the performance results in Figure 13a, the recent price point of DRAM at 5 $/GB [77], and the price range of NAND flash varying from 0.05 to 0.30 $/GB [39, 64].

Accuracy measures how much of the data brought in by the prefetcher is actually used. Higher is better.

Coverage is the portion of prefetched data cache hits among the memory requests. A high coverage means that cache hits are thanks to the prefetcher, while a low coverage indicates that the prefetcher is not contributing.

Lateness is the portion of late prefetches among all the prefetches. A prefetched data is late if it is accessed while it is being prefetched. Lower is better.

Pollution measures how many cache misses are caused by the prefetcher among cache misses. Lower is better.

Observation #4: ***In cases where the prefetcher improves the performance, it is due to achieving high accuracy.*** We plot the four metrics for the evaluated prefetchers in Figure 15. Lateness and pollution are negative metrics (the lower the better), so we invert their bars so that higher is better for all metrics. We observe that the defining characteristic for the workloads where the prefetcher is helpful (Page Rank and Radiosity) is that the accuracy is high. For example, the Leap (LP) prefetcher attains 85% accuracy under Radiosity while only reaching 27% under BERT. Additionally, the Best-offset (BO) prefetcher achieves 48% accuracy under XZ; however, its limited coverage of 4% suggests that despite achieving relatively higher accuracy, the prefetcher is not actively fetching data to contribute to performance improvement.

We further analyze Page Rank to understand why prefetchers are able to reach relatively high accuracy even though the workload has the lowest locality (computed using the stack and block affinity metrics [32]). As Figure 16 shows, the Page Rank exhibits distinct phases in their workload. During the first phase, Page Rank loads graph information and exhibits high locality (Figure 16a). The best-offset prefetcher is also able to attain high coverage and accuracy (Figure 16b). However, in the second phase, Page Rank builds the graph, and the access pattern here has a very low locality. Consequently, the best-offset prefetcher becomes more inactive (low coverage) as its accuracy drops. During the last phase, Page Rank computes the score for each vertex. While its access locality is not high, the prefetcher performs well and most of the accesses hit in the cache. Note that while the pollution is bad, the cache miss rate is very low so its impact on performance

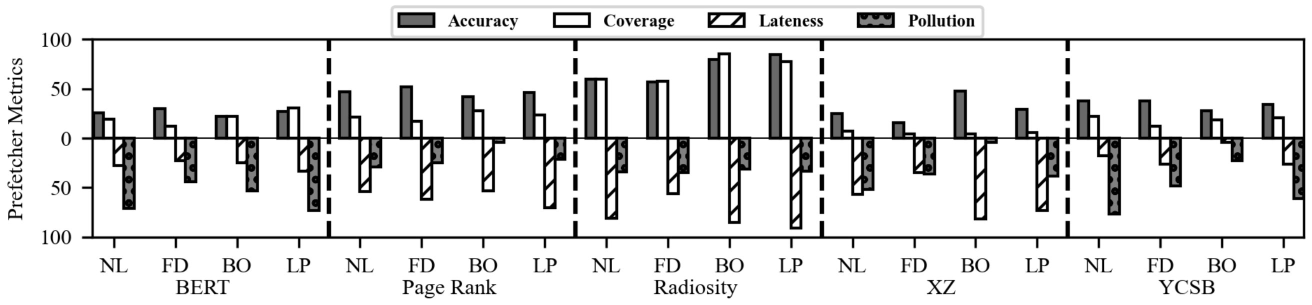
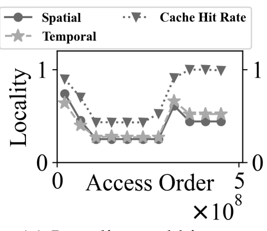
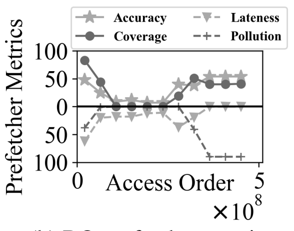


Figure 15: Accuracy, coverage, lateness, and pollution metrics for the prefetchers.



