Fundamental OS Design Considerations for CXL-based Hybrid SSDs

Daniel Habicht
Karlsruhe Institute of Technology
Karlsruhe, Germany

Yussuf Khalil
Karlsruhe Institute of Technology
Karlsruhe, Germany

Lukas Werling
Karlsruhe Institute of Technology
Karlsruhe, Germany

Thorsten Gröninger
Karlsruhe Institute of Technology
Karlsruhe, Germany

Frank Bellosa
Karlsruhe Institute of Technology
Karlsruhe, Germany

Abstract

The first commercial implementations of CXL-based hybrid SSDs (i.e., SSDs that are both byte- and block-addressable) are looming on the horizon. Although previous works have conducted design studies on hardware concepts as well as potential use cases, none have analyzed operating system considerations and abstractions for such storage devices. We find existing abstractions (i.e., *DAX* in Windows and Linux) to be insufficient for hybrid SSDs and propose more appropriate resource management techniques and interfaces in this work. In our evaluation we demonstrate improved throughput by up to 4.1× for applications with strong persistence requirements using the in-memory key-value store *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, Page Cache

ACM Reference Format:
Daniel Habicht, Yussuf Khalil, Lukas Werling, Thorsten Gröninger, and Frank Bellosa. 2024. Fundamental OS Design Considerations for CXL-based Hybrid SSDs. In *2nd Workshop on Disruptive Memory Systems (DIMES '24), November 3, 2024, Austin, TX, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3698783.3699380

1 Introduction

In-memory databases such as *Valkey* [3] (a fork of the more widely known *Redis* [2]) allow configuring persistence by defining a policy for how often `fsync()` shall be called to synchronize the database's *append-only file* (AOF) from volatile memory to non-volatile storage. In *Valkey*, the AOF acts as a redo log and contains all write operations since the last full *base file* rewrite (which happens whenever the AOF grows too large). Generally, smaller synchronization intervals result in less write transaction throughput as shown in Fig. 1. As can be seen in the plot, synchronizing once per second (*everysec*) does not show significant performance impact over using the operating system's implicit periodic synchronization mechanism (*never*). However, the *always* policy, which forces every AOF update to be written back immediately, does reduce throughput by 76.5 % on write-only workloads.

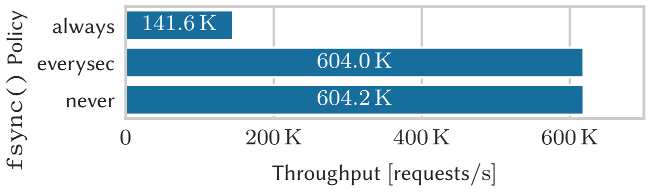

Figure 1. *Valkey* write throughput for different `fsync()` policies (pipeline length = 10). Strong persistence (*always*) reduces throughput by 76.5 % over implicit persistence (*never*).

The novel *Compute Express Link* (CXL) standard [17] has sparked interest among database researchers [28, 29]. CXL promises cache-coherency and low-latency data transmission for accelerator hardware and memory expanders [18]. Starting with CXL 2.0, the notion of persistent memory is introduced together with a flush-on-fail mechanism known as *Global Persistent Flush* (GPF) [17] that enables persistent CPU caches, similar to Intel's *eADR* feature on past *Optane* platforms [21]. Among the upcoming CXL-based persistent memory implementations are hybrid SSDs that offer byte-granular access via CXL in addition to NVMe. Commercial implementations have been announced by Samsung under the name *CXL Memory Module – Hybrid* (CMM-H) [34] as well as Wolley, dubbed *NVMe-over-CXL* [35]. However, at the time of writing, none are publicly available yet.

Although several works have already explored the viability of such hybrid SSD designs as well as potential use cases [7, 11, 24, 42], the authors focused on hardware design aspects and did not put deep thought into implications for operating system design as well as the necessary abstractions.

In this work, we review existing OS abstractions for persistent memory and explain how they are inadequate for hybrid SSDs. Based on these insights, we propose modifications and optimizations for the traditional *POSIX* APIs. Our design increases the performance of *Valkey*'s *always* synchronization policy by up to 4.1×. Our contributions are as follows:

- We establish a model for the design of hybrid SSDs from an OS perspective (§ 3.1)
- We analyze the suitability of existing abstractions in the context of hybrid SSDs (§ 3.2)
- We adapt existing user space interfaces (§ 4.1)
- We propose a persistence-aware kernel page cache for hybrid SSDs (§ 4.2)
- We simulate a virtual hybrid SSD (§ 5)
- We evaluate our design for performance and energy consumption using a real-world key-value store (§ 6)

2 Background

2.1 Compute Express Link (CXL)

CXL [17] has evolved as a new interconnect standard for data center applications, offering interesting new possibilities such as rack-level memory pooling [30] or cache-coherent accelerators [14]. In the context of databases, Lerner and Alonso show that CXL expands the borders for scale-up approaches [29].

