Karlsruhe Institute of Technology

Rethinking Storage I/O for Hybrid NVMe and DAX Block Devices

Master's Thesis
submitted by

cand. inform. Daniel Habicht

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Yussuf Khalil, M.Sc.
	Lukas Werling, M.Sc.

November 20, 2023 – July 19, 2024

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any source or auxiliary means other than these referenced. This thesis was carried out in accordance with the Rules for Safeguarding Good Scientific Practice at Karlsruhe Institute of Technology (KIT).

Karlsruhe, July 19, 2024

Abstract

With recent advances in cache coherent interconnects like Compute Express Link (CXL), providing low-latency load/store semantics for device-attached memory becomes feasible in practice. Next to the expansion of main memory, this also opens up the possibility for a new type of storage product that has been proposed in the past: hybrid storage devices combine a byte-addressable persistent cache accessed through load/store semantics, with conventional, cost-efficient Flash for bulk storage. Such devices seem especially promising for providing memory-like storage performance on workloads with strong persistence requirements and high locality. Storage abstractions in operating systems, however, have been tailored to asynchronous block interfaces that were used for conventional storage devices over decades. While Direct-Access (DAX) abstractions found in operating systems like Linux promise to support byte-addressable storage (e.g., Optane persistent memory), these abstractions are insufficient for hybrid storage because they assume a different, incompatible device model.

In this thesis, we rethink hybrid storage support from the ground up. We propose a persistence-aware page cache for seamlessly integrating hybrid storage devices into the modern I/O stack. Our design aims to provide direct access to storage and optimize performance-critical operations like synchronous writeback (i.e., `fsync`). With the limitations of the small-capacity cache in mind, we design a new user space API centered around fine-granular control over mappings of cache memory. While out-of-scope for this thesis, our design provides the foundation for building towards transparent use of hybrid storage devices. We implement our approach in the Linux kernel.

Our evaluation on micro benchmarks shows up to $37\times$ higher throughput for synchronous writeback. We are able to translate these improvements into up to $4.1\times$ higher throughput on the key-value datastore Valkey while reducing the per-request CPU and energy overhead by $78\,\%$ and $74\,\%$.

Contents

Chapter 1

Introduction

For decades, the gap in performance and latency between main memory and persistent storage shaped how Operating Systems (OSs) and applications approach storage I/O. As even modern low-latency Non-volatile Memory Express (NVMe) Solid State Drives (SSDs) have access latencies of over 10 µs [31], they are at least two orders of magnitude slower than typical main memory at access latencies in the order of 100 ns [14]. Because of this, the interaction between OS and storage device centered around asynchronous block interfaces. Such interfaces use interrupts for signaling completion and Direct Memory Access (DMA) for efficient data transfer [2]. The asynchronous nature of this approach enabled the OS to overlap slow media access of larger storage blocks with running other processes [2].

In recent years, byte-addressable Persistent Memory (PM) technologies became commercially available, e.g., in the form of Intel's Optane PM [22], driving access latencies down to a few hundred nanoseconds [23]; albeit at a higher price per gigabyte than conventional SSDs [16]. Contrary to Flash-based persistent storage that uses asynchronous block I/O (e.g., NVMe [43]) to cope with high access latencies, such devices are directly attached to the CPU's memory subsystem, providing synchronous media access through `load`/`store` instructions. Due to this change of the access paradigm, OSs introduced new storage abstractions, such as Linux's Direct Access (DAX)[32], for making use of byte-addressable, low-latency storage.

In order to get the best of both worlds, that is the low-latency access of PM attached to the CPU's memory subsystem and the low cost of Flash-based storage, hybrid devices seem promising. With 2B-SSD, Bae et al. [6] propose the to our knowledge first hybrid storage architecture that combines a `load`/`store` interface with conventional block I/O. 2B-SSD uses a small integrated persistent cache for realizing `load`/`store` access over Peripheral Component Interconnect Express (PCIe) on a large Flash-based storage [6]. For their work, the authors characterize the hybrid storage approach by combining byte-addressability with conventional block I/O [6], which provides a clear distinction to storage devices that just feature multiple memory technologies on the device. Therefore, conventional NVMe SSDs that feature DRAM caches, regardless of features like power loss protection that guarantee persistence of buffered contents [61], do not classify as hybrid storage under this description, as they lack the `load/store` interface.

Despite promising results on hybrid storage and byte-addressable SSDs [6, 1], hybrid storage devices did not gain commercial traction for several years due to technological limitations. PCIe [44], the most widely used interconnect for attaching high-end NVMe storage, suffers from one major limitation when it comes to byte-addressable storage: as highlighted by Jung [25] in their work on Compute Express Link (CXL)-based SSDs, exposing byte-addressable storage through PCIe's Base Address Registers (BARs) requires uncachable Memory-Mapped I/O (MMIO) from the CPU. As this excludes PCIe-attached storage from being present in CPU caches, it severely degrades performance for memory accesses [25].

With the industry-wide push for the cache coherent CXL [7] interconnect, the technical limitations holding hybrid storage back are a thing of the past. CXL enables memory semantics for device-attached memory with CPU-side caching [13]. Our own measurements (§ 3.1) show a load latency for CXL-attached DRAM that is approximately 3.3 times higher than CPU-local Dynamic Random Access Memory (DRAM). These measurements fall in line with measurements of CXL memory devices by Sun et al. [54].

While the growing availability of CXL-capable hardware encourages work around new hybrid storage products [48, 15, 47], current OSs do not seem to be well-prepared to deal with hybrid storage. Storage abstraction for PM like Linux's DAX subsystem build on the assumption that the `load/store` interface provides low-latency access at all times [32]. This assumption, however, does not hold for hybrid storage devices as they provide `load/store` access to the storage capacity through a much smaller cache.

In this thesis, we present an OS-centric approach for managing hybrid storage devices in modern OSs. For our work, we focus on hybrid storage featuring a persistent cache, i.e., all writes that make it to the device survive a crash. Our approach is centered around the concept of a *persistence-aware page cache* that takes over the management of the persistent cache featured on hybrid storage devices. By leveraging the persistence of the cache, our persistence-aware page cache seamlessly enables direct-access to storage on memory-mapped files, low-latency synchronous writeback, and direct I/O through the `load/store` interface. These optimizations specifically target performance-critical workloads with strong persistence requirements like databases. To cope with the limited cache capacity of hybrid storage devices, we design a new user space API that enables fine-granular control over mappings of cache memory, so-called *DAX mappings*. We implement our approach in Linux [56] version 6.6. Our current implementation limits hybrid storage support to the ext2 and ext4 file system drivers. For evaluating our approach, we assess the impact of our synchronous writeback optimizations regarding performance, CPU efficiency, and power efficiency on an emulated hybrid storage device consisting of a commodity NVMe SSD and a FPGA-based CXL memory expander. We use micro benchmarks and the key-value datastore Valkey [57] for our evaluation and show that our implementation provides up to 37 times more performance for `fsync()` while also significantly improving the CPU and energy efficiency. We can translate this improvement for synchronous writeback into up to $4.1\times$ higher throughput in Valkey when using Valkey's strongest persistence mode while reducing the per-request CPU and energy overhead by $78\,\%$ and $74\,\%$ of conventional storage under certain conditions.

The remainder of this thesis is structured as follows: First, Chapter 2 provides background on technologies and related work regarding hybrid storage devices. Then, Chapter 3 motivates the requirement for new hybrid storage abstractions, defines the device model assumed by our approach, and describes our persistence-aware page cache design as well as our new user space API. Chapter 4 describes our modifications to the Linux kernel including file systems and outlines our hybrid storage device emulation. In addition, Chapter 4 also describes modifications to applications used in our evaluation. After that, we evaluate our implementation in Chapter 5 and discuss the results of our tests. In Chapter 6, we discuss limitations of our current implementation and provide an outlook towards future work on hybrid storage devices and our persistence-aware page cache. In Chapter 7, we conclude this thesis.

Chapter 2

Background

In this chapter, we provide background related to emerging hybrid storage devices and outline related research. Section 2.1 provides a short introduction to CXL, an interconnect technology that can be used to implement the `load`/`store` interface of hybrid storage devices. In Section 2.2, we discuss direct-access (DAX) for files in Linux. Finally, Section 2.3 outlines background for hybrid storage.