(a) Locality and hit rate



(b) BO prefetcher metrics

Figure 16: Page Rank behavior over time.

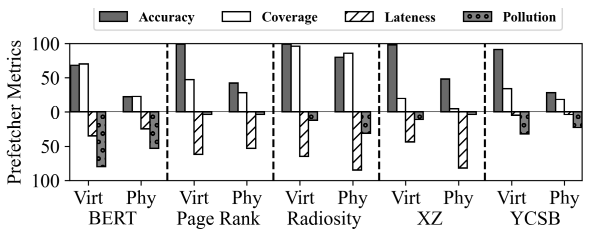


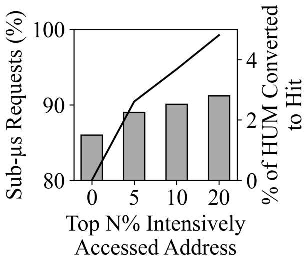
Figure 17: BO prefetcher metrics for virtual vs. physical.

is negligible. This analysis indicates that while the prefetcher is beneficial for the first and last phases, the low locality in the second phase limits the performance.

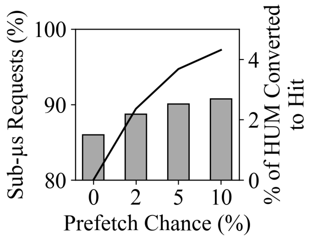
Observation #5: ***Cache pollution is the main reason behind performance degradation when the accuracy is low.*** As shown in Figure 15, BERT and YCSB have low accuracies while their pollutions are high, leading to a result where enabling prefetchers degrades performance (Figure 13a). For XZ, even though the accuracy of the best-offset (BO) prefetcher is low, it is no worse than no prefetcher as it causes little pollution. We attribute this to BO's ability to disable prefetching based on its accuracy. For Page Rank and Radiosity, prefetchers exhibit low pollution although their lateness is high. Cache pollution degrades the performance of a CXL-flash, and prefetchers should be aware of the impact to avoid having detrimental effects on the device.

Observation #6: ***The virtual-to-physical address translation makes it difficult for the CXL-flash to prefetch data.*** To understand the effect of V2P address translation, we simulate the CXL-flash with the best-offset prefetcher using the virtual memory traces of the five application workloads, and Figure 17 compares the four prefetcher metrics between virtual and physical traces. We make two observations. First, aside from Radiosity, we observe a significant drop in accuracy from virtual to physical traces. The BO prefetcher under Page Rank is 99% accurate for the virtual memory trace, but with the physical trace, accuracy drops to 42%. Second, coverage also drops, indicating that the prefetcher becomes less active under physical memory accesses: for example, it drops from 76% to 26% under BERT. The drop in both accuracy and coverage for physical traces shows that the CXL-flash would perform better if it were addressed virtually.

Observation #7: ***If the kernel were to provide memory access pattern hints to the device, the CXL-flash performance improves by converting hit-under-misses into cache hits.*** We consider a hypothetically clairvoyant kernel that knows the physical memory access pattern. This is not too far-fetched as data-intensive applications often iterate multiple times and their behaviors can be profiled. More specifically, we assume that the kernel has information on the top intensively accessed physical frames, and can pass hints to the device prior to their actual accesses. To limit the overhead of kernel involvement, we model a probabilistic generation of access hints. Figure 18a shows the performance improvement for BERT when hints are generated at 10% for the top N% of intensively accessed addresses. We observe that with access hints to more addresses, the percentage of sub-microsecond latencies increases from 86% to 91% by converting hit-under-



(a) Sensitivity to # of addresses.



(b) Sensitivity to hint chance.

Figure 18: Improvement in performance with memory access pattern hints for BERT. Figure 18a is the sensitivity to the number of addresses for which hints are provided. Figure 18b shows the performance improvement as more hints are added. In both figures, the line represents the number of hit-under-misses without hints converted to cache hits with hints.

misses (HUM) into cache hits. Figure 18b considers a variable hint generation chance, from 0% to 10% for the top 10% of intensively accessed addresses. Similarly, we see an overall improvement in performance, though it plateaus at 91%. Our experiments show that host-generated access pattern hints leveraging the host's knowledge of the workload behaviors can potentially improve the CXL-flash performance.