Fundamentally, CXL builds on the physical and electrical interfaces of *PCI Express* (PCIe), but offers three new protocols [17]: `CXL.io`, which essentially equals PCIe's *Transaction Layer*, `CXL.mem`, which offers access to device-attached memory, and `CXL.cache`, which exchanges cache-coherency information over the bus. Although vendor-specific interconnect standards such as *OpenCAPI*, *CCIX*, and *Gen-Z* [12, 15, 36] already supported various subsets of CXL's features, CXL is the first implementation of a cache-coherent CPU-to-device interconnect that has gained enough traction to achieve broad industry support. CXL is supported in current data center CPUs from major vendors, including Intel [32], AMD [13], and Arm [33]. Sun et al. provide an overview of the performance characteristics of real-world CXL implementations [37]. They show that CXL memory expanders can offer latencies less than 2× (ASIC-based) or 5× (FPGA-based) higher than CPU-local DDR5 memory.

2.2 Hybrid SSDs

Bae et al. first proposed making SSDs byte-addressable. With *2B-SSD* [11], they suggest adding an additional *Base Address Register* (BAR) to a conventional block-addressable PCIe-attached NVMe SSD that is used to directly address the SSD's DRAM cache via memory-mapped I/O. The primary BAR remains as management interface as in traditional SSDs. Crucially, they identified SSD-side cache management as a major issue for performance and resorted to offering a simple `ioctl()`-based interface to explicitly pin pages from user space. In turn, *2B-SSD* lacks an OS-based abstraction mechanism for automated cache and resource management.

FlatFlash is another approach proposed by Abulila et al. [7]. Their design employs a tiered memory hierarchy where pages are transparently promoted by the operating system from a byte-addressable SSD to system memory. They argue that the reduced interconnect traffic achieved by transmitting only cache lines instead of entire blocks improves SSD performance and lifetime. However, although they ensure that atomic PCIe memory operations and cache line flushes are used when dealing with addresses mapped to the SSD, they cannot guarantee persistence for promoted pages as they reside in volatile DRAM. In turn, they resort to disabling promotion for pages that require strong persistence.

Jung introduced the idea of using CXL to implement a byte-addressable SSD [24]. They argue that an inherent limitation of PCIe-based designs is the uncacheable BAR space, i.e., the SSD's flash memory contents cannot be cached in the CPU, hence it may only serve as "working memory." However, they did not employ real CXL hardware for their evaluation and instead resorted to an FPGA-based simulation using a custom *RISC-V* processor core and an *OpenExpress* [23] device. Using an algorithm benchmark, they show that a hybrid SSD design could perform well for varying grades of data locality. Their work does not consider OS-level abstractions.

Yang et al. explore the design space for CXL-enabled SSDs by collecting physical memory access traces from various workloads and running them through a custom SSD simulation tool [42]. Similar to other works, they focus on hardware design aspects such as prefetching algorithms. They show that, depending on the workload, a hybrid SSD could serve between 68 % to 91 % of memory accesses with sub-microsecond latency.

Samsung has announced the first commercial implementation of CXL-based hybrid SSDs, known as *CMM-H* [34]. CMM-H offers a *Persistent Memory* (PM) mode as well as a *Tiered Memory* (TM) mode. In PM mode, the SSD only exposes a `CXL.mem` interface, whereas TM mode additionally supports NVMe via `CXL.io`. CXL's *Global Persistent Flush* (GPF) feature for automated cache flushes on power loss is only supported in PM mode.

2.3 Direct Access (DAX)

Sparked by Intel's (now obsolete) *Optane DCPMM* [20] memory technology, several operating systems (e.g., Linux [26] and Windows [31]) have introduced *Direct Access* (DAX) interfaces. Linux offers two DAX modes, *devdax* and *fsdax*. With *devdax*, applications can map portions of the persistent memory directly into their virtual address space. A file system on top of an *fsdax* device allows `mmap()`-ing files into processes while bypassing the operating system's volatile page cache. *fsdax* is supported by several file systems including *ext4*, but requires a per-inode flag to be set on files that should employ DAX-based mappings.

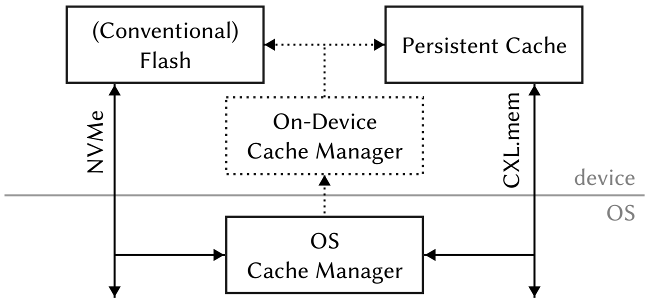

Figure 2. Overview of the hybrid SSD model we base our design on. The OS cache manager can initiate direct transfers with an optional on-device cache manager.

3 Problem Analysis

We have now seen several proposals for hybrid SSDs. Since such devices are not yet commercially available, we now propose our own model for hybrid SSDs. We base our model on available information on future real-world hardware such as CMM-H [34] as well as requirements for OS control. To motivate our design, we then discuss why existing Linux DAX support is insufficient for this model.

3.1 Hybrid SSD Model

Our hybrid SSD model consists of three main components, pictured in Fig. 2: a persistent cache, a conventional Flash-based backing storage, and a cache management component. As conventional Flash-based storage does not provide byte-granular media access, the device requires a byte-addressable cache for implementing the load/store interface. Since the load/store interface provides synchronous media access, the cache's access latency must be low, i.e., in the order of few hundreds nanoseconds, in order to avoid stall cycles due to slow media accesses. This, however, requires a memory technology that is more expensive than conventional Flash (e.g., DRAM). Consequently, our model assumes that the cache only offers a small fraction of the storage's capacity, i.e., few GiB of cache per TiB of backing storage. Further, we assume the availability of a flush-on-fail mechanism, like CXL GPF, to write all modifications in volatile CPU and device caches to the SSD's non-volatile backing storage. With this flush-on-fail mechanism, we can ensure persistence of the on-device cache even when it uses volatile DRAM.