2.1 Compute Express Link (CXL)

CXL is an open industry standard designed by the CXL consortium that provides a set of interconnect protocols for connecting CPUs with a wide array of peripheral devices like accelerators, memory devices, and storage devices [13]. Since the introduction of the first CXL specification in 2019, CXL has achieved industry-wide support [5], with CXL 3.1 from November 2023 being the latest revision to the CXL specification [10]. With competing cache coherent interconnects like Gen-Z, CCIX, and OpenCAPI donating their IP to further advance CXL and concentrate industry efforts [8, 11, 9], CXL turns out to be the next interconnect powering data centers.

Next to resource sharing in the context of datacenters, CXL aims to meet demand for further memory scaling and provide coherent access to memory resources [13]. In order to ease its adoption and improve backwards compatibility, CXL reuses PCIe's physical interface by dynamically multiplexing three subprotocols on top of it [13]:

Device Type	CXL.io	CXL.cache	CXL.mem
CXL Type 1	✓	✓	✗
CXL Type 2	✓	✓	✓
CXL Type 3	✓	✗	✓

Table 2.1: CXL device types and the subprotocols that they support. Based on [7].

1. **CXL.io** implements a non-coherent `load/store` interface that adopts PCIe's transaction layer [7]. This protocol provides fundamental functionality including device discovery and configuration, virtualization, and DMA [13]. Therefore, it is required by all CXL-compliant devices.

2. **CXL.cache** enables CXL devices to cache system memory of the host in a coherent fashion [13]. Similar to CPU caches on modern x86 CPUs, CXL.cache always uses 64 B cache lines [7]. The host is responsible for managing cache coherence [13].

3. **CXL.mem** provides `load/store` semantics for device-attached memory [13]. CXL.mem supports a variety of memory types, including volatile memories like DRAM or PM [7].

For supporting a wide range of applications, the CXL specification defines three device types that differ in the protocols that they support [7]. Table 2.1 outlines CXL protocol support for all three device types. While type 1 and type 2 devices have processing components that require coherent access through CXL.cache, type 3 devices can realize passive memory expanders that expose device-attached memory to the host [7]. Due to scaling and performance difficulties, Jung [25] advocates for building byte-addressable CXL storage with type 3 devices. For this thesis, we assume a hybrid storage device that does not do any computations and thus is a good candidate for a CXL type 3 device.

Another important feature for building hybrid storage devices concerns persistence of caches. Starting with CXL 2.0, Global Persistent Flush (GPF) provides a hardware-based feature that ensures that in-flight changes residing in volatile caches get flushed to a persistence domain in case of an unexpected shutdown like a power loss [7, § 9.8]. This includes CPU caches but also caches and memory buffers on CXL devices. GPF follows a two phase procedure [7, § 9.8.1]: first, devices must abstain from introducing new changes and write in-flight changes in volatile caches back to persistent memory. Then, persistent memory devices must flush their write buffers. Conceptually, GPF fulfills a similar purpose to the extended Asynchronous DRAM Refresh (eADR) feature that Intel introduced for Optane PM on third-gen Xeon Scalable Processors [20].

As technologies like GPF or eADR eliminate the need for applications to explicitly flush caches, they significantly simplify the programming model for working with persistent memory [20]. We argue that, because of this, they are crucial for enabling strong persistence guarantees on emerging hybrid storage devices. Hybrid devices like Samsung's CMM-H [48] already offer GPF support. In Section 2.3, we provide further details on hybrid storage devices such as CMM-H.

2.2 Direct Access (DAX) in Linux

Due to the large performance gap between traditional storage devices and the CPU, OSs typically employ a page cache for buffering storage contents in main memory [2]. Like many other OSs, Linux, since version 2.4, uses a fully unified page cache for `read`/`write` system calls and memory-mapped I/O alike [46, 52]. For file I/O on PM with memory semantics and performance similar to system memory (e.g., Intel Optane PM [22]), this design is obsolete. For one, the page cache introduces an unnecessary copy of file contents [32]. Secondly, strong persistence guarantees offered by PM are lost when file contents are buffered in volatile system memory.

In order to solve these problems, Linux introduced DAX for file systems [32]. When working with memory-mapped files on file systems[1] that support DAX, Linux's DAX subsystem maps the PM pages directly into user space [32]. For `read()` and `write()` calls that would otherwise go through the page cache, DAX-aware file systems also offer improvements. Here, `read()` and `write()` calls break down to a `memcpy` between the I/O buffer in user space and PM [28]. However, Kim et al. [28] show that this I/O model is not optimal for PM as the `memcpy` between the I/O buffer and storage can introduce significant overhead. Werling et al. [62] propose copy offloading for mitigating the overhead of `memcpy` in PM file systems.

To control whether file I/O uses DAX or goes through the volatile page cache, Linux uses a volatile per-inode DAX flag that is stored in the in-memory representation of the file [32]. This flag is determined by the mount option used for the underlying file system and an additional persistent DAX file attribute [32]. Depending on the mount option, the persistent flag might be ignored, i.e., either use DAX always or use it never [32].

In Section 3.3.1, we discuss shortcomings of Linux's DAX subsystem in supporting hybrid storage devices. Finally, we want to point out that Linux is not the only OS that offers DAX support. On NTFS volumes, Windows also offers DAX support that is similar to Linux's DAX subsystem [55].

[1]From the in-tree file systems in Linux 6.6, ext2, ext4, xfs, virtiofs, and erofs support DAX [32].

2.3 Hybrid Storage

Next up, we discuss three hardware approaches for hybrid storage in more detail.

2B-SSD

With 2B-SSD, Bae et al. [6] introduce the concept of a hybrid store that combines a `load`/`store` interface with a conventional block interface for accessing storage. For building a hardware prototype, the authors implement memory semantics through PCIe and use a commodity low-latency NVMe SSD for the backing storage. As NAND flash does not provide the byte-addressability required for memory semantics, they use a DRAM cache to buffer flash pages and enable fine-granular access. 2B-SSD stores the mapping between flash and DRAM cache on the device. In addition, the authors integrate back-up power on the device to write buffered pages as well as the mapping table to flash when a power loss is detected. 2B-SSD enforces consistency between both storage interfaces in hardware. The authors propose a simple ioctl-based interface for managing the cache and using the device's DMA engine for bulk reads. While all writes that reach 2B-SSD are guaranteed to persist, the authors note that writes may be buffered in write-combining caches of the CPU, or in the PCIe root complex. Therefore, 2B-SSD requires explicit flushes to guarantee persistence of writes. [6]

For this thesis, we adopt the authors' concept of a hybrid store. What differentiates our work from 2B-SSD, however, is the focus on suitable OS abstractions for hybrid storage and our OS-centric approach to managing the cache. By integrating the hybrid storage's cache into the page cache, we can avoid calling into the block layer for regular I/O calls on DAX-mapped files, like `write()` or `fsync()`. Further, our persistence-aware page cache abstracts the cache management, maintains consistency, and provides virtualization of the cache. This has the benefit that neither the device nor applications have to implement this functionality themselves and that the OS can control resource usage, i.e., cache capacity, on a per-task or per-file basis.

CXL Memory Module Hybrid (CMM-H)

Samsung's CXL Memory Module Hybrid (CMM-H) is a CXL-based SSD (type 3 device) that combines NAND flash with a DRAM cache for enabling low-latency `load`/`store` access to storage [49]. CMM-H supports two modes of operation: in persistent memory mode, CMM-H provides byte-addressable persistent memory that is accessed through CXL.mem [49]. In tiered memory mode, the backing SSD can either be accessed through CXL.io[2] or CXL.mem by going through the DRAM cache [48]. To improve the hit rate of the on-device cache, CMM-H offers an interface for providing hints to the internal cache controller [49].

[2]Based on the available information [49], we expect block-based I/O through NVMe.

At the time of writing, CMM-H is not yet commercially available. While Samsung hints at Linux support [49], we are not aware of any concrete information on how tiered memory mode slots into the I/O stack of supported OSs. Since we expect that tiered memory mode offers both I/O interfaces for accessing storage, we think that CMM-H might be a promising candidate for evaluating our approach. We plan to reevaluate our approach for CMM-H in the future.

Hybrid Memory Subsystem (HMS)

At Super Computing 2019, IBM introduced the hybrid memory subsystem (HMS), a hybrid memory expander that combines Samsung zNAND [50] for persistent storage with Magnetoresistive Random Access Memory (MRAM)[3] and DRAM [24]. Similar to Samsung's CMM-H, HMS offers `load`/`store` semantics on persistent storage through a DRAM cache, but also includes an additional SRAM cache [24]. In case of HMS, the coherent OpenCAPI [53] interconnect enables memory semantics [24]. While CMM-H offers an API for passing hints to the cache controller [49], we are not aware of a similar mechanism for HMS. Instead, the cache management of HMS seems to rely on hardware cache prefetching that works best for sequential workloads [24]. For future work, we plan to investigate whether our approach can be adapted for HMS. As HMS uses an FPGA in its design and features Arm cores running the firmware [24], there is at least a theoretical potential for customizations.