6 Related Work

The evaluated cache policies and prefetching algorithms are well-studied in prior proposals. However, most of them are for managing and optimizing the CPU cache [41, 55, 65], where the latency difference between a cache hit and a cache miss is much smaller than that in a CXL-flash. CFLRU [60] and Leap [53] share a similar design space to our device; however, the memory access intensity they face is not as extreme as what a CXL-flash needs to handle. Therefore, it is crucial to evaluate the effectiveness of these policies and algorithms under the design space of a CXL-flash.

Techniques to mitigate the performance degradation due to address translations and limitations of flash has been explored in prior works. Utilizing huge pages can reduce the number of address translations [54, 58]. FlashMap [40] and FlatFlash [23] combine address translation of the SSD with the page table to reduce overheads. eNVY [73] employs write buffering, page remapping, and a cleaning policy to enable direct memory addressability and sustain performance. Future research in CXL host systems should further explore the potential benefits of host-generated hints and insights from these prior works to reduce the overheads.

Memory disaggregation organizes memory resources across servers as a network-attached memory pool, enabling meeting the high memory requirements for data-intensive applications [37, 38, 52, 69]. While our work does not directly investigate memory disaggregated systems, using CXL-flash as disaggregated memory helps overcome the memory wall.

Utilization of non-DRAM to expand memory has been explored in prior works [28,40,63,74]. HAMS [75] aggregates persistent memory and ULL flash as a memory expansion by managing data paths among hosts and memory hardware in an OS-transparent manner. Suzuki et al. [67] present a lightweight DMA-based interface that bypasses the NVMe protocols to enable flash read access with DRAM-like performance. SSDAlloc [26] is a memory manager and a runtime library that allows applications to use flash as a memory device through the OS paging mechanism, which can cause overheads when accessing data in SSDs. FlatFlash [23] exposes a flat memory space using DRAM and flash memory by integrating the OS paging mechanism and the SSD's internal mapping table. While these prior works primarily focus on OS-level management and host-device interaction, our work builds on top of them by investigating the design decisions internal to a memory expansion device.

Memory expansion with CXL Type 3 devices is an active research area [36, 42, 50, 71]. Pond [50] utilizes CXL to improve DRAM memory pooling in cloud environments and proposes machine-learning models to manage local and pooled memory. While this work investigates how to use a CXL Type 3 device in a cloud setting, our work studies how to implement one using flash memory. DirectCXL [36] successfully connects host processors with external DRAM via CXL in real hardware and develops a software runtime to directly access the resources. Lastly, CXL-SSD [42] advocates combining CXL and SSD to expand host memory. While we share the same goal with this work, it mainly discusses the CXL interconnect and scalability potentials of CXL-SSDs.

ML-specific designs build systems that address the memory wall challenge [45, 48, 51]. MC-DLA [48] proposes an architecture that aggregates memory modules to expand the memory capacity for training ML models on accelerators. Behemoth [45] observes that many NLP models require large amounts of memory but not a lot of bandwidth, and proposes a flash-centric training framework that manages data movement between memory and SSDs to overcome the memory wall.

7 Conclusion

We explore the design space of a CXL-flash device and evaluate existing optimization techniques. Using physical memory traces, we find that 68–91% of memory access achieves a sub-microsecond latency for a CXL-flash device that can last at least 3.1 years. We discover that the address translation for virtual memory makes it particularly difficult for the CXL-flash's prefetcher to be effective and suggest passing kernel-level access pattern hints to further improve the performance.

While we attempt to generalize the results by testing the device with a variety of workloads and design parameters, it is important to acknowledge a few limitations. The current design of a CXL-flash as explored in this paper does not consider the flash's internal tasks such as garbage collection and wear leveling. In addition, the host system considered may not fully reflect the new system characteristics introduced by CXL. Therefore, we believe more work needs to be done in the CXL-flash research space, and our work can be a platform on which future research can build upon.

