Transparent DAX Mappings: Towards Automatic Kernel Bypass with CXL-Based Hybrid SSDs

Yussuf Khalil
Samsung
Copenhagen, Denmark
Karlsruhe Institute of Technology
Karlsruhe, Germany

Daniel Habicht
Karlsruhe Institute of Technology
Karlsruhe, Germany

Pascal Ellinger
Karlsruhe Institute of Technology
Karlsruhe, Germany

Frank Bellosa
Karlsruhe Institute of Technology
Karlsruhe, Germany

Javier González
Samsung
Copenhagen, Denmark

Adam Manzanares
Samsung
San Jose, CA, US

Vivek Shah
Samsung
Copenhagen, Denmark

Abstract

Upcoming CXL-based hybrid SSDs offer a persistent memory interface in addition to traditional block-based communication. Previous research argued that the non-uniform access latencies (on-device DRAM cache hits vs. misses) of these devices call for new OS-based resource management strategies as existing DAX mechanisms were not built with this device model in mind. In this work, we sketch *Transparent DAX Mappings*, a novel approach to enable a broad range of applications to benefit from hybrid SSDs. By opportunistically handing out DAX mappings to processes through a collaboration between the kernel and the C standard library, we aim to achieve automatic kernel bypass for unmodified I/O-heavy applications. We evaluate a preliminary implementation of our design on a Samsung CMM-H prototype and demonstrate up to 96.2 % increased throughput in *Valkey*.

CCS Concepts: • **Hardware** → **Memory and dense storage**; **Non-volatile memory**; • **Software and its engineering** → *File systems management*; **Memory management**; • **Information systems** → **Storage management**.

Keywords: Hybrid SSDs, CXL, DAX, Kernel Bypass

ACM Reference Format:
Yussuf Khalil, Daniel Habicht, Pascal Ellinger, Frank Bellosa, Javier González, Adam Manzanares, and Vivek Shah. 2025. Transparent DAX Mappings: Towards Automatic Kernel Bypass with CXL-Based Hybrid SSDs. In *Workshop on Disruptive Memory Systems (DIMES '25), October 13–16, 2025, Seoul, Republic of Korea.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3764862.3768178

DIMES '25, October 13-16, 2025, Seoul, Republic of Korea

ACM ISBN 979-8-4007-2226-4/25/10
https://doi.org/10.1145/3764862.3768178

1 Introduction

Compute Express Link (CXL) [7] is a recent interconnect standard that offers memory semantics (i.e., byte addressing and cacheability) for accessing attached devices. Several vendors have announced CXL-based persistent memory technologies, e.g., Samsung [45] and Wolley [37, 38]. These devices combine NAND flash with a DRAM cache and guarantee persistence for cache contents via CXL's *Global Persistent Flush* (GPF) [5] feature. Generally, these devices are expected to offer a NVMe-like (i.e., block-granular and asynchronous) interface in addition to memory semantics, hence we call them *hybrid SSDs*.

Intel *Optane DCPMM* [8, 14] is an earlier technology that offered persistent memory semantics as well. While its lower price compared to DRAM [14] made it an interesting candidate for system memory, it was also much more expensive than NAND-based SSDs [12]. In contrast, hybrid SSDs are expected to be at a similar price point as NVMe SSDs ($0.23 vs. $0.20 per GB for Samsung CMM-H and NVMe, respectively [45]) because they also use cost-effective NAND flash for providing storage. Further, Optane competed with DRAM-based memory for the same DIMM slot capacity, while being unable to fully replace DRAM due to its performance characteristics [35, 39, 41]. In the context of Optane as a storage solution, this presented a trade-off between system memory and storage capacity. Hybrid SSDs, on the other hand, can potentially fulfill the same purpose as NVMe-based SSDs with similar baseline performance while also not competing with system memory for DIMM slots.

For these reasons, we advocate for exploring how CXL-based hybrid SSDs can serve in a *general-purpose* storage setting while leveraging their persistent memory properties. Whereas Optane was considered an additional tier in the memory hierarchy [14], we aim for a solution where hybrid SSDs can be a *drop-in replacement* for traditional NVMe SSDs. As conjectured by Habicht et al. [11] using an emulated device, and as we will empirically demonstrate with

real hardware in § 2, memory accesses to a hybrid SSD provide strong performance guarantees only when served by its DRAM cache. Hence, it is mandatory to design new resource management strategies for exploiting the scarce persistent cache.

We propose *Transparent DAX Mappings* (TDMs) as a novel approach to deliver the advantages of employing storage with memory semantics to unmodified applications using the traditional POSIX file interface. By extending the existing DAX mechanism in Linux with profiling-guided device cache management for hybrid SSDs and automatic distribution of cache resources, TDMs enable `read()`/`write()` calls to entirely bypass the kernel. Our contributions are as follows:

- We define a set of requirements for broad applicability of hybrid SSDs (§ 4)
- We propose TDMs to automatically enable kernel bypass for unmodified applications with POSIX I/O (§ 4.2)
- We sketch a lightweight profiling-guided policy to select candidates for TDMs (§ 4.3)
- We outline how TDMs can be extended for file append operations (§ 4.4)
- We implement a simple prototype to test the viability of our design (§ 5)
- We evaluate our prototype on real hardware using the *Valkey* datastore (§ 6)

2 Samsung CMM-H

We employ a Samsung CMM-H (*CXL Memory Module – Hybrid*) [30] prototype as the foundation for our design. Zeng et al. [45] provide an overview of the performance characteristics of an earlier prototype. The prototype they used for their evaluation was based on an AMD Versal FPGA with a 1 TB NVMe SSD as backing storage and 16 GiB of DRAM. In their work, they argue that a more specifically tailored NAND architecture may benefit the achievable performance.