[3]This MRAM is manufactured by Everspin. We do not know the specific MRAM type.

Chapter 3

Approach

With the rising availability of CXL-capable hardware, hybrid storage devices that combine cost-effective storage technologies, like Flash, with fast and byte-addressable memory for providing synchronous direct-access to storage, seem like the logical next step in advancing storage performance. Predominant storage technologies, however, have shaped the development of OS storage abstraction. While modern OSs, like Linux [56], do support direct-access storage [32], we argue that current storage abstractions are inadequate for emerging hybrid storage devices.

In this chapter, we rethink storage I/O for emerging hybrid storage devices from the ground up. First, we characterize the synchronous and asynchronous I/O interfaces on the basis of performance, software overhead, and energy efficiency (§ 3.1). Based on the observations of this analysis, we define an abstract hybrid storage device model that serves as foundation for building hybrid storage support in OSs (§ 3.2). Next, we discuss shortcomings of Linux's storage abstractions in supporting our hybrid storage model and propose an alternative user-facing API that fits the hybrid storage approach (§ 3.3). Finally, we outline our persistence-aware page cache design for supporting hybrid storage devices alongside conventional storage (§ 3.4).

3.1 I/O Interface Characterization

The choice of the I/O interface between the storage device and the OS is a deciding factor in shaping the performance of a system. Due to the large discrepancy in performance between the CPU and traditional storage technologies, conventional storage devices feature an asynchronous block interface. An asynchronous block interface for storage typically uses interrupts for signaling completions and DMA for efficient transfer of blocks [2]. The key advantage in using such an interface is that the OS can overlap slow I/O with the execution of tasks not waiting on the completion of outstanding I/O requests [2]. Figure 3.1 shows the interaction between the storage device and the CPU. As polling a

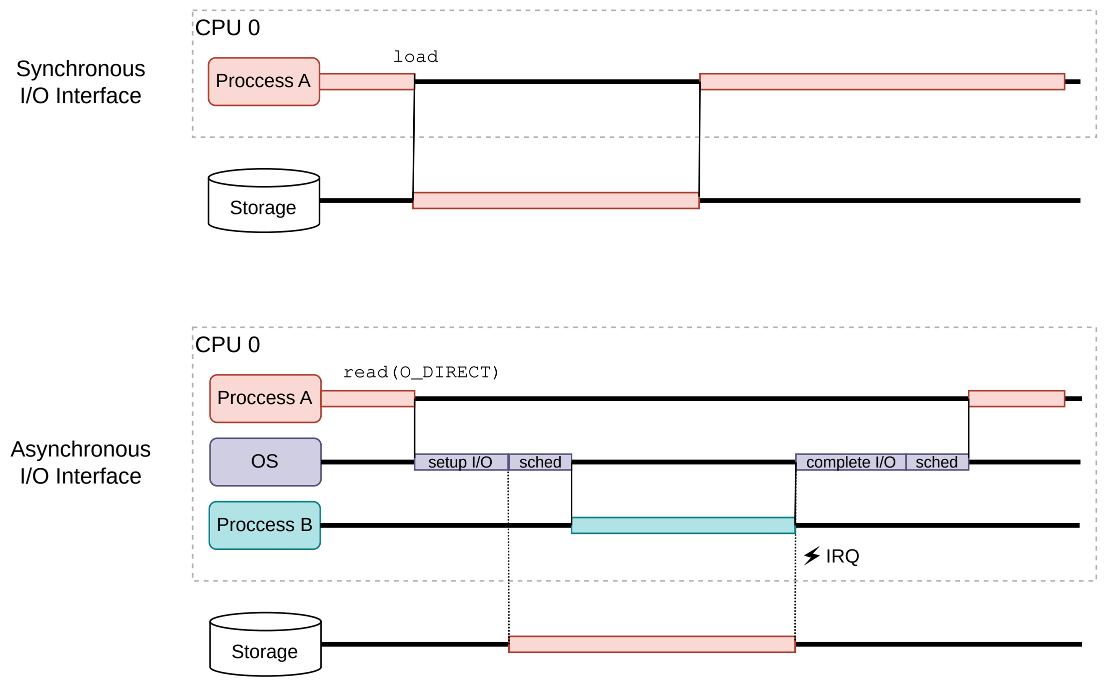

Figure 3.1: Simplified interaction between user space, kernel, and storage for the synchronous (top) and the asynchronous (bottom) I/O interface. In the synchronous case, the application uses memory-mapped I/O (DAX). The asynchronous case shows block-based direct I/O. Synchronous/asynchronous refers to the interaction between the CPU and the storage device. The timing is not representative of real-world I/O.

block device for completion also presents a synchronous interface, the characterization of polled I/O vs. interrupt-driven I/O, as described in [2, §36], is analog to the discussion around synchronous vs. asynchronous I/O interfaces. As shown in Figure 3.1, while the asynchronous I/O model helps in better utilizing the CPU on long-running I/O requests, it also introduces additional overhead for the setup and completion of I/O requests, including the tracking of outstanding I/O requests, and for context switching between tasks.

When memory or storage devices use the memory subsystem for providing direct media access through `load` and `store` instructions, this typically implies synchronous, blocking access to the resource (see Figure 3.1). In contrast to an asynchronous I/O interface, the synchronous access paradigm does not introduce additional software overhead. As frequently accessed resources exposed through the memory subsystem typically feature small-granular, low-latency access (e.g., DRAM), the additional overhead of an asynchronous access model would outweigh the benefit of overlapping media accesses and computations. This assumption, however, does not hold when storage devices expose comparatively slow memory technologies, like Flash, through the memory subsystem.

Here, slow memory accesses would block the CPU for extended amounts of time leading to an increase in stall cycles.

Observation $\mathcal{O}$ 3.1: An asynchronous I/O interface enables overlapping media access and computation but introduces additional overhead. Resources exposed through a synchronous memory subsystem block the CPU core for the entire media access. Therefore, only storage with DRAM-like access latencies should be exposed through the memory subsystem while slow storage should be exposed through an asynchronous interface.

To obtain a better understanding of the strengths and weaknesses of both, the synchronous `load/store` interface and the asynchronous block interface, we analyze two technologies that are representative for each interface on a synthetic I/O workload. In this test we use CXL-attached DRAM for the synchronous `load/store` interface and a Flash-based NVMe SSD for the asynchronous block interface. We use a custom build of `fio` [3] based on version 3.37 for simulating an I/O workload that randomly writes into a 8 GiB storage block. As our `fio` modifications (§ 4.4.2) are limited to the mmap I/O engine, they are irrelevant for this test. For each technology, we repeat our test with varying block sizes and number of I/O threads.

As we want to characterize the raw I/O interfaces without the file system distorting our measurements, we run `fio` on the raw storage device. We use a Samsung 970 PRO (1 TB) as NVMe SSD. To minimize the I/O overhead introduced by Linux's storage stack, we use `fio`'s *io_uring* I/O engine with a queue depth of 8 and direct I/O. By setting the queue depth to a higher value, we allow the NVMe device to make use of parallelism in hardware. Note that a per-thread queue depth of 8 might not necessarily give the best results for few threads.

For the synchronous `load/store` interface, we use 16 GiB CXL-attached DRAM featured on an Intel Agilex 7 I-Series Field Programmable Gate Array (FPGA) [18] with CXL hard IP. Using the `daxctl` [19] utility, we configure the CXL-attached DRAM as *devdax* device. We use `fio`'s *dev-dax* I/O engine that memory-maps the devdax device for direct access from user space. While the CXL-attached DRAM is not a real storage device, it is perfectly suitable for emulating the synchronous I/O interface of a potential hybrid storage device because we only use it for modeling performance. Using Intel's memory latency checker (MLC) [21], we measure access latencies of about 350 ns for the CXL-attached DRAM. This is approximately 3.3 times higher than the access latency ($\sim$105 ns) of CPU-local DRAM on our evaluation platform. In Section 5.1, we provide a more detailed description of our testing methodology and hardware setup.