Acknowledgements

We thank our Shepherd, Animesh Trivedi, and the anonymous reviewers for their insightful comments. We thank Jongmoo Choi for reviewing the early draft of this paper. This work is funded in part by FADU Inc., the National Research Foundation of Korea (NRF-2018R1A5A1060031), the MOTIE (Ministry of Trade, Industry & Energy) (1415181081), and the KSRC (Korea Semiconductor Research Consortium) (20019402).

References

[1] Apache Spark. https://spark.apache.org/.

[2] CCIX consortium. https://www.ccixconsortium.com/.

[3] CXL consortium. https://www.computeexpresslink.org/.

[4] DDR memory speeds and compatibility. https://www.crucial.com/support/memory-speeds-compatability.

[5] The difference between RAM speed and CAS latency. https://www.crucial.com/articles/about-memory/difference-between-speed-and-latency.

[6] GAP benchmark suite. https://github.com/sbeamer/gapbs.

[7] Gen-Z consortium. https://genzconsortium.org/specifications/.

[8] Memcached. http://memcached.org/.

[9] Memory-Semantic SSD. https://samsungmsl.com/ms-ssd/.

[10] MongoDB. https://www.mongodb.com/home.

[11] NVIDIA H100Tensor core GPU. https://www.nvidia.com/en-in/data-center/h100/.

[12] OpenCapi consortium. https://opencapi.org/.

[13] PCIe 5.0 multi-port switch. https://www.rambus.com/interface-ip/controllers/pci-express-controllers/pcie5-multi-port-switch/.

[14] Redis. https://redis.io/.

[15] Samsung PM9A3. https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/.

[16] SPEC CPU 2017. https://www.spec.org/cpu2017/.

[17] The Splash-3 benchmark suite. https://github.com/SakalisC/Splash-3.

[18] TensorFlow code and pre-trained models for BERT. https://github.com/google-research/bert.

[19] Valgrind. https://valgrind.org/.

[20] VoltDB. https://github.com/VoltDB/voltdb.

[21] XZ utils. https://www.tukaani.org/xz/.

[22] Yahoo! cloud serving benchmark. https://github.com/brianfrankcooper/YCSB.

[23] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen mei Hwu. FlatFlash: Exploiting the byte-accessibility of SSDs within a unified memory-storage hierarchy. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, page 971–985. Association for Computing Machinery, 2019. https://doi.org/10.1145/3297858.3304061.

[24] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 1.00 edition, August 2018.

[25] Grant Ayers, Heiner Litz, Christos Kozyrakis, and Parthasarathy Ranganathan. Classifying memory access patterns for prefetching. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, page 513–526. Association for Computing Machinery, 2020. https://dl.acm.org/doi/10.1145/3373376.3378498.

[26] Anirudh Badam and Vivek S. Pai. SSDAlloc: Hybrid SSD/RAM memory management made easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, page 211–224. USENIX Association, 2011. https://www.usenix.org/legacy/event/nsdi11/tech/full_papers/Badam.pdf.

[27] Scott Beamer, Krste Asanović, and David Patterson. The GAP benchmark suite. *arXiv:1508.03619 [cs.DC]*, 2015. https://arxiv.org/abs/1508.03619.

[28] Adrian M. Caulfield, Laura M. Grupp, and Steven Swanson. Gordon: Using flash memory to build fast, power-efficient clusters for data-intensive applications. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, page 217–228. Association for Computing Machinery, 2009. https://dl.acm.org/doi/10.1145/2528521.1508270.

[29] Xi E. Chen and Tor M. Aamodt. Hybrid analytical modeling of pending cache hits, data prefetching, and MSHRs. In *2008 41st IEEE/ACM International Symposium on Microarchitecture*, pages 59–70, 2008.

[30] Eli Collins and Zoubin Ghahramani. LaMDA: our breakthrough conversation technology, 2021. https://blog.google/technology/ai/lamda/.

[31] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154. Association for Computing Machinery, 2010. https://dl.acm.org/doi/10.1145/1807128.1807152.