We use a more recent prototype [30] that has its digital logic implemented in an Altera Agilex 7 FPGA and features 48 GiB of DRAM with a CXL 2.0 ×8 interface. However, it still employs an internal PCIe 4.0 ×4 bus to communicate with a commodity Samsung PM9A3 960 GB NVMe SSD [29] as its backing storage. From the host perspective, CMM-H presents itself as a *Type 3* device according to the CXL nomenclature [5]. In turn, the entire storage capacity, i.e., 960 GB, is exposed as a memory-only NUMA node and can be accessed via `CXL.mem` transactions. CXL's *Global Persistent Flush* (GPF) feature [5] is available to guarantee persistence for data stored in on-device DRAM and in CPU caches. The prototype features an internal cache manager that performs LRU-style evictions from the device's DRAM cache. However, the host may also perform explicit prefetch and evict operations via a `CXL.io`-based API to manage the cache. Support for DMA-based block-granular asynchronous transfers via `CXL.io` is expected in a future version of the prototype. With a DMA protocol, it should generally be possible to build a block device driver that allows to use the device as if it were a traditional SSD. Lin et al. [23] provide an analysis of the performance characteristics of this prototype.

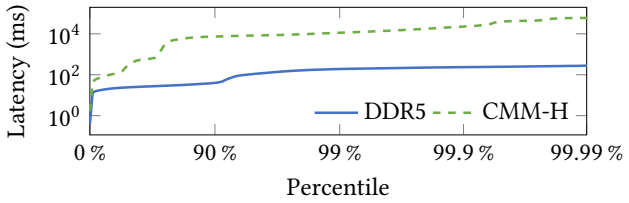

Figure 1. Latency percentiles from a 100 % `SET` *Memtier* [27] benchmark. CMM-H exhibits worst-case latencies two orders of magnitude higher than CPU-local DDR5 memory.

Conceptually, CMM-H could also be used as system or application memory. Its reliance on NAND flash as backing storage, however, can create prolonged access latencies that lead to potentially intolerable CPU stalls. To assess its suitability as memory, we perform a benchmark that aims for a worst-case scenario. Figure 1 shows the latency distribution when executing 3.5 billion `SET` operations across 24 threads on pairwise distinct keys with 8 byte values using *Memtier* [27] on an empty *Valkey* [33] in-memory database with persistence disabled. This configuration results in a working set size of 189 GiB. When using CMM-H as the sole memory for *Valkey*, latencies are up to two orders of magnitude higher than when using CPU-local DDR5 memory.

3 Related Work

Early works by Bae et al. [3] initially explored the idea of SSDs with byte-granular access interfaces. Abulila et al. [1] suggested *FlatFlash*, a solution for embedding hybrid SSDs in a tiered memory/storage hierarchy. Jung [15] proposed using CXL for hybrid SSDs. Kwon et al. [19] present *ExPAND* as a design to perform last-level cache prefetching on a CXL-enabled SSD. *SkyByte* [47] divides a hybrid SSD's DRAM into a write log with cacheline-sized entries and a page-granular data cache. Yang et al. [43] further explore the design space for CXL-based hybrid SSDs and the potential of caching and prefetching techniques. Lee et al. [20] analyze `TRIM` commands in the context of CXL-attached flash storage. Lim et al. [22] propose data structure optimizations for databases employing CMM-H. Song [31] suggests that CMM-H could be used in conjunction with GPUs for AI workloads. Soltaniyeh et al. [30] explore its feasibility as large, but cheap memory expansion given the low cost of NAND vs. DRAM.

Operating system-level works on hybrid SSDs have primarily focused on the area of file systems so far. *DJFS* [44] enables file system journaling on a directory-granular level, tailored for CMM-H's limited cache capacity. Zhan et al. [46] propose *RomeFS*, a file system that utilizes both the block and the byte-granular interfaces on hybrid SSDs. For this,

they partition the available storage space into different zones according to latency requirements. *ByteFS* [21] is another approach that employs the SSD's DRAM for log-structured writes at cacheline granularity and instead refrains from caching page data on the device.

Habicht et al. [11] have shown that careful management of the device's DRAM cache is necessary to profit from having a synchronous byte-addressable interface on an SSD. They propose to extend the existing DAX interfaces in commodity operating systems with more fine-granular mapping capabilities in contrast to their current file-level (Linux) or file system-level (Windows) granularity. Further, their design includes the device's persistent DRAM cache into the operating system's page cache (*persistence-aware page cache*) to eliminate block layer operations for writing back file contents while maintaining a coherent view between memory-mapped I/O and POSIX I/O.

Kernel bypass for file operations on persistent memory was previously proposed in *SplitFS* [16], *FLEX* [40], and *MadFS* [49]. However, these approaches were designed in the context of Optane and lack the resource management capabilities necessary for CXL-based hybrid SSDs.

Wolley is another vendor that has announced CXL-enabled hybrid SSDs, dubbed *NVMe-over-CXL* [38]. Their design relies on the traditional NVMe standard for communicating with SSDs, however, they propose exposing NVMe's *Controller Memory Buffer* (CMB) [26] via CXL [9].

4 Approach

As described in § 1, we aim at building an operating system mechanism that makes CXL-based hybrid SSDs employable in a *general-purpose* fashion as a *drop-in replacement* for traditional NVMe SSDs. Hence, we assume a simple system architecture that has no constraints other than having a CXL-capable CPU and at least one NVMe SSD with an OS-managed file system and OS-managed CPU applications. We define the following requirements for our mechanism:

- **R1** A performance and/or energy benefit must be achieved in a broad range of scenarios.
- **R2** Performance must never be worse than with NVMe.
- **R3** Must be *simple* to use, i.e., not require additional developer or operator effort.