We evaluate both interfaces regarding three metrics: sustained write bandwidth, CPU efficiency, and energy efficiency. For measuring CPU and energy efficiencies, we adopt the efficiency metric for evaluating PM file systems proposed by Werling et al. [62], as it not only captures the performance of the storage device but also the impact on the rest of the system. Analogous to Werling et al. [62], we measure the CPU cost in CPU time

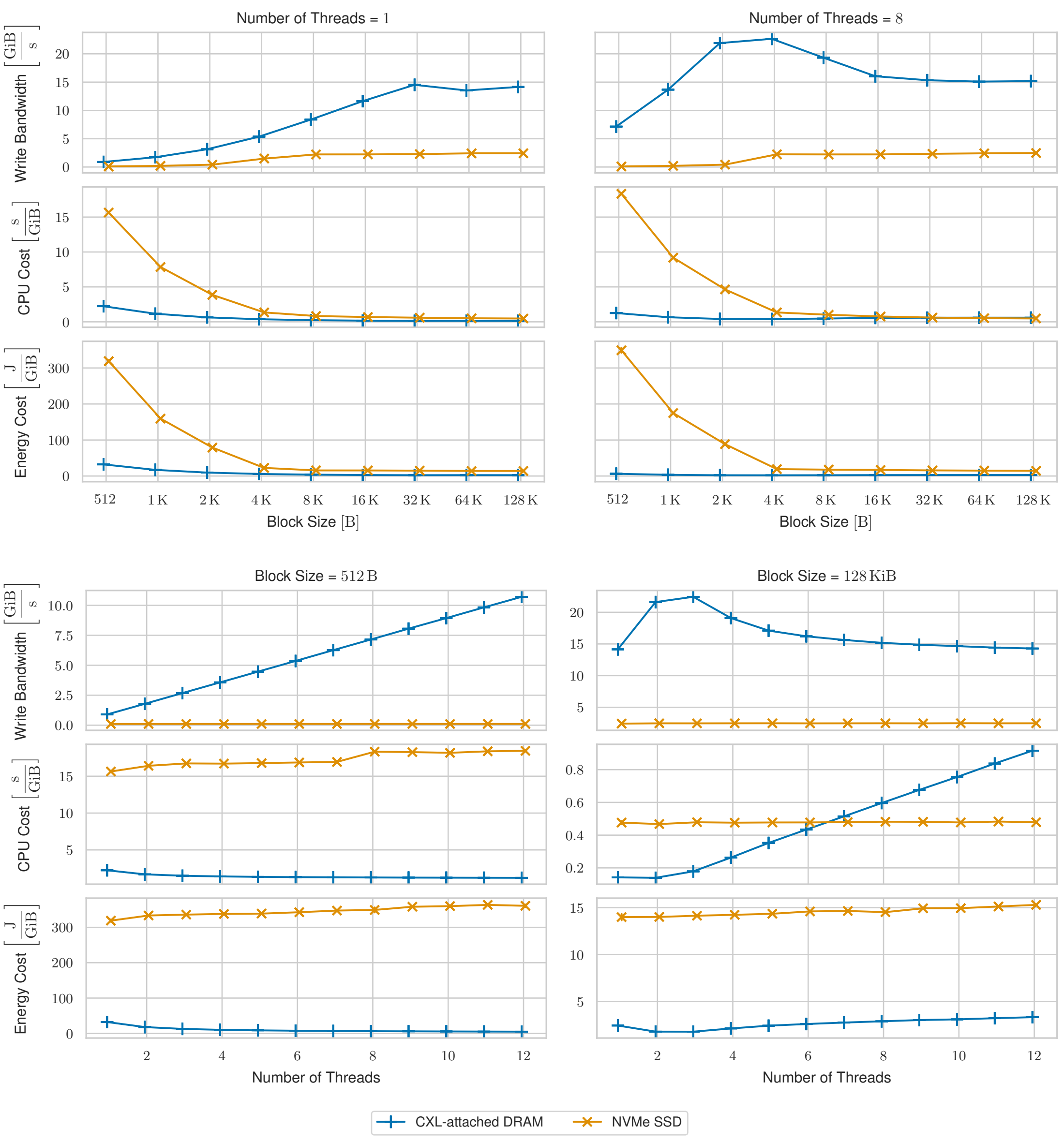

Figure 3.2: Write bandwidth, CPU efficiency, and energy efficiency of the synchronous `load`/`store` interface (CXL-attached DRAM) and the asynchronous block interface (NVMe SSD) as a function of the block size (top) and the number of I/O threads (bottom). The left column shows measurements for a single I/O thread (top left) and a block size of 512 B (bottom left). The right column shows measurements for eight I/O threads (top right) and a block size of 128 KiB (bottom right).

(as reported by `/proc/stat`) per gigabyte written and the energy cost in joules per gigabyte written. In order to maximize its effectiveness, an I/O interface must maximize the throughput while minimizing the CPU and energy costs.

Figure 3.2 shows our measurements for all three metrics as a function of the block size (top) and the number of I/O threads (bottom). Our test shows that the sustained write bandwidth of the NVMe SSD is sensitive to the block size. For a block size of $512\,\mathrm{B}$, we only measure a bandwidth of up to $105\,\mathrm{MiB/s}$, even for a high number of I/O threads. For block sizes of $4\,\mathrm{KiB}$ and up, even few I/O threads can fully saturate the write bandwidth at approximately $2.5\,\mathrm{GiB/s}$. A single thread is sufficient for reaching the maximum bandwidth at a block size of $64\,\mathrm{KiB}$. The CPU overhead introduced by the asynchronous block I/O linearly falls off as the block size increases. When `fio` fully saturates the write bandwidth of the device, the CPU overhead remains stable at a low level. The energy cost behaves similar to the CPU overhead for the NVMe SSD. This shows that block sizes and thread counts that achieve the maximum throughput also achieve the lowest CPU and energy overhead. Therefore, we do not have to trade off one metric against the others.

The CXL-attached DRAM always operates on blocks of $64\,\mathrm{B}$, i.e., one cache line, or smaller words when using non-temporal stores. Therefore, the specified block size only controls the CPU overhead introduced by `fio` for tracking I/O statistics. When we increase the pressure on the synchronous interface, i.e., when we increase the block size or the number of I/O threads, we first observe an increase in the throughput similar to the NVMe SSD. After reaching a peak bandwidth of approximately $22.6\,\mathrm{GiB/s}$, the bandwidth drops when increasing the pressure even further (below $15\,\mathrm{GiB/s}$). We do not know the root cause for this behavior.

When using a single I/O thread, the bandwidth plateaus at around $14\,\mathrm{GiB/s}$ but does not fall off significantly. We expect that the store buffer size of the microarchitecture in combination with the high access latency of CXL-attached DRAM determine the maximum per-core write bandwidth. The CPU overhead stays relatively flat when not going beyond the bandwidth peak. After the bandwidth peak, however, the CPU overhead significantly increases. The use of multiple I/O thread amplifies the CPU overhead in this case. For six I/O threads and up, CXL-attached DRAM even consumes more CPU time per gigabyte than the NVMe SSD at a block size of $128\,\mathrm{KiB}$. We assume the increase in CPU overhead to be caused by an increase in stall cycles due to blocked memory accesses. This effect is also visible for the energy cost.

Observation $\mathcal{O}$ 3.2: CXL-attached DRAM provides better throughput, CPU overhead and energy cost in almost all scenarios, especially for small blocks where the NVMe SSD suffers from read and write amplification.

Observation $\mathcal{O}$ **3.3:** The NVMe SSD gracefully handles high loads without sacrificing CPU or energy efficiency. CXL-attached DRAM, on the other hand, suffers from increasing CPU and energy costs when multiple writers exceed the maximum bandwidth. The OS does not counteract this overload.

Next to performance and energy consumption, price is another aspect that we want to compare between both technologies. While conventional DRAM or emerging memory technologies, like Optane, offer better performance than Flash, they cannot compete on price per gigabyte. To show this difference, we compare the lowest price per gigabyte for DDR 5 ($4800\,\mathrm{MT/s}$) DIMMs and PCIe Gen 4 NVMe SSDs using pricing data obtained from `pcpartpicker.com` (US region) [45] at the time of writing. For DRAM, we see prices of approximately $2.2\,\$/\mathrm{GiB}$ whereas the price for the NVMe SSD comes down to $0.052\,\$/\mathrm{GiB}$. While these prices give only limited insights into the exact cost of the memory technology, they clearly show the large price gap between both technologies.

Observation $\mathcal{O}$ **3.4:** Fast, byte-addressable storage technologies which are suitable for implementing a synchronous `load`/`store` interface, are a major cost driver. Commodity storage technologies that are geared towards an asynchronous block interface, like Flash, are suitable for providing cost-effective storage solutions.