For our approach, we assume a minimal device that only offers an interface for hardware-assisted data movement between cache and storage and leaves cache management to the operating system. This includes the mapping between cache and storage, the allocation of cache space, and a paging mechanism for providing load/store access on the entire storage capacity. The device exposes the cache as device-attached memory to the host.

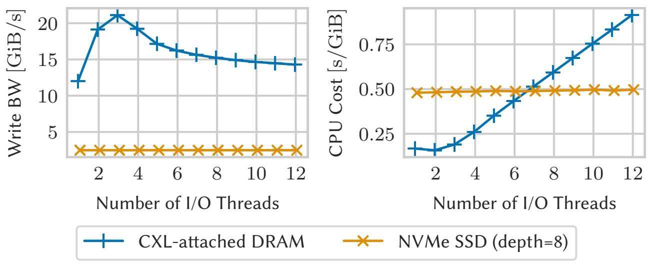

Figure 3. Random write throughput and CPU efficiency for CXL-attached DRAM and a conventional NVMe SSD.

3.2 Hybrid SSD with Linux DAX

When we apply our hybrid SSD model to Linux's existing DAX support, we notice two problems. First, Linux assumes non-blocking load/store access on the entire storage capacity at all times [26]. The hybrid SSD, however, can only provide this access through the much smaller cache. Consequently, when an application accesses an uncached file range through memory-mapped I/O, the resulting memory access stalls the CPU for the entire duration of the much slower access to the backing storage. There is no mechanism in place to defer the access until the required data is present in the cache. As a result, the CPU efficiency of I/O drops.

The second problem concerns control over DAX mappings. As of now, Linux uses a per-inode DAX flag to determine whether I/O goes through the volatile page cache or uses direct access to storage [26]. Given that Linux expects that the entire device is byte-addressable at all times, there is little use in supporting more fine-granular control over DAX even when only a small subset of a file requires DAX for strong persistence guarantees (e.g., consistency of metadata). If we apply the same approach to our hybrid SSD model, parts of a file with strong persistence requirements will have to compete with other, non-critical parts for cache capacity, resulting in a higher latency for critical writes. Regarding hybrid SSDs, we consider this to be the most severe limitation of Linux's existing DAX support.

Cache capacity is not the only resource that operating systems need to consider. When saturating the bandwidth of the load/store interface, the access latency rises. In turn, the number of stall cycles increases and the CPU efficiency (in s/GiB as defined by Werling et al. [40]) of I/O drops. Figure 3 shows this effect for a random write *fio* [10] (version 3.37) workload on CXL-attached DRAM. Unlike the CXL-attached DRAM, the NVMe SSD's block interface is not prone to decreased CPU efficiency with I/O-heavy workloads.

Windows suffers from similar limitations. For example, DAX control is limited to entire NTFS volumes [38]. To support hybrid SSDs efficiently, storage abstractions require *(1)* fine-granular control over where to use the load/store interface and the block interface, and *(2)* strategies for mitigating the CPU overhead of a cache miss on the load/store interface.

4 Approach

Based on our problem analysis in § 3, we now propose operating system abstractions for storage I/O that unfold the full potential of future hybrid SSDs. This includes a new user space API for DAX (§ 4.1) that enables fine-granular control over persistent cache usage as well as a new take on the on-device cache management that uses the page cache (§ 4.2). Figure 4 gives an overview of our design.

4.1 User Space API

To give applications fine-granular control over DAX use, we introduce a new `mmap()` flag (`MAP_DAX`) for requesting DAX mappings explicitly. This flag guarantees direct access to storage, meaning that writes on a mapping established with `MAP_DAX` are not buffered in volatile memory. While file ranges covered by DAX mappings are required to use the load/store interface of a hybrid SSD, all other mappings may be backed by volatile system memory.

By default, all DAX users contend for the same on-device cache capacity. In order to provide better performance isolation between tasks, we propose `mlock()` for pinning pages into the hybrid SSD's cache. For this, we introduce a new resource limit that allows to configure the maximum amount of DAX pages pinned on a per-task basis. Due to the differences in the device model assumed by Linux DAX and our hybrid SSD model, pinning DAX pages was previously unnecessary which resulted in `mlock()` skipping those pages.

4.2 Persistence-Aware Page Cache

For seamlessly integrating emerging hybrid SSDs into the I/O stack of modern operating systems, we propose a *persistence-aware page cache*. The idea of this design is to insert pages from the hybrid device's cache, so called *DAX pages*, into the page cache to provide direct access to storage. In addition, certain operations can leverage strong persistence guarantees of DAX pages (e.g., `fsync()`). With this approach, we put the operating system in control of the device's cache management. Apart from hardware-assisted data movement between cache and backing storage, we assume that a hybrid SSD exposes its on-device cache as device-attached memory.

Most of storage I/O goes through the page cache including memory-mapped file I/O and regular POSIX I/O calls like `read()` and `write()`. To provide a coherent view on the hybrid SSD irrespective of the storage interface used, we can leverage the fact that both direct access and buffered I/O go through the persistence-aware page cache. For DAX pages, direct I/O can be served from the page cache. While this might seem counter-intuitive, it does not contradict the definition of direct I/O, namely that the I/O request is served between the user buffer and the storage device [1], which includes the on-device cache.