In alignment with requirement **R3**, we replace the NVMe SSDs in the aforementioned architecture with CXL-based SSDs. The amount of consumed CPU interconnect resources (i.e., PCIe/CXL lanes) stays thereby unchanged, which is a subtle, yet important difference to Optane-based persistent memory from a deployment perspective as outlined in § 1.

In the next subsection, we start by reviewing Linux's current DAX interface for persistent memory and how previous work has proposed to adapt it for hybrid SSDs. Then, we iterate over the components of our approach and describe how they align with the established requirements.

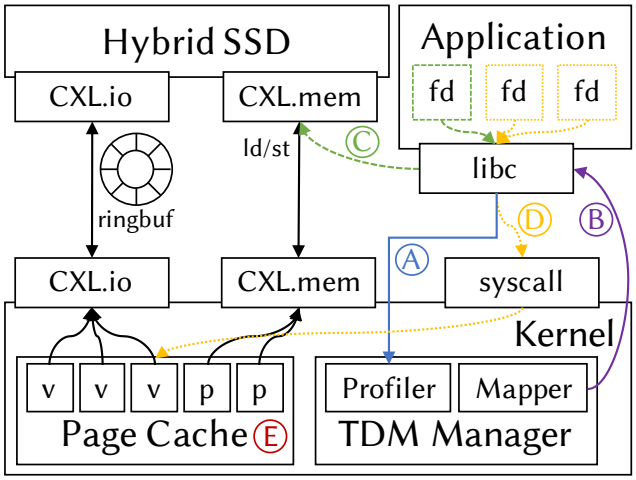

Figure 2. Overview over our proposed design. The hybrid SSD can be accessed asynchronously via `CXL.io` or synchronously via `CXL.mem`. The *TDM Manager* collects profiling data Ⓐ for each file descriptor (fd) provided by `libc` in user-space (§ 4.3). Then, it uses the data to select files that are promoted to transparent DAX and communicates these to `libc` Ⓑ (§ 4.2). POSIX I/O can then be handled in user space via `CXL.mem` Ⓒ, whereas other file handles continue using the traditional system call path Ⓓ. The page cache contains both volatile (v) and persistent (p) memory pages to guarantee coherence between the various I/O paths Ⓔ.

4.1 Direct Access (DAX) for Hybrid SSDs

Linux's current *Direct Access* (DAX) interface for persistent memory comes in two flavors: `devdax` and `fsdax` [17]. `fsdax` is used to bypass the operating system's page cache when calling `mmap()` on a file descriptor, whereas `devdax` is used for raw device access without a file system. To use `fsdax`, system operators set a specific per-file flag in the file system. As per the system architecture defined in § 4, we focus solely on `fsdax`, i.e., a traditional file-based storage abstraction. In this case, virtual mappings in a process point directly to the physical pages belonging to a file on persistent memory.

Habicht et al. [11] showed that the DAX interface has several shortcomings that make it unsuitable for hybrid SSDs, as it was designed with Optane in mind. More precisely, the non-uniform access latencies caused by the difference between DRAM cache hits and misses may cause prolonged CPU stalls since `CXL.mem` accesses are synchronous (as shown in § 2). Therefore, it is prohibitive to blindly map arbitrary device addresses into user space. Instead, they propose adding a new `MAP_DAX` flag to `mmap()` to give applications fine-granular control about whether to use DAX. Via `mlock()`, applications can then request pinning DAX-mapped pages to the device's persistent cache. `CXL.mem` transactions are thereby

guaranteed to have predictable latencies. Non-DAX operations can be handled via the traditional block layer with a `CXL.io`-based DMA protocol.

Arguably, this concept introduces a new burden on application developers, violating requirement **R3** of being *simple* to use. Explicit DAX in applications can undoubtedly yield the best performance. In light of requirement **R1** of broad applicability, however, we argue with the 80/20 principle [18] of achieving *most* of the potential result with only little work.

4.2 Transparent DAX Mappings

Transparent DAX Mappings (TDMs) aim to provide unmodified applications with the benefits of persistent caches and bypassing the kernel for I/O on hybrid SSDs. Figure 2 provides an overview of our design. While keeping it possible for applications to explicitly claim DRAM space on the hybrid SSD device in the way described in § 4.1, our approach further distributes unclaimed space to DAX-unaware applications. For this purpose, we introduce a new *TDM Manager* into the OS kernel for centralized resource management. The TDM manager keeps track of available device DRAM space and selects files for *promotion* and *demotion*. POSIX I/O on promoted files can leverage direct access to the device's persistent DRAM capacity in a manner that is *transparent* to the application. We begin by explaining the promotion scheme and sketch the selection policy in more detail in § 4.3.

When a file is *promoted*, the TDM manager determines the physical addresses belonging to this file on the device. These address ranges are then pinned into the hybrid SSD's DRAM in order to provide strong access latency guarantees. Next, it establishes mappings in the virtual address spaces of all processes referring to that file. Finally, it communicates these mappings to the `libc` via shared memory Ⓑ. The `libc` can then transparently handle `read()`/`write()` POSIX calls using `memcpy()` operations Ⓒ instead of going through the system call path Ⓓ. This enables a broad range of applications to potentially benefit from kernel bypass (**R1**) without application developer effort (**R3**).

Demotion becomes necessary whenever an application explicitly requests a DAX mapping (as in § 4.1) and device DRAM space needs to be reclaimed to serve the request. Further, the TDM selection mechanism (§ 4.3) may choose to promote another file instead at any time. For demotion, the operations described in the previous paragraph are rolled back so that operations are handled via system calls again.