[32] Cory Fox, Dragan Lojpur, and An-I Andy Wang. Quantifying temporal and spatial localities in storage workloads and transformations by data path components. In *2008 IEEE International Symposium on Modeling, Analysis and Simulation of Computers and Telecommunication Systems*, pages 1–10, 2008. https://ieeexplore.ieee.org/document/4770561.

[33] Eran Gal and Sivan Toledo. Algorithms and data structures for flash memories. *ACM Comput. Surv.*, 37(2):138–163, 2005. https://dl.acm.org/doi/10.1145/1089733.1089735.

[34] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W Mahoney, and Kurt Keutzer. Ai and memory wall. *RiseLab Medium Post*, 2021. https://github.com/amirgholami/ai_and_memory_wall.

[35] Joseph E. Gonzalez, Reynold S. Xin, Ankur Dave, Daniel Crankshaw, Michael J. Franklin, and Ion Stoica. GraphX: Graph processing in a distributed dataflow framework. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 599–613. USENIX Association, 2014. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/gonzalez.

[36] Donghyun Gouk, Sangwon Lee, Miryeong Kwon, and Myoungsoo Jung. Direct access, high-performance memory disaggregation with DirectCXL. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 287–294. USENIX Association, 2022. https://www.usenix.org/conference/atc22/presentation/gouk.

[37] Juncheng Gu, Youngmoon Lee, Yiwen Zhang, Mosharaf Chowdhury, and Kang G. Shin. Efficient memory disaggregation with Infiniswap. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 649–667. USENIX Association, 2017. https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/gu.

[38] Zhiyuan Guo, Yizhou Shan, Xuhao Luo, Yutong Huang, and Yiying Zhang. Clio: A hardware-software codesigned disaggregated memory system. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 417–433. Association for Computing Machinery, 2022. https://dl.acm.org/doi/abs/10.1145/3503222.3507762.

[39] Gertjan Hemink and Akira Goda. 5 - nand flash technology status and perspectives. In Andrea Redaelli and Fabio Pellizzer, editors, *Semiconductor Memories and Systems*, Woodhead Publishing Series in Electronic and Optical Materials, pages 119–158. Woodhead Publishing, 2022. https://www.sciencedirect.com/science/article/pii/B9780128207581000030.

[40] Jian Huang, Anirudh Badam, Moinuddin K. Qureshi, and Karsten Schwan. Unified address translation for memory-mapped SSDs with FlashMap. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, ISCA '15, page 580–591. Association for Computing Machinery, 2015. https://www.usenix.org/conference/hotstorage18/presentation/koh.

[41] Norman Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373, 1990. https://ieeexplore.ieee.org/document/134547.

[42] Myoungsoo Jung. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *HotStorage '22*, page 45–51. Association for Computing Machinery, 2022. https://doi.org/10.1145/3538643.3539745.

[43] Paresh Kharya and Ali Alvi. Using DeepSpeed and Megatron to train Megatron-Turing NLG 530B, the world's largest and most powerful generative language model, 2021. https://developer.nvidia.com/blog/using-deepspeed-and-megatron-to-train-megatron-turing-nlg-530b-the-worlds-largest-and-most-powerful-generative-language-model/.

[44] Bryan S. Kim, Jongmoo Choi, and Sang Lyul Min. Design tradeoffs for SSD reliability. FAST'19, page 281–294. USENIX Association, 2019. https://www.usenix.org/conference/fast19/presentation/kim-bryan.

[45] Shine Kim, Yunho Jin, Gina Sohn, Jonghyun Bae, Tae Jun Ham, and Jae W. Lee. Behemoth: A flash-centric training accelerator for extreme-scale DNNs. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 371–385. USENIX Association, 2021. https://www.usenix.org/conference/fast21/presentation/kim.

[46] Sungjoon Koh, Changrim Lee, Miryeong Kwon, and Myoungsoo Jung. Exploring system challenges of ultra-low latency solid state drives. In *Proceedings of the 10th USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'18, page 15. USENIX Association, 2018. https://www.usenix.org/conference/hotstorage18/presentation/koh.