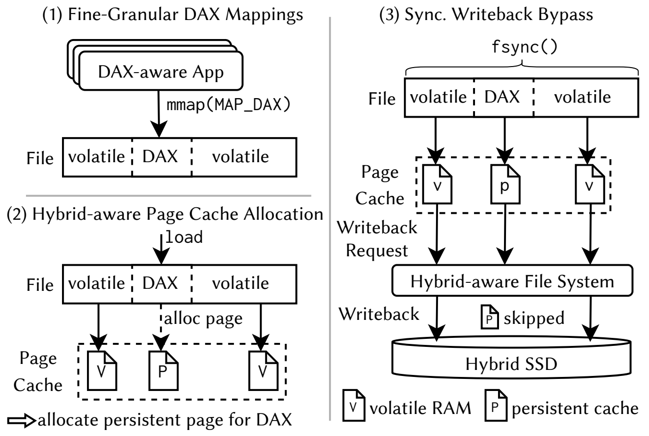

Figure 4. Overview of our design.

Another benefit of letting the operating system manage the on-device cache mapping through the page cache is that the operating system can reflect the cache mapping state in its page tables. This enables the operating system to defer load/store accesses that miss the cache until storage contents are fetched from the backing storage. For DAX-mapped file contents not present in the cache, the present bit in page table entries (PTEs) is cleared. When applications try to access such a page, the page fault handler uses the hardware-assisted data movement to bring in file contents from the backing storage. In the meantime, the operating system can schedule a different task.

To bring persistence awareness to the page cache, our design adds two components. The first is the page cache's DAX-aware page allocation. To determine the type of page the page cache needs to allocate, it checks if there is any overlapping DAX mapping for this file range. This DAX mapping is not required to belong to the task that triggered the allocation, but can originate from any task with a memory mapping on the associated file. If such an overlapping mapping exists, the page cache allocates a clean page in the hybrid device's cache, fetches the storage contents from the backing storage, and inserts it into the page cache. Otherwise, volatile system memory is allocated and file contents are obtained through the block interface. The page cache stores the mapping between cache and backing storage analogous to the mapping between buffered file contents and the backing block device.

When regular mappings and DAX mappings overlap, the DAX mapping takes precedence for determining the page allocation type due to the stronger persistence requirements of DAX mappings. When establishing a new DAX mapping, file contents buffered in volatile system memory for the mapped file range would violate the invariant of the persistence-aware page cache. To restore the invariant, volatile page cache entries must be either migrated to DAX pages (*DAX upgrade*) or evicted from the page cache. If there is free cache capacity, those page cache entries can be migrated with a fast on-CPU `memcpy()`. Otherwise, we drop them from the page cache. This, however, might incur slow I/O for writing dirty pages back to permanent storage.

The second component of our approach are optimizations to I/O calls that can benefit from persistence guarantees of DAX pages. Synchronous writeback, like calls of `fsync()`, block the calling task until file data is guaranteed to have reached the storage device. As `fsync()` becomes a performance bottleneck for applications with strong persistence requirements (Fig. 3), we use the persistence guarantee of DAX mappings to increase their performance. During synchronous writeback, DAX pages can be skipped without breaking `fsync()` semantics. Only during asynchronous writeback or when being evicted from the page cache, file contents from DAX pages are synced with the backing storage.

5 Implementation

We implement the persistence-aware page cache in Linux 6.6.0 [39]. Our design requires explicit support by file systems as parts of the writeback code path reside in file system code. Supporting the `MAP_DAX` flag and our writeback optimizations in a particular file system requires modifications of less than 150 LoC. Currently, we support *ext2* and *ext4*. In total, our modifications including memory management code and file system changes account for approximately 1500 LoC. Since there are no commercial hybrid SSDs available for evaluating our prototype, we emulate a hybrid SSD using a commodity NVMe SSD and a CXL memory expander based on Altera's *Agilex 7 I-Series* FPGA [9]. In place of an on-device cache manager, our prototype uses the host's storage stack for moving storage pages between the cache (CXL-attached DRAM) and the backing storage (NVMe SSD). This emulated device still provides a good performance model for workloads that are dominated by on-device cache hits. The CXL-attached DRAM is exposed to the host system as a memory-only NUMA node. Thus, we can use Linux's page allocation and reclaim mechanisms.

In order to track DAX mappings on files, we add a per-file interval tree [27] of DAX *virtual memory areas* (VMAs) alongside the interval tree of all VMAs already used by Linux. This enables the efficient lookup of DAX mappings without having to scan all VMAs of a cacheable object. When allocating memory, the page cache checks if the underlying device is a hybrid SSD, i.e., it has a reference on an object describing a hybrid SSD. This object includes the NUMA node information required for allocating cache pages. If the underlying device is a hybrid SDD and a DAX mapping overlaps the new page cache entry, the page cache allocates a cache page and marks it as persistent using a new page flag. Otherwise, volatile system memory is allocated using the existing allocation logic.