To ensure coherence between TDMs, explicit DAX mappings, traditional `mmap()`, and traditional POSIX I/O, we insert persistent pages from the device's cache into the operating system's page cache Ⓔ. This is compatible with the design for explicit DAX proposed by Habicht et al. [11] and further eliminates the need for explicit synchronization during promotion. When promoting a file already present in the volatile page cache with dirty contents, it can be migrated into the device cache synchronously via `memcpy()`.

4.3 Profiling-Guided Selection

How to choose files for TDM promotion? is a crucial question for fulfilling requirements **R1** and **R2**, i.e., general-purpose performance benefits and no performance penalty compared to NVMe. The performance benefit we aim for stems from handling as many file operations as possible via `CXL.mem` without system calls (**R1**), but with strong latency guarantees (**R2**) through limiting `CXL.mem` to the device DRAM. Operations on non-promoted files continue to be handled via a `CXL.io`-based block-granular and asynchronous interface. We generally assume that operations on these files achieve similar performance as with NVMe over PCIe given similar SSD controllers. However, management overhead for TDMs, including file promotion or demotion, or unnecessary CXL bus traffic may be detrimental for the overall performance.

To perform promotion candidate selection, we hence propose a lightweight profiling mechanism built around `libc`. `libc` as a central intermediary on the function call path for all POSIX I/O has full knowledge of all operations performed both via system calls *and* via transparent DAX. Hence, it can easily keep track of the total read and write bandwidths per file descriptor over a sliding window with $O(1)$ space and time overhead. This bandwidth data can serve as a prediction for future file requests. By using a sliding window, we aim to avoid being influenced by large one-off events to make the prediction more robust. In addition, we plan to evaluate whether other metrics are also worthwhile to track for profiling. For example, monitoring `fsync()` calls was previously proposed by *Ziggurat* [48] in the context of tiered storage. We communicate the gathered information to the in-kernel TDM manager via shared memory Ⓐ.

The TDM manager is responsible for deciding the set of files for transparent DAX. To that end, we aim for finding a set that maximizes the expected bandwidth across files while keeping the total size of DAX-mapped files below the on-device DRAM size (i.e., to achieve strong latency guarantees). We propose sorting the list of files by bandwidth (as reported by `libc`) and greedily filling the set starting from the largest. This approximation requires $O(n \log n)$ time and $O(n)$ space overhead [25] and should therefore be reasonably fast (**R2**). To further reduce practical runtime overhead, we suggest ignoring file descriptors with bandwidth predictions below a certain threshold during file set calculation. Recalculating the set is only necessary upon significant bandwidth changes. Files added to the set get *promoted*, while those removed from the set are *demoted* (§ 4.2).

4.4 File Append Operations

Supporting file append operations without involving the file system driver (i.e., maintaining kernel bypass) is a challenging endeavor, yet mandated by requirement **R1** of covering many use cases. Some applications, e.g., *Valkey* with its

append-only file (AOF) persistence mode [34], create significant bandwidth solely through append operations. Due to their bandwidth, we consider these interesting candidates for TDMs and hence want to extend TDM support to append-heavy scenarios. *SplitFS* [16], an earlier Optane-focused approach, implements file appends in user space by introducing a temporary file for applications to write append data into. New data is then integrated into the file system asynchronously by the kernel through swapping extents.

We propose an approach with a similar concept, however, the resource management necessary for hybrid SSDs makes this significantly more complicated. To avoid intolerable `CXL.mem` latencies, newly appended data must be stored in the device DRAM, i.e., it competes for the scarce DRAM space with other (explicit or transparent) DAX mappings to existing file blocks. For this reason, we suggest monitoring for append bandwidth separately from (non-append) write bandwidth during profiling. Predictions based on this information can be used to reserve buffer space for append data in device DRAM if the file selection policy considers the append bandwidth high enough (§ 4.3).

5 Implementation

To assess the viability of our approach, we implement a simplified prototype that focuses on providing kernel bypass for `write()` operations on TDM-enabled files (§ 4.2) as well as support for append operations (§ 4.4). Currently, our early-stage prototype does not employ profiling-guided TDM selection (§ 4.3) but instead relies on manual, application-specific configuration. In order to support writes, we use a write-allocate block cache (managed by a user space library) that serves them via `memcpy()`. Our final implementation plans to use this cache primarily for appends (as described in § 4.4), however, we currently also use it for non-appending writes as it provides similar kernel bypass capabilities as our proposed design for these, thus allowing us to focus on evaluating the potential benefit of TDMs.

The current prototype consists of two main components: `libtdm`, the first component, is loaded into processes via `LD_PRELOAD` and hooks into `open()`, `read()`, and `write()` calls. The library manages the aforementioned block cache entirely in user space. Due to the write-allocate policy, read-only data does not engage the kernel bypass. While this does not matter for writes, it presents a deviation from our intended design that also considers TDMs for `read()`. On process startup, a fixed-size cache on the hybrid SSD is allocated and prefetched into its on-device DRAM cache using the API provided by the device (§ 2). Our `open()` implementation decides whether to enable kernel bypass when the path matches a manually pre-configured one. `write()` operations are then served via the cache, whereas cache misses during `read()` calls are handled via the system call path. When using TDMs, we can bypass the kernel on `fdatasync()` as

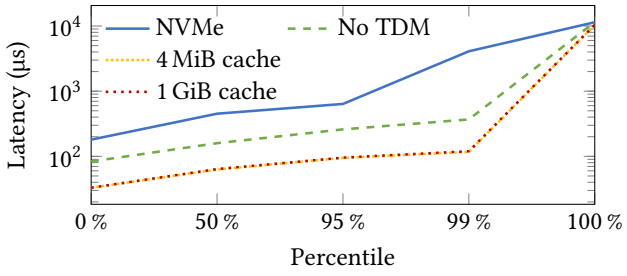

Figure 3. Request latency percentiles during *YCSB* [6] initialization with *Valkey* [33] (`fsync=always`). Our prototype improves 99^{th} percentile latencies by 67.5 %.

all file blocks are in the device's persistent cache already. We employ the *S3-FIFO* [42] algorithm for cache eviction. A background thread writes evicted blocks back by calling into the file system. In order to avoid stalling block allocations due to outstanding write-backs, `libtdm` maintains a small pool of free blocks.