[47] David Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of the 8th Annual Symposium on Computer Architecture, Minneapolis, MN, USA, May 1981*, pages 81–88, 1981. http://dl.acm.org/citation.cfm?id=801868.

[48] Youngeun Kwon and Minsoo Rhu. Beyond the memory wall: A case for memory-centric HPC system for deep learning. In *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture*, page 148–161. IEEE Press, 2018. https://dl.acm.org/doi/10.1109/MICRO.2018.00021.

[49] Dusol Lee, Duwon Hong, Wonil Choi, and Jihong Kim. MQSim-E: An enterprise SSD simulator. *IEEE Computer Architecture Letters*, 21(1):13–16, 2022.

[50] Huaicheng Li, Daniel S. Berger, Stanko Novakovic, Lisa Hsu, Dan Ernst, Pantea Zardoshti, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. Pond: CXL-based memory pooling systems for cloud platforms. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, March 2023. https://arxiv.org/abs/2203.00241.

[51] Shengwen Liang, Ying Wang, Youyou Lu, Zhe Yang, Huawei Li, and Xiaowei Li. Cognitive SSD: A deep learning engine for in-storage data retrieval. In *Proceedings of the 2019 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 395–410. USENIX Association, 2019. https://dl.acm.org/doi/10.5555/3358807.3358841.

[52] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, page 267–278. Association for Computing Machinery, 2009. https://doi.org/10.1145/1555754.1555789.

[53] Hasan Al Maruf and Mosharaf Chowdhury. Effectively prefetching remote memory with leap. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 843–857. USENIX Association, 2020. https://www.usenix.org/conference/atc20/presentation/al-maruf.

[54] Theodore Michailidis, Alex Delis, and Mema Roussopoulos. Mega: Overcoming traditional problems with os huge page management. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, SYSTOR '19, page 121–131. Association for Computing Machinery, 2019. https://doi.org/10.1145/3319647.3325839.

[55] Pierre Michaud. Best-offset hardware prefetching. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 469–480, 2016. https://ieeexplore.ieee.org/document/7446087.

[56] Sparsh Mittal. A survey of recent prefetching techniques for processor caches. *ACM Comput. Surv.*, 49(2), 2016. https://dl.acm.org/doi/10.1145/2907071.

[57] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, page 89–100. Association for Computing Machinery, 2007. https://doi.org/10.1145/1250734.1250746.

[58] Ashish Panwar, Aravinda Prasad, and K. Gopinath. Making huge pages actually useful. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, page 679–692. Association for Computing Machinery, 2018. https://doi.org/10.1145/3173162.3173203.

[59] K. Parat. and A. Goda. Scaling trends in NAND flash. In *2018 IEEE International Electron Devices Meeting (IEDM)*, pages 2.1.1–2.1.4, 2018. https://ieeexplore.ieee.org/document/8614694.

[60] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jin-soo Kim, and Joonwon Lee. CFLRU: A replacement algorithm for flash memory. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, page 234–241. Association for Computing Machinery, 2006. https://dl.acm.org/doi/10.1145/1176760.1176789.

[61] Minsoo Rhu, Natalia Gimelshein, Jason Clemons, Arslan Zulfiqar, and Stephen W. Keckler. VDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1 – 13. IEEE Press, 2016.

[62] Christos Sakalis, Carl Leonardsson, Stefanos Kaxiras, and Alberto Ros. Splash-3: A properly synchronized benchmark suite for contemporary research. In *2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 101–111, 2016. https://ieeexplore.ieee.org/abstract/document/7482078.

[63] Mohit Saxena and Michael M. Swift. FlashVM: Virtual memory management on flash. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference*, USENIXATC'10, page 14. USENIX Association, 2010. https://www.usenix.org/conference/usenix-atc-10/flashvm-virtual-memory-management-flash.

[64] Carol Sliwa. SSD and NAND flash prices will decline through start of 2021, 2020. https://www.techtarget.com/searchstorage/news/252487918/SSD-and-NAND-flash-prices-will-decline-through-start-of-2021.