On the generic writeback code path, we add a flag indicating that file systems should skip DAX pages during writeback. This flag is set for all types of synchronous writeback, e.g., `fsync()` and `fdatasync()`, `sync_file_range()`, or `msync()` with `MS_SYNC`. Asynchronous writeback, however, is not affected. File systems supporting hybrid SSDs leave DAX pages marked dirty and do not tag them for writeback, thus reducing the latency of synchronous writeback. Currently, our file system support limits lightweight `fsync()` to data pages. Metadata and journal blocks do not make use of the on-device cache.

6 Evaluation

To showcase the potential of hybrid SSDs for workloads with strong persistence requirements, we evaluate our software prototype using the key-value store *Valkey* [3] (version 7.2.5) with AOF persistence [5]. As discussed in § 1, the *always* AOF `fsync()` policy drastically impacts performance, reducing the usefulness of *Valkey*'s strongest persistence mode. In order to mitigate this overhead, we propose to employ a DAX mapping on the AOF that enables lightweight `fsync()` without weakening any durability guarantees.

Since *Valkey*'s AOF implementation uses appending writes (i.e., `O_APPEND` [1]), we implement a *mmap* AOF backend that uses memory-mapped I/O for writing to the AOF. When running out of allocated space in the AOF, the *mmap* backend resizes the AOF using `fallocate()` and pre-fetches newly allocated blocks into the page cache. AOF resizing is done in a background worker. Thus, *Valkey*'s main thread handling command processing is not slowed down. To avoid page cache pollution with already written AOF contents, the *mmap* backend explicitly drops previously written parts from the page cache during resize. The AOF's page cache footprint never exceeds 40 MiB.

To use the load/store interface of hybrid SSDs, our *mmap* AOF backend can optionally use the `MAP_DAX` `mmap()` flag for the AOF mapping. With this option, the writeback of the on-device cache is deferred to the eviction of AOF contents from the page cache during the background AOF resize.

6.1 Evaluation Setup and Methodology

For our emulated hybrid SSD, we use a *Samsung 970 Pro* 1 TB NVMe SSD as the backing permanent storage and a CXL memory expander based on an *Altera Agilex 7 I-Series* FPGA [9]. The CXL memory expander is equipped with 16 GiB DDR4 @ 3200 MT/s. On our evaluation setup featuring an *Intel Xeon Silver 4416+* with 128 GiB DDR5 @ 4800 MT/s, we disable *Turbo Boost* and *Hyper Threading* in an effort to reduce variance in our measurements. The memory latency measured with Intel MLC [22] is about 3.2× higher for the CXL-attached DRAM (~358 ns) than for CPU-local DRAM (~111 ns). We use the same NVMe SSD as a baseline for our evaluation.

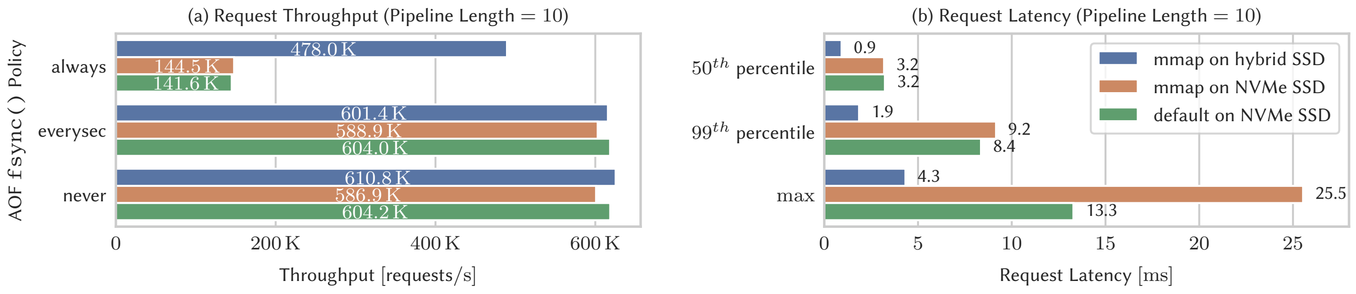

Figure 5. a. SET throughput for all AOF `fsync()` policies using the *mmap* backend (with and without hybrid SSD) and *Valkey*'s default AOF implementation. **b.** Latency percentiles for *mmap* backend (with and without hybrid SSD) and the default AOF implementation. The *mmap* backend suffers from high worst-case latencies regardless of the storage device used.

We argue that our emulated hybrid SSD provides a realistic performance outlook for future hybrid SSD products in the evaluated workload because the main difference between our emulated device and future hybrid SSD hardware, namely the data movement between Flash storage and the on-device cache, is not present on the hot path of *Valkey*'s AOF persistence. Instead, data movement between storage and cache is offloaded to the background worker that is only invoked during AOF resize operations.

Before each test run, we format the storage device under test with a clean *ext4* file system that has journaling disabled.

For measuring the CPU and energy efficiency, we adopt the efficiency metrics for analyzing PM file systems proposed by Werling et al. [40]. This CPU efficiency is quantified by dividing the sum of CPU time each core was active[1] by the amount of work done (i.e., the number of requests processed). Analogously, the energy efficiency is quantified by the consumed energy divided by the work done. We obtain the amount of energy consumed by measuring the average power at the wall plug. Similar to Werling et al., we measure the idle CPU load and idle power consumption of our evaluation system before running our benchmark and subtract this base load from our measurements to isolate the evaluated workload from the rest of the system.