3.2 Hybrid Storage Model

With our observations from the previous section in mind, we now go on to define a device model that serves as basis for designing our OS abstractions. For our device model, we adopt Bae et al.'s concept of a hybrid store that combines a synchronous `load`/`store` interface with a conventional asynchronous block interface [6]. Here, synchronous and asynchronous refer to the interaction between host and storage, i.e., this classification is irrespective of the user space API used for I/O. Contrary to the hardware architecture presented by Bae et al. [6], our hybrid storage model aims to describe such devices at a much higher level without specifying concrete technologies or hardware details. With this approach, we aim to build general hybrid storage abstractions that are not tied to a specific piece of hardware. Similar to 2B-SSD [6] or Samsung's CMM-H [48], our device model features two interfaces for accessing storage, a cache for serving the `load`/`store` interface, and a high-capacity backing storage. Figure 3.3 outlines the hybrid storage model. In the following, we will discuss each component.

The synchronous interface allows for direct access of the storage device using `load` and `store` instructions. While RISC architectures typically use explicit `load` and `store` instructions accessing main memory, some architectures like x86 have no direct equivalent in their Instruction Set Architecture (ISA). For abstracting this behavior, we consider all instructions accessing memory to be `load`/`store` instructions. Contrary to

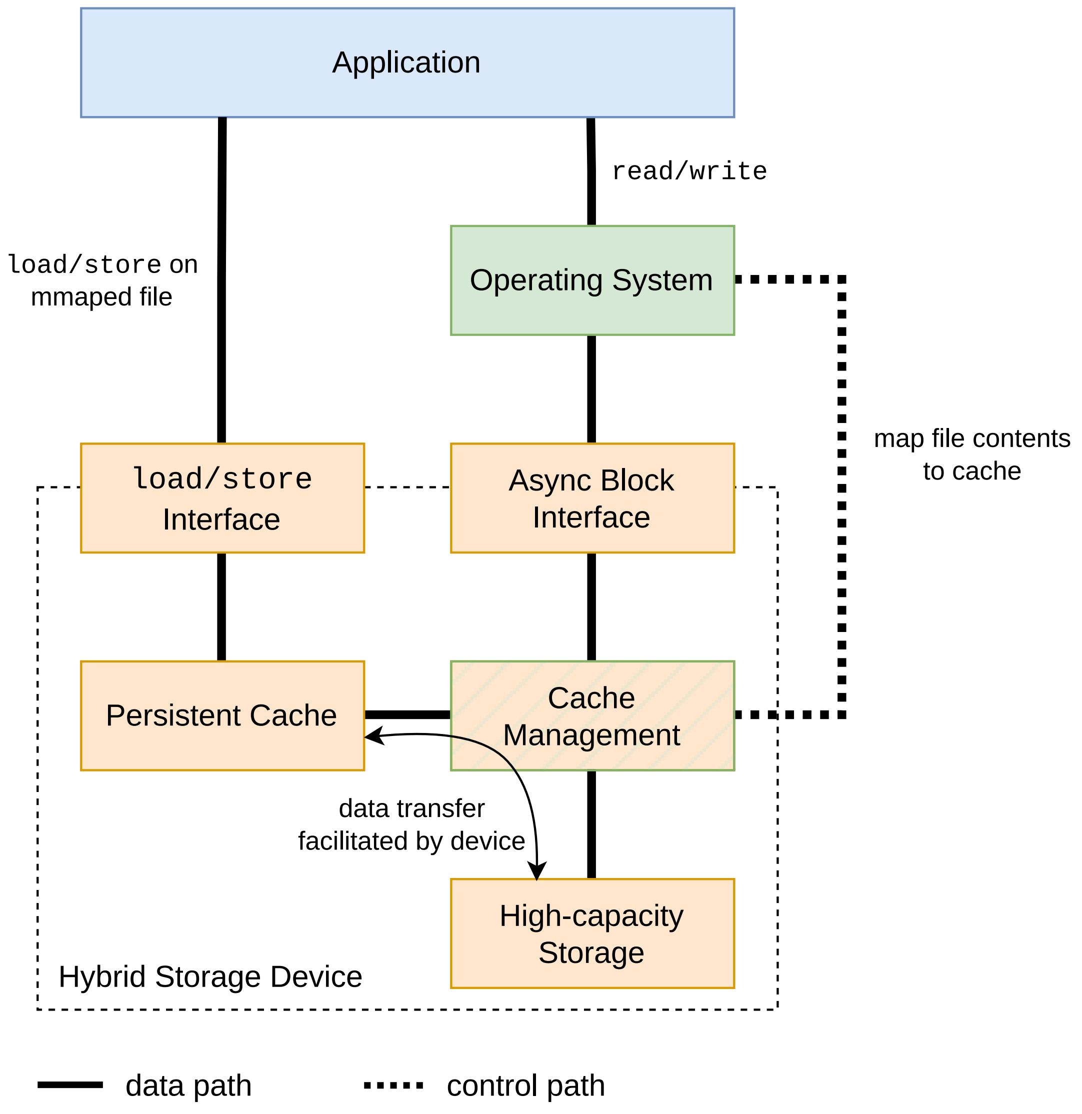

Figure 3.3: Overview of our hybrid storage model. The device features a synchronous and an asynchronous storage interface. Both interfaces provide access the same storage capacity. The hybrid storage device uses a persistent cache for providing low-latency, fine-granular access to storage. A cache management component in the OS or the device handles data movement between the backing storage and the persistent cache and maintains coherence. The operating system orchestrates the cache management.

the asynchronous block interface that allows the OS to hide long storage access latencies by scheduling non-blocked processes, the synchronous `load/store` interface blocks the executing CPU core for the entirety of the storage access. Therefore, the synchronous `load/store` interface must be backed by a storage technology that offers access latencies comparable to DRAM as otherwise I/O through the synchronous interface would waste CPU time ($\mathcal{O}$ 3.1).

In order to implement the synchronous `load/store` interface on top of cost-effective storage technologies like Flash, our hybrid storage model incorporates a cache backed by a byte-addressable memory technology. For our hybrid model, we require a persistence domain that at least includes the persistent cache itself. This means that all writes that reach the cache persist a crash. The cache's persistence domain, however, could go even further, including the CPU caches (see. CXL's GPF, § 2.1). For our model, we assume that pushing writes to the persistence domain takes significantly less time than writing modifications to the backing storage. We leave it up to implementations to define a concrete persistence domain as well as decide if the cache is written back to persistent storage in case of a crash or if the cache is backed by a PM medium that allows for recovery of lost writes. Based on CMM-H's focus on Total Cost of Ownership (TCO) [49], we expect that cost-effectiveness will be a major component in the rollout of hybrid storage devices. Therefore, we assume a hybrid storage devices that aims to minimize TCO. As a low-latency persistent cache drives up the TCO of the hybrid storage device ($\mathcal{O}$ 3.4), we assume a cache-to-storage ratio $r \in [0.001, 0.01]$, i.e., few GiB of cache per TiB of storage, to maintain our goal of optimizing the TCO.

The hybrid storage device further requires a component for managing the persistent cache. For one, this includes the virtualization of the persistent cache. Similar to the concept of virtual memory, the cache management creates the illusion of a persistent cache that is as large as the backing storage. In the virtual memory analogy, the *cache address space* represents the physical address space and the *storage address space* represents the virtual address space. We assume that the hybrid device uses the same page size as the host for the management of the persistent cache. i.e., one cache line in the persistent cache equals one host page. In our model, the OS and the storage device share the mapping between *storage address space* and *cache address space*. The OS can then use this information to reflect the current cache mapping in its page tables in order to fault on user space accessing data not present in the persistent cache. This way, the OS can defer the media access until the requested page is present in the persistent cache and thus guarantee low-latency synchronous access. Similar to the asynchronous I/O interface, this allows the OS to overlap the slow aspect of the cache miss, i.e., the access to the backing storage, with other non-blocked tasks. Further, the cache management handles the data movement between cache and storage, which includes the swapping of data blocks and dirty block tracking. In order to achieve a fully virtualized cache, the cache management needs to implement an allocation and reclaim mechanism.

What sets our hybrid storage model apart from using separate devices for the persistent cache and the backing storage is the data movement between the two. When the cache management component brings in data from the backing storage into the persistent cache, the hybrid device facilitates the data movement, not the host system. For one, this approach cuts the bottom half of the host's storage stack, namely the block layer and the device driver, out of the cache management. As we have already discussed before, the storage stack is a major contributor to the access latency for high-end NVMe drives [31]. Further, only data requested by the host must travel through the interconnect that connects the host and the hybrid storage device. For example, when the host tries to read a cache line worth of data through the synchronous interface and the data is not present in the persistent cache, the device brings a full page into the persistent cache but sends just the cache line off to the host. For workloads that access only a small percentage of each page, this significantly reduces the amount of data transferred to the host, thus reducing interconnect traffic. The idea of reduced I/O traffic was previously explored by Abulila et al. [1].