[65] Santhosh Srinath, Onur Mutlu, Hyesoon Kim, and Yale N. Patt. Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74, 2007. https://ieeexplore.ieee.org/document/4147648.

[66] Suhas Jayaram Subramanya, Harsha Vardhan Simhadri, Srajan Garg, Anil Kag, and Venkatesh Balasubramanian. BLAS-on-flash: An efficient alternative for large scale ML training and inference. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 469–484. USENIX Association, 2019. https://www.usenix.org/conference/nsdi19/presentation/subramanya.

[67] Tomoya Suzuki, Kazuhiro Hiwada, Hirotsugu Kajihara, Shintaro Sano, Shuou Nomura, and Tatsuo Shiozawa. Approaching dram performance by using microsecond-latency flash memory for small-sized random read accesses: A new access method and its graph applications. *Proc. VLDB Endow.*, 14(8):1311–1324, apr 2021. https://doi.org/10.14778/3457390.3457397.

[68] Arash Tavakkol, Juan Gómez-Luna, Mohammad Sadrosadati, Saugata Ghose, and Onur Mutlu. MQsim: A framework for enabling realistic studies of modern multi-queue SSD devices. In *Proceedings of the 16th USENIX Conference on File and Storage Technologies*, page 49–65. USENIX Association, 2018. https://www.usenix.org/conference/fast18/presentation/tavakkol.

[69] Shin-Yeh Tsai, Yizhou Shan, and Yiying Zhang. Disaggregating persistent memory and controlling them remotely: An exploration of passive disaggregated key-value stores. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 33–48. USENIX Association, 2020. https://www.usenix.org/conference/atc20/presentation/tsai.

[70] Iulia Turc, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Well-read students learn better: On the importance of pre-training compact models. *arXiv preprint arXiv:1908.08962v2*, 2019.

[71] Daniel Waddington, Moshik Hershcovitch, Swaminathan Sundararaman, and Clem Dickey. A case for using cache line deltas for high frequency VM snapshotting. In *Proceedings of the 13th Symposium on Cloud Computing*, SoCC '22, page 526–539. Association for Computing Machinery, 2022. https://dl.acm.org/doi/abs/10.1145/3542929.3563481.

[72] Ellis Herbert Wilson, Myoungsoo Jung, and Mahmut T. Kandemir. ZombieNAND: Resurrecting dead NAND flash for improved SSD longevity. In *IEEE 22nd International Symposium on Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 229–238, 2014. https://doi.org/10.1109/MASCOTS.2014.37.

[73] Michael Wu and Willy Zwaenepoel. Envy: A non-volatile, main memory storage system. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, page 86–97. Association for Computing Machinery, 1994. https://doi.org/10.1145/195473.195506.

[74] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *Proceedings of the 18th USENIX Conference on File and Storage Technologies*, page 169–182. USENIX Association, 2020. https://www.usenix.org/conference/fast20/presentation/yang.

[75] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Nam Sung Kim, Mahmut Taylan Kandemir, and Myoungsoo Jung. Revamping storage class memory with hardware automated memory-over-storage solution. In *Proceedings of the 48th Annual International Symposium on Computer Architecture*, ISCA '21, page 762–775. IEEE Press, 2021. https://dl.acm.org/doi/abs/10.1109/ISCA52012.2021.00065.

[76] Jie Zhang, Miryeong Kwon, Donghyun Gouk, Sungjoon Koh, Changlim Lee, Mohammad Alian, Myoungjun

Chun, Mahmut Taylan Kandemir, Nam Sung Kim, Ji-hong Kim, and Myoungsoo Jung. FlashShare: Punching through server storage stack from kernel to firmware for ultra-low latency SSDs. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI'18, page 477–492. USENIX Association, 2018. https://www.usenix.org/conference/osdi18/presentation/zhang.

[77] Tobias Ziegler, Carsten Binnig, and Viktor Leis. Scale-store: A fast and cost-efficient storage engine using dram, nvme, and rdma. SIGMOD '22, page 685–699. Association for Computing Machinery, 2022. https://doi.org/10.1145/3514221.3526187.