As the CPU efficiency metric considers the load on the entire system, thus also including CPU overhead of the data movement between storage and cache, we expect real-world hybrid SSDs to improve CPU efficiency over our emulated device. The energy efficiency measured when using the emulated device suffers from similar limitations.

6.2 Valkey AOF Performance

In the following, we present throughput and latency statistics from *valkey-benchmark*'s write-only SET benchmark [4]. In our testing, other write-only workloads showed similar behavior to SET.

As shown in Fig. 5 (a), our *mmap* AOF backend offers performance similar to the default AOF implementation when using conventional storage. With the hybrid SSD, the *mmap* backend achieves a 237.6 % higher throughput for the costly *always* AOF `fsync()` policy. In the case of *everysec* and *never*, *Valkey*'s performance is hardly affected by the use of the hybrid SSD. This is expected as these policies issue few to none `fsync()` calls. We do not see a negative effect from the increased memory latency of our CXL-attached DRAM.

Fig. 5 (b) shows percentiles for the request latency with the *always* `fsync()` policy. Compared to the default AOF backend, the median request latency of the *mmap* backend with our hybrid SSD is reduced by 71.5 %, and the 99th percentile even more by 77.8 %. The worst-case latency, however, does not follow this trend, and is only reduced by 67.4 %. This behavior is not exclusive to the hybrid SSD, but visible for the *mmap* backend in general.

We suspect that the comparatively high worst-case latencies of the *mmap* backend are caused by `fsync()` calls syncing outstanding file metadata following AOF writes. For the most cases, calls to `fsync()` after an AOF write only need to persist data pages. The AOF metadata only changes during a resize operation. If a call to `fsync()` in the main thread occurs during an AOF resize in the background worker, the `fsync()` writes the AOF metadata even though the main thread is only concerned with syncing AOF writes that do not require the updated metadata for persistence. If our theory is correct, we expect to solve this problem by using a synchronous writeback interface for AOF writes, in order to only sync specific ranges of the AOF (e.g., `msync()`).

Valkey can improve throughput with command pipelining [6]. When the pipeline length increases, the number of AOF writes per request decreases because *Valkey* coalesces commands from the pipeline into a single AOF write [5]. Consequently, the number of `fsync()` calls under the *always* policy is anti-proportional to the pipeline length.

Figure 6 (a) shows the impact of the pipeline length on the throughput. When each write command is synced immediately, the hybrid SSD achieves a throughput of up to 4.1×. Due to the decreased `fsync()`-to-write ratio for long pipelines, the lead of the hybrid SSD over conventional I/O shrinks (Fig. 6 (b)). Similar to the throughput, the *mmap*

[1] $\sum t_{sys} + t_{user} + t_{irq} + t_{softirq}$ with t_x as reported in `/proc/stat`

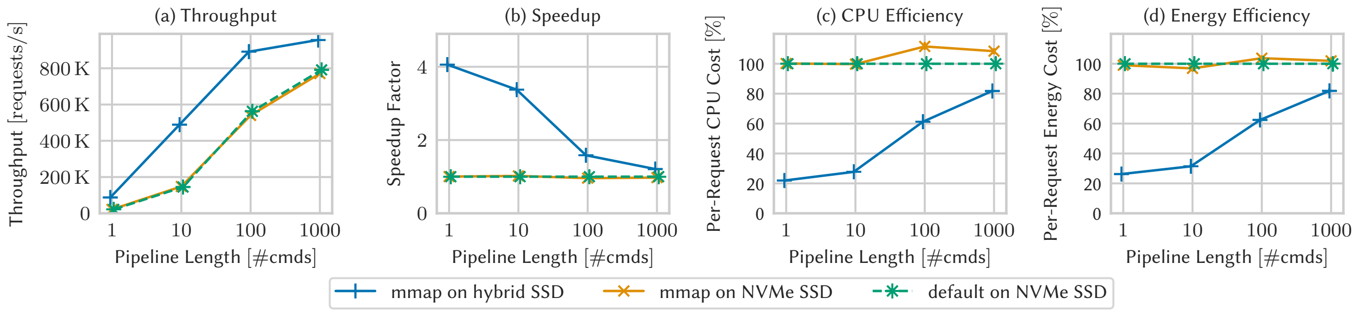

Figure 6. Throughput, speedup factor, CPU efficiency, and energy efficiency as a function of the pipeline length for the *always* `fsync()` policy. The impact of lightweight `fsync()` decreases with an increasing pipeline length.

backend with the hybrid SSD also improves the CPU and energy efficiency of request processing. Figure 6 (c) shows that our hybrid SSD helps to reduce the per-request CPU overhead by up to 78 %. The per-request energy consumption (Fig. 6 (d)) is reduced by up to 74 %.

6.3 Discussion

One problem that we observed during `fsync()` micro benchmarks on DAX mappings concerns the dirty state tracking of Linux. Since our current implementation leaves DAX pages marked dirty during synchronous writeback, the number of dirty pages continuously grows until the next asynchronous writeback. During calls of `fsync()`, file systems scan all dirty pages of a file, but when the dirty set of this file grows too large, the CPU overhead of frequently scanning all dirty pages gets impractical. In order to avoid this problem, we propose to introduce a third dirty state, namely the *out-of-sync* state, that describes a page whose content is guaranteed to persist but is not synced with the backing storage. During lightweight `fsnyc()` calls, we skip the writeback of dirty DAX pages but mark them out-of-sync, thus excluding them from future synchronous writeback. As our AOF *mmap* backend maintains a small page cache footprint for DAX-mapped contents, it does not suffer from the limitations of the currently implemented dirty state tracking.