The second component is `ktdm`, a kernel module that keeps track of all block cache instances. When a process exits with dirty blocks in its block cache, `ktdm` synchronizes all dirty contents with the backing file. This avoids data loss when a process crashes. In case the file saw append operations (§ 4.4), `ktdm` further handles resizing the file to the new size.

Other than lacking automatic TDM selection and full kernel bypass for `read()` operations, our current prototype is further incapable of maintaining coherence across processes. In turn, concurrent processes may not access the same file if it was configured for TDM. As described in § 4.2, we plan to alleviate this issue by employing the page cache for guaranteeing a coherent view on TDM-enabled files between multiple processes. Nonetheless, our prototype allows us to evaluate the effectiveness of bypassing the kernel particularly for `write()`. A full implementation of our approach will further allow us to assess the impact of kernel bypass for `read()` as well as the quality of our TDM selection policy.

In our evaluation (§ 6), we configure TDMs to be enabled for files we consider *hot*. We expect that a full implementation of our proposed automatic selection mechanism (§ 4.3) would make the same choices.

6 Evaluation

We perform an evaluation of our prototype described in § 5 using the core workloads of the *Yahoo! Cloud Serving Benchmark* (YCSB) [6] with the key-value datastore *Valkey* [33]. Our test system is equipped with two AMD EPYC 9454 (48 cores @2.75 GHz) CPUs [2] with 256 GiB of DDR5-4800 memory each and a Samsung CMM-H prototype as described in § 2. A Samsung PM9A3 960 GB NVMe SSD [29] serves as a baseline to compare storage performance. We run Ubuntu 24.04 [4] with Linux 6.16-rc4 [32] on our test system. All results presented in this section are averaged over 30 runs.

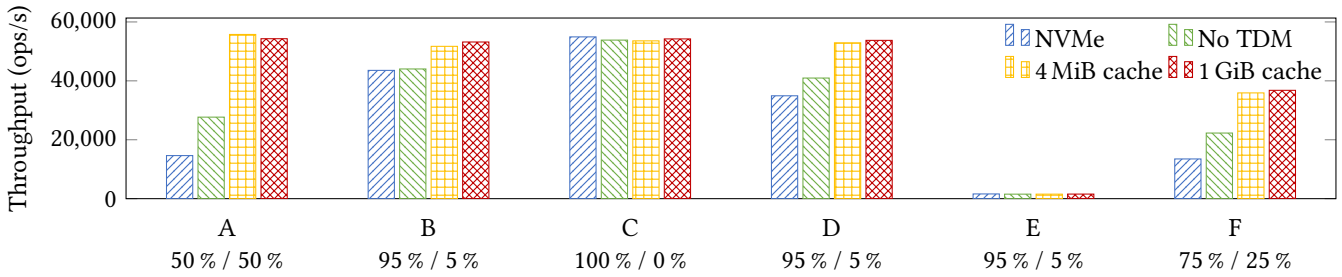

Figure 4. Request throughput for *YCSB* [6] workloads *A* – *F* with *Valkey* [33] (`fsync=always`). Percentages indicate ratio between read and write operations for each workload. Write-heavy scenarios (e.g., *A*) are up to 96.2 % faster with our prototype.

In contrast to § 2, where we benchmarked *Valkey* using a purely in-memory configuration, we employ *Valkey*'s *append-only file* (AOF) persistence mode here [34]. In this mode, all write operations are appended to a file (i.e., the AOF) such that the dataset can be recovered by replaying writes from this file. *Valkey* allows configuring the frequency of `fsync()` calls to cater to the user's requirements. For our evaluation, we configure `fsync=always`, i.e., every single AOF write is synchronized immediately [1]. As our current prototype does not support sharing TDM-enabled files between processes, we disable *Valkey*'s AOF rewrite and snapshotting features as they run in forked-off processes. We configure `libtdm` (§ 5) to select all files in the AOF directory and set the cache size to either 0 B (no TDM), 4 MiB, or 1 GiB. With TDMs disabled, all file operations are handled via the kernel path using Linux's traditional page cache implementation. In this case, given the lack of a DMA interface in the current prototype revision (§ 2), `fsync()` calls lead to flushing the page cache via memory operations instead of DMA.

Figure 3 shows latency percentiles for insert operations during the loading (i.e., write-only) phase of *YCSB* workload *A*. Even without our TDM prototype, CMM-H's large hardware-managed cache enables latency improvements up to 91.0 % compared to the NVMe SSD in our system without any changes to *Valkey* or the operating system. With TDMs enabled, we can further improve median latencies by 60.0 % and by 67.5 % in the 99^{th} percentile. Notably, despite the AOF reaching a size of 116 MiB, there is no significant difference in latencies between 4 MiB and 1 GiB cache sizes. This demonstrates that keeping a small number of blocks for append operations is sufficient to achieve the maximum performance benefit.

Figure 4 depicts the request throughput across the various *YCSB* workloads as well as their respective read/write ratios. Similar to the latencies in Figure 3, throughput is already increased by up to 89.3 % in these workloads without TDMs thanks to the device's large cache without software changes.