The last component of our model is the asynchronous block interface. This interface works similar to traditional storage devices and is accessed through the block storage stack of the OS. Our hybrid storage model does not mandate the use of the synchronous `load`/`store` interfaces for working with memory-mapped files. It is up to the OS to decide which interface to use: either direct access to storage through the synchronous interface or indirect access through volatile pages in the *page cache* that are periodically written back using the asynchronous block interface.

Due to the fact that both interfaces provide a view on the same portion of storage, we need to consider coherence of the storage device. We define our model to provide a coherent view on the storage device. This means that writes from the synchronous `load`/`store` interface can be observed from the asynchronous block interface and vice versa. This, however, requires cooperation from the OS when the OS wants to buffer parts of storage in DRAM. We leave it up to future work to define a consistency model that specifies when changes from the other interface can be observed and in which order.

3.3 Software Interface for Hybrid Storage

While the hybrid storage model covers what the OS can expect from hardware, a holistic approach must also cover the interaction between the application and the OS. For this, we first analyze how well storage abstractions of contemporary OSs map to our hybrid storage model. Section 3.3.1 discusses Linux's current DAX API and highlights shortcomings regarding the hybrid storage model. Based on our findings, Section 3.3.2 proposes extensions to the POSIX API [17] which provide applications with direct access to storage. Our reimagined API focuses on compatibility with existing OSs and resource management for the persistent cache featured on the hybrid device.

3.3.1 Case Study: Linux's DAX Subsystem

Linux does support direct access to memory-mapped files through its DAX subsystem and DAX-aware file systems. While we are not aware of a formal device model for the DAX subsystem, the kernel documentation establishes two requirements for implementing DAX support in block device drivers [32]. The first requirement is that the entire storage capacity is accessible through the memory subsystem at all time, meaning that devices cannot use paging to implement direct access through a small cache [32]. The second requirement is less precise. It states that the storage device must not stall the CPU for an extended period [32]. This requirement aligns with our observations from Section 3.1. In summary, Linux assumes low-latency uniform storage access to the entire storage device through the memory subsystem. When comparing this implied model to our hybrid storage device model, it becomes clear that both models are incompatible due to the use of a persistent cache for implementing the synchronous I/O interface. To support our hybrid device model, the DAX subsystem would have to relax the blocking requirements for the `direct_access()` interface [32] so that storage devices can bring in pages that are not present in the on-device cache. Relaxing these requirements, however, would require a significant rework of the DAX subsystem and all its users.

Another major obstacle of supporting our hybrid device model through the DAX subsystem is resource management. As Linux assumes uniform storage access through the memory subsystem at all times, there is no use case that requires DAX mappings while limiting the amount of storage that is DAX mapped at the same time. This is particularly apparent in the current user space API for working with DAX-aware file systems as it provides no fine-granular control over DAX usage. Because the DAX subsystem uses a per-inode flag, all mappings of a particular file must either be fully backed by the storage (DAX) or by system RAM (page cache). Some DAX-aware file systems, like ext2, do not even support per-inode DAX control, but require a `dax` mount option that sets the flag for all files [32].

To motivate the need for more fine-grained control over DAX mappings, we imagine the DAX-aware application *userfat* that implements a simple file allocation table (FAT)-based file system on top of a regular binary file. In order to ensure that the file system is crash-consistent, the application calls `fsync()` [35] for some operations that modify the FAT (e.g., before reusing a block from a deleted file). *userfat* leverages a DAX mapping on the FAT to reduce the cost of frequent calls to `fsync()` after small writes. While only the file system metadata, i.e., the FAT, requires the persistence properties of the DAX mapping, all data must go through the DAX subsystem and therefore through the synchronous I/O interface. As a result of this, accessing data blocks might displace metadata pages in the persistent cache, leading to an increase in slow accesses that stall the CPU.

Even though our example is very basic, we argue that it models a common pattern: a small amount of data in a file (here metadata) observes many small writes and requires strong persistence guarantees (frequently flushed). When applications follow this pattern, Linux's API for DAX does not provide us with the fine-granular control required. Resource management in the context of hybrid storage devices is not limited to just the persistent cache capacity, but also requires bandwidth management. As our observations ($\mathcal{O}$ 3.1 and $\mathcal{O}$ 3.3) in Section 3.1 show, the CPU efficiency drops when the required bandwidth exceeds the maximum sustained bandwidth. If future devices also show the drop in bandwidth when too much load is put on the persistent cache, proper bandwidth management is not only required for the CPU efficiency but also the I/O performance. Similar to the management of cache capacity, Linux's DAX API does not provide a mechanism for limiting the bandwidth of DAX mappings. We limit the scope of this thesis to the management of cache capacity. In future work, we plan to explore approaches for bandwidth management on DAX mappings.

3.3.2 User Space API for Hybrid Storage

In order to allow applications to directly access parts of our hybrid storage device, we use memory-mapped storage that is accessible through the synchronous `load`/`store` interface. We refer to these memory mappings as *DAX mappings*. In the context of this thesis, a DAX mapping does not imply the use of Linux's DAX subsystem, but is a general concept independent of the implementation. We call all pages that back DAX mappings and provide direct-access to storage through the `load`/`store` interface *DAX pages*.

For our software interface, we are going to focus on the POSIX API that uses `mmap` for establishing memory mappings. While this covers Linux as well as many other UNIX-like OSs, Windows uses a slightly different API [42]. We argue, however, that our interface is universal enough to be adapted to other APIs like Windows' file mappings.

We propose two modifications to the POSIX API with the goal of making the capabilities of our hybrid storage model available to user space in a familiar manner. While the POSIX API specifies both `mmap` and `mlock` [17], our modifications make use of API extensions only available in Linux, namely the `MAP_SHARED_VALIDATE` flag [38]. This extension, however, could be ported to other OSs.

`mmap()` is the primary interface for establishing DAX mappings. Instead of using the per-inode DAX flag for determining if the OS must establish a DAX or page cache mapping, we introduce a new `mmap` flag (`MAP_DAX`). For now, we limit the use of `MAP_DAX` to shared mappings of files. In addition, we impose the validation of `mmap` flags through `MAP_SHARED_VALIDATE`. We require this flag because the regular `mmap` API does not fail on unknown flags [38]. Doing so gives users of our API the guarantee that they got a DAX mappings in case of a successful `mmap` call.

`mlock()` is normally used for pinning parts of the virtual address space to main memory [37]. By using `mlock`, applications can ensure that the locked address range is excluded from page reclaim. For our DAX mappings, we define `mlock` to pin the given range to the persistent cache. Due to the device model assumed by Linux's DAX code, there previously was no need for pinning DAX mappings which resulted in `mlock` skipping those pages [56].

Another aspect of `mlock`'s API are resource limits. While POSIX leaves it up to concrete implementations to define limits [17], Linux uses per-process resource limits, so called *rlimits*, for limiting the amount of pages a process can lock to main memory. Since main memory and the persistent cache capacity are two distinct resources that administrators might want to ration separately, we introduce a new *rlimit* (`RLIMIT_-MEMLOCK_HYBRID`) for limiting the amount of pages that can be pinned to the persistent cache.

`madvise()` provides an interface for passing usage hints for virtual memory ranges to the kernel [36]. On Linux the `MADV_DONTNEED` advice flag signals that the kernel can free resources associated with the given memory range [36]. At first glance this might look like a good fit for explicitly evicting pages from the persistent cache. However, `MADV_DONTNEED`'s semantics for shared mappings are inadequate for this use case. While `MADV_DONTNEED` is destructive for anonymous private mappings, i.e., memory contents are lost, contents of shared mappings must be retained [36]. Therefore, Linux does not evict file contents from the page cache on `MADV_DONTNEED`.

As performance-critical applications might not want to rely on the non-deterministic nature of reclaiming pages of the persistent cache on memory pressure, we provide the `MADV_DROPCACHE` advice flag for evicting page cache contents. Contrary to `MADV_-DONTNEED`, `MADV_DROPCACHE` immediately drops the associated pages from the page cache. For regular system memory pages, `MADV_DROPCACHE` causes buffered changes to be lost. For DAX pages, however, we must differentiate between two cases: when the DAX page is clean or contains changes that have not been synced, we free the page immediately. When the DAX page contains synced changes that have not been written to the backing storage, we issue a writeback before freeing the page. This writeback is necessary to uphold the persistence guarantee provided by *sync* operations. Since we write the current page contents to the backing storage, the writeback might include changes that have not been synced. We argue that this behavior falls in line with regular memory-mapped file semantics. Due to the destructive effect of `MADV_DROPCACHE` on shared mappings, we propose that this operation can only be used on writable mappings. While POSIX does not specify `madvise`, we can implement a similar interface on top of POSIX's `posix_fadvise` [17].