Although real hardware is not yet available, we believe our hybrid SSD model closely reflects what can be expected from future devices. Despite this limitation, our evaluation shows that the OS-level abstractions we devised can achieve substantial performance improvements when it comes to in-memory database synchronization. We argue that our approach makes the trade-off between performance and strong persistence guarantees less painful. However, we acknowledge that future hardware may deviate from our model. Re-evaluating on real-world hardware will show whether our design works in practice.

As various industry stakeholders push for bringing CXL into consumer hardware [8, 16], we argue that end-user software with an internal database (e.g., *SQLite* [19]) may be another potential use case for our design.

7 Future Work

While our current design provides an easy-to-use interface for DAX that requires minimal changes to code already using memory-mapped I/O, applications require modifications regardless. As seen with *Optane DCPMM*, shipping explicit DAX support in applications takes time and significant effort. Especially for consumer applications where developers lack the resources to support upcoming technologies with few initial users, this is a serious obstacle in the widespread adoption of hybrid SSDs.

In order to utilize otherwise unused on-device cache capacity, we propose *Transparent DAX Mappings* (TDMs) as a future extension to our design. With TDMs, we aim to opportunistically provide user space applications with transparent DAX access (i.e., within `read()` and `write()`) without having to explicitly request a DAX mapping, thereby bypassing the kernel and avoiding block layer protocol overhead. Although prior works like *FLEX* [41], *SplitFS* [25], or *MadFS* [43] previously explored DAX for bypassing the kernel on data operations, they assume a storage device that does not require careful resource management. Unlike prior research, we plan to build TDMs around our ideas of fine-granular DAX control and hybrid SSD-aware resource management. Further, we intend to explore techniques for dynamically detecting frequently synced file ranges and speculatively migrate them to the on-device cache.

8 Conclusion

In this work, we looked at future hybrid SSDs from an operating system's perspective. We have established a model and, building on that model, devised OS-level abstractions and interfaces that we deem more appropriate than what is offered by state-of-the-art operating systems. Our evaluation using *Valkey* has shown that our design can reduce the performance disadvantage of strong persistence with an 4.1× increase in throughput, thereby reducing the trade-off between persistence and performance. However, further evaluation using real hardware in the future is necessary.

References

[1] 2024. *open(2) – Linux manual page*. https://man7.org/linux/man-pages/man2/open.2.html

[2] 2024. *Redis - The Real-time Data Platform*. https://redis.io/

[3] 2024. *Valkey: an open source, in-memory data store*. https://valkey.io/

[4] 2024. *Valkey Benchmark*. https://valkey.io/docs/topics/benchmark/

[5] 2024. *Valkey Persistence*. https://valkey.io/docs/topics/persistence/

[6] 2024. *Valkey Pipelining*. https://valkey.io/docs/topics/pipelining/

[7] 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. https://doi.org/10.1145/3297858.3304061

[8] Paul Alcorn. 2022. *AMD Working to Bring CXL Memory Tech to Future Consumer CPUs*. https://www.tomshardware.com/news/amd-working-to-bring-cxl-technology-to-consumer-cpus

[9] Altera Corporation. 2023. *Agilex™ 7 FPGA I-Series Development Kit (2x R-Tile and 1x F-Tile)*. https://www.intel.com/content/www/us/en/products/details/fpga/development-kits/agilex/agi027.html

[10] Jens Axboe. 2022. *Flexible I/O Tester*. https://github.com/axboe/fio

[11] 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)*. IEEE, Los Angeles, CA, 425–438. https://doi.org/10.1109/ISCA.2018.00043

[12] Brad Benton. 2017. CCIX, Gen-Z, OpenCAPI: Overview & Comparison. In *OpenFabrics Alliance (OFA) Workshop*, Vol. 13. OpenFabrics Alliance. https://openfabrics.org/images/eventpresos/2017presentations/213_CCIXGen-Z_BBenton.pdf

[13] Ravi Bhargava and Kai Troester. 2024. AMD Next-Generation "Zen 4" Core and 4th Gen AMD EPYC Server CPUs. *IEEE Micro* 44, 3 (2024), 8–17. https://doi.org/10.1109/MM.2024.3375070

[14] Anthony M Cabrera, Aaron R Young, and Jeffrey S Vetter. 2022. Design and analysis of CXL performance models for tightly-coupled heterogeneous computing. In *Proceedings of the 1st International Workshop on Extreme Heterogeneity Solutions* (Seoul, Republic of Korea) *(ExHET '22)*. Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3529336.3530817

[15] Greg Casey and Kurtis Bowman. 2017. Gen-Z - An Overview and Use Cases. In *OpenFabrics Alliance (OFA) Workshop*, Vol. 13. OpenFabrics Alliance. https://www.openfabrics.org/images/eventpresos/2017presentations/214_Gen-Z_GCasey.pdf

[16] San Chang. 2023. *Making the Case for CXL Native Memory*. https://conferenceconcepts.app.box.com/s/k5awbwpyxteq9u4r8a3ey75yfx16aue1

[17] CXL Consortium. 2023. *Compute Express Link Specification Revision 3.1*.