[1] A single write to the AOF may contain multiple write operations from different clients or pipelined execution [34]. *YCSB* does not make use of pipelining. We employ two threads for requests.

`fsync`	TDM	Geometric Mean
`always`	Disabled	21239
`always`	4 MiB cache	27686
`no`	Disabled	27499

Table 1. Geometric mean throughput across all *YCSB* [6] workloads for different configurations. With TDMs, `fsync=alyways` achieves similar performance as `fsync=no`.

With TDMs, write-heavy workloads such as *A* or *F* show a further improvement of 96.2 % and 65.1 %, respectively. However, even read-dominated workloads with only few write operations show a large increase of 20.8 % (*B*) and 31.2 % (*D*) with TDMs which we attribute to the large performance burden imposed by `fsync()` calls otherwise. Unsurprisingly, read-only workloads (*C*) do not show a significant difference.

Although our implementation is in an early stage, we believe the demonstrated results show the viability of our proposed approach. Table 1 compares the geometric means across all *YCSB* [6] workloads with weak (`fsync=no`) and strong persistence (`fsync=always`) on CMM-H. With TDMs, the overhead for strong persistence guarantees diminishes, without requiring any changes to *Valkey*. We attribute the minor increase of 0.6 % seen with TDMs compared to `fsync=no` to measurement inaccuracies.

7 Future Work

The design presented in this work is still in an early stage and not yet complete. Most obviously, an implementation and evaluation of our proposed kernel-based profiling-guided resource management as described in § 4.3 is still missing.

Other than that, we plan to implement support for additional OS features such as `mmap()` and `inotify`. The latter can be solved by asynchronously notifying the kernel about write operations, which further allows it to perform metadata updates during runtime. `mmap()`, however, makes profiling more complicated. To this end, we aim to explore a performance counter-based profiling approach.

Remodeling Linux's `read()` and `write()` semantics for TDMs without breaking existing applications is challenging. For example, many applications rely on sector atomicity of writes [28], which our current implementation does not provide.

Another challenge regarding TDMs concerns file system-level persistence. While mechanisms like GPF can guarantee that writes are persisted on the hybrid SSD in case of a power outage, users might still not be able to recover the written data from a file due to file metadata being lost. Linux provides a solution for this issue by offering the `MAP_SYNC` flag that is part of the *synchronous page faults* mechanism. For shared DAX mappings, `MAP_SYNC` guarantees that while file contents are mapped writable into user space, these file contents are accessible at the same position even after a crash or reboot [24], meaning that metadata required for reading from this file offset is persisted. If file metadata required for retrieving writes is still unstable, `MAP_SYNC` forces a flush of this metadata when writing to the DAX mapping [36]. For TDMs we intend to reuse this mechanism for ensuring file system-level persistence of writes.

Lin et al. [23] show that the current CMM-H prototype (§ 2) can reach up to 25 GB/s read bandwidth when hitting the on-device cache, while the write bandwidth is significantly lower. Based on this data, we suggest that the TDM manager should consider `CXL.mem` bandwidth as another constrained resource in addition to cache capacity. When the TDM manager does not account for bandwidth consumed by TDMs, applications explicitly using DAX via `mmap()` could suffer from a decrease in quality of service. This, however, contradicts the idea of using TDMs on otherwise unused resources. With upcoming features in Intel's *Memory Bandwidth Monitoring* (MBM) [13, §3.1.4.2] and *Memory Bandwidth Allocation* (MBA) [13, §3.2.4.4] technologies, we aim to build an accounting scheme that measures per-task `CXL.mem` bandwidth. Then, to counteract an overload caused by deploying TDMs too aggressively, the TDM manager can either disengage certain TDMs or use a mechanism like MBA to throttle the TDM bandwidth on a per-task basis.

As an additional enhancement, we want to investigate how to extend our approach to partial files while keeping the profiling overhead low. This may be useful, e.g., for database applications that store all tables in a single file as opposed to creating one file per table.

This work has focused on writes, however, TDMs may also be useful for reads. Previous works argued for serving reads via `CXL.mem`, based on the observation that some workloads typically only need small portions of a block [10]. In turn, TDMs could help to reduce read amplification and bus traffic.

Several other open questions remain that touch hybrid SSDs in a more general sense. For example, *How to efficiently virtualize hybrid SSDs for multiple tenants?*, or *How to build a RAID out of hybrid SSDs with two different interfaces?*

8 Conclusion

In this work, we proposed the concept of *Transparent DAX Mappings* (TDMs) as a mechanism to automatically manage and distribute the scarce persistent DRAM capacity of CXL-based hybrid SSDs. Only the DRAM can be accessed with low enough latency via `CXL.mem` to prevent CPU stalls and hence justifies careful and centralized resource management. With TDMs, the kernel identifies file handles that benefit the most from kernel bypass and feeds unmodified applications with direct access mappings during runtime. While our implementation is not yet complete, preliminary results show up to 96.2 % increased throughput in *Valkey*. Our work makes the case for managing the device's persistent cache entirely within the operating system. We hence argue that hybrid SSDs should feature an API that allows the host to pin pages in the device's cache. However, more work is necessary to answer whether hybrid SSDs can fully serve as a drop-in replacement for traditional NVMe drives.

References

[1] Ahmed Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-mei Hwu. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 971–985. doi:10.1145/3297858.3304061

[2] Advanced Micro Devices, Inc. 2022. *AMD EPYC™ 9454*. https://www.amd.com/en/products/processors/server/epyc/4th-generation-9004-and-8004-series/amd-epyc-9454.html

[3] Duck-Ho Bae, Insoon Jo, Youra Adel Choi, Joo-Young Hwang, Sangyeun Cho, Dong-Gi Lee, and Jaeheon Jeong. 2018. 2B-SSD: The Case for Dual, Byte- and Block-Addressable Solid-State Drives. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 425–438. doi:10.1109/ISCA.2018.00043

[4] Canonical Ltd. 2024. *Ubuntu 24.04*. https://ubuntu.com/

[5] Compute Express Link Consortium, Inc. 2024. *Compute Express Link Specification Revision 3.2*.