3.4 Persistence-aware Page Cache

For the management the hybrid storage device's persistent cache, we take an OS-centric approach that is centered around the concept of a *persistence-aware page cache*. For this, the hybrid device must fully expose the persistent cache's memory to the OS. Contrary to a traditional page cache design that assumes to exclusively use volatile system memory for caching file contents, our persistence-aware page cache allows leveraging properties of the underlying memory technology. In addition, we repurpose the page cache for establishing DAX mappings to user space. By inserting DAX pages into the page cache, we implicitly store the mapping from the storage address space to the cache address space. To obtain this mapping from a page cache page, we require two translations. First, the page cache maps a DAX page to the file address space. Then, the file system translates this file offset into a logical device offset from the storage address space.

In order to bring persistence-awareness to the page cache, we introduce two modifications. The first modification concerns the allocation of pages for caching file contents. In order to decide if the page cache must allocate a page backed by volatile memory or the hybrid device's cache, the OS must track which file ranges are covered by DAX mappings. When DAX mappings cover a faulting file range, the page cache allocates memory from the hybrid device's persistent cache. Otherwise, the page cache allocates volatile system memory. In addition to tracking file ranges covered by DAX mappings, the page cache must store for each cached page whether it is backed by volatile or non-volatile memory. While this might seem redundant, it is strictly necessary as a non-volatile page might outlive the last DAX mapping covering its associated file range. When a file range stops being covered by DAX mappings, the page cache does not remove DAX pages from the range. Doing so, we can avoid additional page faults on future accesses and continue to benefit from the persistence of the backing memory. A potential downside of this approach, however, is that DAX pages linger around non-DAX file ranges, thus increasing memory pressure on the small persistent cache.

The writeback handling is the second modification of our persistence-aware page cache. Here, we differentiate between two causes of writebacks: explicit *sync* operations, like `fsync()` [35], and periodic asynchronous writebacks (e.g., Linux's per-*bdi*[1] writeback flusher thread [4]). As *sync* operations must block the calling thread until the OS can guarantee the persistence of all writes up to the point of the *sync*, they block the caller from making further progress, thus reducing overall performance. Therefore, we aim to minimize the cost of synchronous writeback. To retain the semantic of the *sync*, we must ensure that all modifications reach the persistence domain of the storage device. For traditional storage devices, as well as file contents cached in volatile memory, this comes down to a full writeback to the backing storage device. Depending on the storage device and its protocol, an additional flush command might be required to guarantee that the

[1]`struct backing_dev_info` (bdi) in `backing-dev-defs.h` [56]

storage device does not buffer some I/O operations in volatile caches. For pages backed by the persistent cache, however, our model assumes all writes either already reached the persistence domain or can reach the persistence domain with a comparatively cheap operation (e.g., flush of CPU cache lines). We refer to this type of *sync* operation as *lightweight sync* to differentiate between a *full sync* that writes all modifications to the backing storage.

For *lightweight syncs*, we propose to keep the dirty state of all page cache entries backed by the persistent cache intact instead of clearing it. Doing so enables the periodic asynchronous writeback to still pick up on cached pages that are out-of-sync with the backing storage capacity. Our evaluation (§ 5.2) indicates that it might be beneficial to use three dirty states for tracking the writeback status of DAX pages in the page cache:

- The *clean* state signals that a DAX page is in-sync with the backing storage. All writes are guaranteed to persist a crash.
- DAX pages in the *out-of-sync* state are out-of-sync with the backing storage, but all writes have reached the persistence domain. From a crash-consistency viewpoint, this state is equivalent to the *clean* state.
- *dirty* pages are neither in-sync with the backing storage nor do they guarantee persistence of all past writes.

On *lightweight syncs*, DAX pages in the *dirty* state transition to the *out-of-sync* state while volatile pages transition to the *clean* state. A fourth state that describes a page that is in-sync with the backing storage but does not guarantee persistence of all writes cannot exist. Pages backed by volatile system memory only require the *clean* and *dirty* state.

In the case of asynchronous writebacks, we fully write each *dirty/out-of-sync* page back to the storage capacity even for pages backed by the hybrid device's persistent cache. While this is not strictly necessary for DAX pages, there is no benefit of skipping the writeback similar to the *lightweight sync*, because no user task is blocked from making progress. Additionally, having more pages of the persistent cache be in sync with the backing flash storage even reduces the cost of reclaiming pages. This is especially important when the hybrid device runs out of persistent cache capacity during page allocation. In this case, clean pages can instantly be reclaimed without waiting for their writeback to the backing storage.

The benefits of *lightweight syncs* over *full syncs* on DAX mappings, i.e., all cached pages are DAX pages, are twofold. For one, *lightweight syncs* do not access the storage capacity, thus eliminating the time required waiting for I/O completion. This helps to reduce the *sync* latency, but also uses the more energy efficient I/O interface and prolongs the lifetime of the backing storage. Secondly, *lightweight syncs* shorten the code path for *sync* operations. To complete *lightweight syncs*, the OS does not need to call into the block layer or below as we do not require the asynchronous block interface for completing the *sync*. As discussed before, the overhead of the storage stack becomes increasingly important for high performance I/O when using low-latency storage devices.

In order to provide applications with direct access to the complete storage address space through the small persistent cache, the OS must multiplex the cache between all users. For this, we use an approach that is similar to the concept of virtual memory. File contents are paged into the persistent cache when the persistence-aware page cache allocates a new DAX page. When the hybrid device runs out of free DAX pages, the OS must reclaim DAX pages that are currently in use by the page cache. To reclaim a DAX page, its contents must be written to the backing storage, should the page contents and the backing storage be out-of-sync. On future buffered I/O on reclaimed file ranges, the page cache allocates a new page and fetches the file contents from the backing storage. As the persistence-aware page cache decides on each allocation anew whether it must allocate a volatile page or a DAX page from the hybrid storage device, the faulting file range might have different persistence properties after swapping.

Similar to virtual memory, one downside of the cache virtualization is *thrashing*. When the I/O workload that requires direct access does have a working set that is significantly larger than the persistent cache and the workload does not exhibit temporal locality, the OS will have to aggressively reclaim DAX pages. Here, *thrashing* describes the situation where the application is blocked from making significant progress due to excessive swapping of file contents between the persistent cache and the backing storage. To mitigate the impact of thrashing on performance-critical tasks whose working set is comparatively small, tasks can pin DAX pages to the page cache. This excludes them from the reclaim process. Since pinning DAX pages reduces the amount of cache available to other tasks as well as the number of reclaim candidates, excessive pinning can increase thrashing for other tasks. We leave it up to future work to explore more anti-thrashing mitigations.

The last aspect of the cache management that we need to discuss is the coherence of both I/O interfaces. As most of the storage I/O, i.e., memory-mapped I/O and regular `read`/`write` calls, go through the page cache, the page cache already provides us with a coherent view on our hybrid storage device in most situations. One key exception, however, is direct I/O[2] as it intends to bypass the page cache. To work around this problem, we propose to serve direct I/O requests from the page cache for DAX-mapped file ranges. While serving direct I/O from the page cache might seem contradictory in itself, the fact that we provide direct access to the storage device through the page cache ensures that the OS does not buffer direct I/O. Since our persistence-aware page cache guarantees that DAX pages in the page cache are always up-to-date, direct I/O requests cannot read outdated data. When direct I/O is paired with synchronous I/O (e.g., `O_SYNC` or `O_DSYNC` [40]), the persistence-aware page cache has to uphold persistence

[2]also called *unbuffered I/O*

guarantees. Users of synchronous I/O expect that completed writes cannot be lost as they should have reached the storage device and be flushed from volatile caches. This persistence requirement, however, can also be fulfilled by the persistence-aware page cache by directly writing into the page cache and issuing a *lightweight sync* on the file range before completing the I/O request. When the requested file range is either not present in the page cache or backed by volatile pages, the handling of direct I/O is left unchanged, i.e., the storage device serves the request through the asynchronous block interface.