[18] 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 (jul 2024), 37 pages. https://doi.org/10.1145/3669900

[19] D. Richard Hipp. 2024. *SQLite*. https://www.sqlite.org

[20] Intel Corporation. 2019. *Intel® Optane™ DC Persistent Memory Product Brief*. https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf

[21] Intel Corporation. 2021. *eADR: New Opportunities for Persistent Memory Applications*. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

[22] Intel Corporation. 2024. *Intel® Memory Latency Checker*. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html

[23] Myoungsoo Jung. 2020. OpenExpress: Fully Hardware Automated Open Research Framework for Future Fast NVMe Devices. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 649–656. https://www.usenix.org/conference/atc20/presentation/jung

[24] 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. https://doi.org/10.1145/3538643.3539745

[25] 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. https://doi.org/10.1145/3341301.3359631

[26] Linux kernel contributors. 2023. *Direct Access for Files – The Linux Kernel Documentation*. https://docs.kernel.org/6.6/filesystems/dax.html

[27] Rob Landley. 2023. *Red-Black Trees (Rbtree) in Linux – The Linux Kernel Documentation*. https://docs.kernel.org/6.6/core-api/rbtree.html

[28] Sangjin Lee, Alberto Lerner, Philippe Bonnet, and Philippe Cudré-Mauroux. 2024. Database Kernels: Seamless Integration of Database Systems and Fast Storage via CXL. In *14th Conference on Innovative Data Systems Research (CIDR'24)*. https://www.cidrdb.org/cidr2024/papers/p43-lee.pdf

[29] Alberto Lerner and Gustavo Alonso. 2024. CXL and the Return of Scale-Up Database Engines. *Proc. VLDB Endow.* 17, 10 (aug 2024), 2568–2575. https://doi.org/10.14778/3675034.3675047

[30] Huaicheng Li, Daniel S. Berger, Lisa Hsu, Daniel Ernst, Pantea Zardoshti, Stanko Novakovic, Monish Shah, Samir Rajadnya, Scott Lee, Ishwar Agarwal, Mark D. Hill, Marcus Fontoura, and Ricardo Bianchini. 2023. 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, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 574–587. https://doi.org/10.1145/3575693.3578835

[31] Microsoft Corporation. 2022. *Understand Direct Access (DAX) and create DAX volumes with persistent memory devices*. https://learn.microsoft.com/en-us/windows-server/storage/storage-spaces/persistent-memory-direct-access

[32] Nevine Nassif, Ashley O. Munch, Carleton L. Molnar, Gerald Pasdast, Sitaraman V. Lyer, Zibing Yang, Oscar Mendoza, Mark Huddart, Srikrishnan Venkataraman, Sireesha Kandula, Rafi Marom, Alexandra M. Kern, Bill Bowhill, David R. Mulvihill, Srikanth Nimmagadda, Varma Kalidindi, Jonathan Krause, Mohammad M. Haq, Roopali Sharma, and Kevin Duda. 2022. Sapphire Rapids: The Next-Generation Intel Xeon Scalable Processor. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 44–46. https://doi.org/10.1109/ISSCC42614.2022.9731107

[33] Andrea Pellegrini. 2021. Arm Neoverse N2: Arm's 2nd generation high performance infrastructure CPUs and system IPs. In *2021 IEEE Hot Chips 33 Symposium (HCS)*. IEEE, 1–27. https://doi.org/10.1109/HCS52781.2021.9567483

[34] Rekha Pitchumani. 2023. *CMM-H (CXL Memory Module – Hybrid): Samsung's CXL-based SSD for the Memory-centric Computing Era*. Samsung. https://semiconductor.samsung.com/us/news-events/tech-blog/webinar-memory-semantic-ssd/

[35] Bernard Shung, San Chang, and Terry Cheng. 2023. *NVMe over CXL (NVMe-oC): An Ultimate Optimization of Host-Device Data Movement.* https://sc23.supercomputing.org/proceedings/exhibitor_forum/exhibitor_forum_files/exforum118s2-file2.pdf

[36] J. Stuecheli, W. J. Starke, J. D. Irish, L. B. Arimilli, D. Dreps, B. Blaner, C. Wollbrink, and B. Allison. 2018. IBM POWER9 opens up a new era of acceleration enablement: OpenCAPI. *IBM Journal of Research and Development* 62, 4/5 (2018), 8:1–8:8. https://doi.org/10.1147/JRD.2018.2856978

[37] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Chihun Song, Jinghan Huang, Houxiang Ji, Siddharth Agarwal, Jiaqi Lou, Ipoom Jeong, Ren Wang, Jung Ho Ahn, Tianyin Xu, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture* (Toronto, ON, Canada) *(MICRO ’23)*. Association for Computing Machinery, New York, NY, USA, 105–121. https://doi.org/10.1145/3613424.3614256

[38] Tom Talpey. 2017. Persistent Memory in Windows Server 2016. In *Persistent Memory Summit 2017*. SNIA.org, SNIA 5201 Great America Parkway Suite 320 Santa Clara, CA 95054, 23 pages. https://www.snia.org/sites/default/files/PM-Summit/2017/presentations/Tom_Talpey_Persistent_Memory_in_Windows_Server_2016.pdf

[39] Linus Torvalds. 2023. *Linux Kernel.* https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?h=v6.6

[40] 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 (DIMES ’23)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/3609308.3625265

[41] 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. https://doi.org/10.1145/3297858.3304077

[42] 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

[43] 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