[6] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (Indianapolis, Indiana, USA) *(SoCC '10)*. Association for Computing Machinery, New York, NY, USA, 143–154. doi:10.1145/1807128.1807152

[7] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.* 56, 11, Article 290 (July 2024), 37 pages. doi:10.1145/3669900

[8] Peter Desnoyers, Ian Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Mike Mesnier, Carl Waldspurger, Avani Wildani, and Erez Zadok. 2023. Persistent Memory Research in the Post-Optane Era. In *Proceedings of the 1st Workshop on Disruptive Memory Systems* (Koblenz, Germany) *(DIMES '23)*. Association for Computing Machinery, New York, NY, USA, 23–30. doi:10.1145/3609308.3625268

[9] Bill Gervasi. 2024. *NVMe Over CXL*. https://files.futurememorystorage.com/proceedings/2024/20240806_DCTR-102-1_Gervasi.pdf

[10] Bill Gervasi and San Chang. 2024. *NVMe Over CXL™ Defines Memory Class Storage to Improve System Performance*. http://wolleytech.com/wp-content/uploads/2024/09/wolley_nvme_over_cxl_white_paper.pdf

[11] Daniel Habicht, Yussuf Khalil, Lukas Werling, Thorsten Gröninger, and Frank Bellosa. 2024. Fundamental OS Design Considerations for CXL-based Hybrid SSDs. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems* (Austin, TX, USA) *(DIMES '24)*. Association for Computing Machinery, New York, NY, USA, 51–59. doi:10.1145/3698783.3699380

[12] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. 2022. SSDs Striking Back: The Storage Jungle and Its Implications to Persistent Indexes.. In *CIDR*, Vol. 22. 1–8. https://vldb.org/cidrdb/papers/2022/p64-huang.pdf

[13] Intel Corporation. 2025. Intel® Resource Director Technology (Intel® RDT) Architecture Specification. https://cdrdv2.intel.com/v1/dl/getContent/851356?fileName=356688-003-intel-rdt-architecture-spec.pdf

[14] Intel Corporation. 2019. *Intel® Optane™ Persistent Memory Product Brief*. https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html

[15] Myoungsoo Jung. 2022. Hello bytes, bye blocks: PCIe storage meets compute express link for memory expansion (CXL-SSD). In *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems* (Virtual Event) *(HotStorage '22)*. Association for Computing Machinery, New York, NY, USA, 45–51. doi:10.1145/3538643.3539745

[16] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 494–508. doi:10.1145/3341301.3359631

[17] Linux kernel contributors. 2023. *Direct Access for Files – The Linux Kernel Documentation*. https://docs.kernel.org/6.15/filesystems/dax.html

[18] Richard Koch. 2011. *The 80/20 Principle: The Secret of Achieving More with Less: Updated 20th anniversary edition of the productivity and business classic*. Hachette UK.

[19] 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* (Boston, MA, USA) *(HotStorage '23)*. Association for Computing Machinery, New York, NY, USA, 24–30. doi:10.1145/3599691.3603406

[20] Hayan Lee, Jungwoo Kim, Wookyung Lee, Juhyung Park, Sanghyuk Jung, Jinki Han, Bryan S. Kim, Sungjin Lee, and Eunji Lee. 2025. Revisiting Trim for CXL Memory. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage '25)*. Association for Computing Machinery, New York, NY, USA, 24–30. doi:10.1145/3736548.3737825

[21] Shaobo Li, Yirui (Eric) Zhou, Hao Ren, and Jian Huang. 2025. ByteFS: System Support for (CXL-based) Memory-Semantic Solid-State Drives. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 116–132. doi:10.1145/3669940.3707250

[22] Seungho Lim, Seung Won Yoo, Joontaek Oh, Wonseb Jeong, Hyunsub Song, Hyeonho Song, Donghun Lee, and Youjip Won. 2024. Key-Space Partitioned LSM Tree for CMM-H. In *2024 13th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*. 1–6. doi:10.1109/NVMSA63038.2024.10693654

[23] Zhen Lin, Yujie Yang, Lingfeng Xiang, Lianjie Cao, Faraz Ahmed, Jia Rao, Hui Lu, and Puneet Sharma. 2025. Can Hardware Outsmart Software in Tiered Memory Management? A CMM-H Case Study. In *Proceedings of the 18th ACM International Systems and Storage Conference* (Virtual, Israel) *(SYSTOR '25)*. Association for Computing Machinery, New York, NY, USA, 85–91. doi:10.1145/3757347.3759140

[24] Linux man-pages project. 2025. *mmap(2) – Linux manual page*. https://man7.org/linux/man-pages/man2/mmap.2.html Linux man-pages 6.15.

[25] Kurt Mehlhorn and Peter Sanders. 2008. *Algorithms and Data Structures: The Basic Toolbox* (1 ed.). Springer Publishing Company, Incorporated. doi:10.1007/978-3-540-77978-0

[26] NVM Express, Inc. 2025. *NVM Express® Base Specification Revision 2.2*. https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-Revision-2.2-2025.03.11-Ratified.pdf

[27] Redis Ltd. 2025. *memtier_benchmark*. https://github.com/RedisLabs/memtier_benchmark

[28] Andy Rudoff. 2019. *Protecting Software from Itself: Powerfail Atomicity for Block Writes*. https://pirl.nvsl.io/PIRL2019-content/PIRL-2019-Andy-Rudoff.pdf

[29] Samsung Semiconductor. 2022. *MZQL2960HCJR-00A07(960GB) | SSD | Samsung Semiconductor Global*. https://semiconductor.samsung.com/ssd/datacenter-ssd/pm9a3/mzql2960hcjr-00a07/

[30] Mohammadreza Soltaniyeh, Gongjin Sun, Xuebin Yao, Amir Beygi, Ramdas Kachare, Dongwan Zhao, Hingkwan Huen, Andrew Chang, Senthil Murugesapandian, and Caroline Kahn. 2025. Revisiting Memory Hierarchies with CMM-H: Use Device-side Caching to Integrate DRAM and SSD for a Hybrid CXL Memory. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems* (Boston, MA, USA) *(HotStorage '25)*. Association for Computing Machinery, New York, NY, USA, 45–51. doi:10.1145/3736548.3737828

[31] Jaihyuk Song. 2025. AI Revolution Driven by Memory Technology Innovation. In *2025 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 68. 26–36. doi:10.1109/ISSCC49661.2025.10904790

[32] Linus Torvalds and Linux Kernel Contributors. 2025. *Linux Kernel (6.16-rc4)*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/?h=v6.16-rc4

[33] Valkey contributors. 2025. *Valkey (8.1.2)*. https://valkey.io

[34] Valkey contributors. 2025. *Valkey Documentation – Persistence*. https://valkey.io/topics/persistence/

[35] Lukas Werling, Yussuf Khalil, Peter Maucher, Thorsten Gröninger, and Frank Bellosa. 2023. Analyzing and Improving CPU and Energy Efficiency of PM File Systems. In *Proceedings of the 1st Workshop on Disruptive Memory Systems* (Koblenz, Germany) *(DIMES '23)*. Association for Computing Machinery, New York, NY, USA, 31–37. doi:10.1145/3609308.3625265

[36] Matthew Wilcox, Ross Zwisler, and Linux Kernel Contributors. 2025. *fs/dax.c — Linux Kernel 6.16 source code*. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/dax.c?h=v6.16

[37] Wolley. 2024. *CXL Persistent Memory*. https://wolleytech.com/solutions-service/cxl-persistent-memory/

[38] Wolley. 2024. *NVMe-over-CXL*. https://wolleytech.com/solutions-service/nvme-over-cxl/

[39] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of intel optane persistent memory: a close look at its on-DIMM buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 488–505. doi:10.1145/3492321.3519556

[40] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. 2019. Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 427–439. doi:10.1145/3297858.3304077

[41] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*. USENIX Association, Santa Clara, CA,

169–182. https://www.usenix.org/conference/fast20/presentation/yang

[42] Juncheng Yang, Yazhuo Zhang, Ziyue Qiu, Yao Yue, and Rashmi Vinayak. 2023. FIFO queues are all you need for cache eviction. In *Proceedings of the 29th Symposium on Operating Systems Principles* (Koblenz, Germany) *(SOSP '23)*. Association for Computing Machinery, New York, NY, USA, 130–149. doi:10.1145/3600006.3613147

[43] 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)*. USENIX Association, Boston, MA, 601–617. https://www.usenix.org/conference/atc23/presentation/yang-shao-peng

[44] Seung Won Yoo, Joontaek Oh, Myeongin Cheon, Bonmoo Koo, Wonseb Jeong, Hyunsub Song, Hyeonho Song, Donghun Lee, and Youjip Won. 2025. DJFS : Directory-Granularity Filesystem Journaling for CMM-H SSDs. In *23rd USENIX Conference on File and Storage Technologies (FAST 25)*. USENIX Association, Santa Clara, CA, 35–51. https://www.usenix.org/conference/fast25/presentation/yoo

[45] Jianping Zeng, Shuyi Pei, Da Zhang, Yuchen Zhou, Amir Beygi, Xuebin Yao, Ramdas Kachare, Tong Zhang, Zongwang Li, Marie Nguyen, Rekha Pitchumani, Yang Soek Ki, and Changhee Jung. 2025. Performance Characterizations and Usage Guidelines of Samsung CXL Memory Module Hybrid Prototype. arXiv:2503.22017 [cs.AR] https://arxiv.org/abs/2503.22017

[46] Yekang Zhan, Haichuan Hu, Xiangrui Yang, Shaohua Wang, Qiang Cao, Hong Jiang, and Jie Yao. 2024. RomeFS: A CXL-SSD Aware File System Exploiting Synergy of Memory-Block Dual Paths. In *Proceedings of the 2024 ACM Symposium on Cloud Computing* (Redmond, WA, USA) *(SoCC '24)*. Association for Computing Machinery, New York, NY, USA, 720–736. doi:10.1145/3698038.3698539

[47] Haoyang Zhang, Yuqi Xue, Yirui Eric Zhou, Shaobo Li, and Jian Huang. 2025. SkyByte: Architecting an Efficient Memory-Semantic CXL-based SSD with OS and Hardware Co-design. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 577–593. doi:10.1109/HPCA61900.2025.00051

[48] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*. USENIX Association, Boston, MA, 207–219. https://www.usenix.org/conference/fast19/presentation/zheng

[49] Shawn Zhong, Chenhao Ye, Guanzhou Hu, Suyan Qu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Michael Swift. 2023. MadFS: Per-File Virtualization for Userspace Persistent Memory Filesystems. In *21st USENIX Conference on File and Storage Technologies (FAST 23)*. USENIX Association, Santa Clara, CA, 265–280. https://www.usenix.org/conference/fast23/presentation/zhong