This approach not only solves the coherency problem, but should improve I/O performance, energy efficiency, and device lifetime by using the synchronous interface over the asynchronous block interface for DAX-mapped file ranges. In addition, this approach reduces write amplification when issuing small direct I/O requests on DAX-mapped file ranges. However, we do not expect to find many workloads that benefit from this because mixing direct I/O and buffered I/O on the same file range seems counterintuitive. For future work, we want to explore dynamically upgrading a file range to DAX in order to speed up direct I/O.

Because DAX mappings assume the strong persistence of DAX pages, volatile pages must not back a DAX mapping. When the page cache only considers the page placement during the allocation of page cache entries, the page cache cannot maintain this property. Assuming an existing file mapping with volatile pages in the page cache, the creation of a new DAX mapping that intersects with this file range violates the persistence guarantee of the DAX mapping. To solve this problem, we propose a *DAX upgrade* mechanism that restores the persistence guarantee of our persistence-aware page cache. When establishing a new DAX mapping, the page cache tries to preemptively migrate all volatile pages covered by the new mapping to the persistent cache. Should the hybrid device run out of new DAX pages during the *DAX upgrade*, we propose to truncate volatile page cache pages that cannot be migrated. In order to not lose any writes buffered in volatile pages, truncation of dirty pages requires writeback. The page cache re-fetches truncated parts of the page cache on-demand from the backing storage.

We argue that reclaiming DAX pages during the *DAX upgrade* can drastically increase the `mmap` latency due to I/O during writeback. In addition, upgrading more pages to DAX puts even more pressure on the already heavily utilized persistent cache. As it is uncertain whether application access preemptively migrated pages in the near future, we must carefully balance the potential benefit of *DAX upgrades* against their aforementioned cost. Here, we decide to truncate the page cache in favor of reclaiming DAX pages.

When DAX and non-DAX mappings intersect in the file address space, the intended behavior of `mlock` is unclear. On one hand, we expect the OS to charge a `mlock` call on a non-DAX region against the *rlimit* for pinning pages to DRAM. On the other hand, overlapping mappings with different persistence-requirements imply that all pages must be backed by the persistent cache. Therefore, we have also good reasons for charging the `mlock` against the *rlimit* for DAX pages. For now, we propose to fail `mlock` calls on file ranges that have both, regular and DAX mappings. *DAX upgrades* weaken the isolation between processes, because a process can trigger *DAX upgrades* that happen completely transparent for other processes and cause `mlock` to fail under certain conditions.

Chapter 4

Implementation

Based on the description of our persistence-aware page cache design from the previous chapter, we outline the implementation of our hybrid storage support and modifications to two existing applications in this chapter. First, we present our modifications to the core memory management code and the page cache of the Linux kernel [56] (§ 4.1). Since we do not have access to hardware that fits our hybrid storage model, we implement a block device driver that emulates a hybrid storage device by wrapping a regular block device and a memory device (§ 4.2). Next, we outline our modifications to the ext2 and ext4 file system drivers (§ 4.3). Finally, we present modifications to `fio` [3] and Valkey [57] that make use of the DAX capabilities offered by hybrid storage (§ 4.4).

4.1 Kernel Support for Hybrid Storage

For implementing kernel support for our hybrid storage model, we assume that the hybrid storage device exposes its persistent cache as separate memory-only Non-Uniform Memory Access (NUMA) node. Doing so, we can reuse Linux's page allocator as well as its page reclaim mechanism without any additional modifications. In addition, our implementation assumes that the hybrid storage device's persistence domain covers the CPU caches, i.e., no flush of CPU cache lines is required for guaranteeing persistence. Considering that, as described in Section 2.1, features like CXL GPF [7, § 9.8] or eADR [20] are available today, we argue that this is a realistic assumption.

For now, our implementation is limited to persistence on the device level. This means that the device will persist all writes to DAX pages, but on the file system level we might lose data regardless. This has to do with the handling of file system metadata in our current implementation of the persistence-aware page cache. Section 6.2 describes this problem in more detail and outlines a solution. We base our implementation on the Linux kernel version 6.6 [56]. Overall our kernel modifications, excluding the hybrid device emulation (§ 4.2), account for approximately 1500 Lines of Code (LoC).

4.1.1 Hybrid Storage Representation

As hybrid storage devices share many properties with conventional storage devices (e.g., the asynchronous block interface), we implement them as block devices with extended capabilities for DAX. For this, we introduce a *hybrid device* abstraction (see Listing 4.1). Block devices that feature hybrid storage support include a reference on the *hybrid device*. The hybrid block device driver is responsible for allocating and registering the hybrid device in the block device. File systems and other system code can detect whether a block device supports hybrid DAX capabilities by checking the reference included in the block device. Each *hybrid device* belongs to exactly one block device.

To allow file system and page cache code to allocate pages from the NUMA node representing the persistent cache, each *hybrid device* includes the NUMA node identifier n in addition to a node mask with only the n'th bit set. While this might seem redundant, the API used for allocating pages in the page cache (`__folio_alloc()` [56]) requires a node mask and a preferred node ID within the node mask. In order to not have to prepare a node mask on each allocation, we include a preallocated node mask in the *hybrid device*. In addition, *hybrid devices* allow the backing block device driver to attach implementation-specific per-device data to each *hybrid device*.

```
struct hybrid_device {
    /* numa node id for page allocation */
    int nid;

    /* node mask with only nid set */
    nodemask_t *nvm_node_mask;

    /* private driver data */
    void *private;

    // ...
};
```

Listing 4.1: Simplified definition of in-kernel *hybrid device* representation.

4.1.2 Memory Management Modifications

To support fine-granular control over DAX mappings, we introduce the `MAP_DAX` flag for `mmap()`. As we have already mentioned in Section 3.3.2, `MAP_DAX` is only valid in conjunction with the `MAP_SHARED_VALIDATE` [38] mapping type on memory-mapped files. For private mappings (`MAP_PRIVATE`) and regular shared mappings (`MAP_SHARED`), `mmap` silently drops the `MAP_DAX` flag. When applications successfully request a DAX mapping using the `MAP_DAX` mmap flag, the resulting Virtual Memory Area (VMA) receives the `VM_DAX` *vm flag*.

When applications memory-map cacheable objects, e.g., regular files or block devices, Linux inserts the resulting VMA into a per-object interval tree that belongs to the `address_space` representing the contents of the object [56]. Linux implements this interval tree as augmented red-black tree that caches the leftmost node [30, 56]. Linux uses the interval of the logical file range covered by the VMA for insertions into this interval tree. In order to avoid having to scan the entire tree of VMAs when exclusively looking for DAX mappings, we add another interval tree that contains all DAX VMAs (i.e., that have the `VM_DAX` flag) to each `address_space`. This interval tree is used in addition to the tree that contains all mappings, meaning that DAX mappings are present in both while non-DAX mappings are only present in the full VMA tree. Using the interval tree of DAX mappings, we can determine in $\mathcal{O}(1)$ if a file has any DAX mappings, and efficiently iterate over all of them.

Whenever memory mapping updates cause a DAX mapping to be inserted to the mapping interval trees, Linux has to ensure that no pages in volatile memory back DAX mappings. To uphold this invariant for DAX mappings, we proposed *DAX upgrades* in Section 3.4. At the time of writing, our prototype does not implement the speculative migration of pages from volatile system memory to the persistent cache. Instead, our prototype always truncates all volatile pages from the file range that is DAX mapped. For future work, we intend to implement the migration of pages through Linux's in-kernel `migrate_pages()` API [29]. The page migration may stop early when the migration would otherwise cause slow I/O due to memory pressure on the persistent cache or a source page being under writeback. In case any pages backed by volatile memory remain, they get truncated from the page cache.

Besides changes to memory mappings, we modify the behavior of `mlock()` [37] on DAX mappings. Instead of charging all pages against the `RLIMIT_MEMLOCK` [37] *rlimit*, our prototype introduces the `RLIMIT_MEMLOCK_HYBIRD` *rlimit* that limits the number of locked DAX pages per task. When a call of `mlock()` exceeds either `RLIMIT_MEMLOCK` or `RLIMIT_MEMLOCK_HYBRID`, the operation fails without locking any pages. Contrary to `mlock()`, our implementation does not support locking DAX pages using the `MAP_LOCKED` mmap flag as it may silently fail to populate the mapped range [38]. We argue that the silent failure of locking DAX pages might result in unexpected performance, which was the primary reason for locking DAX pages in the first place. Therefore, we decide to not support `MAP_LOCKED` over providing watered-down `mlock` semantics to user space. As of now, we do not implement per–hybrid device limits for the number of locked DAX pages across all tasks.

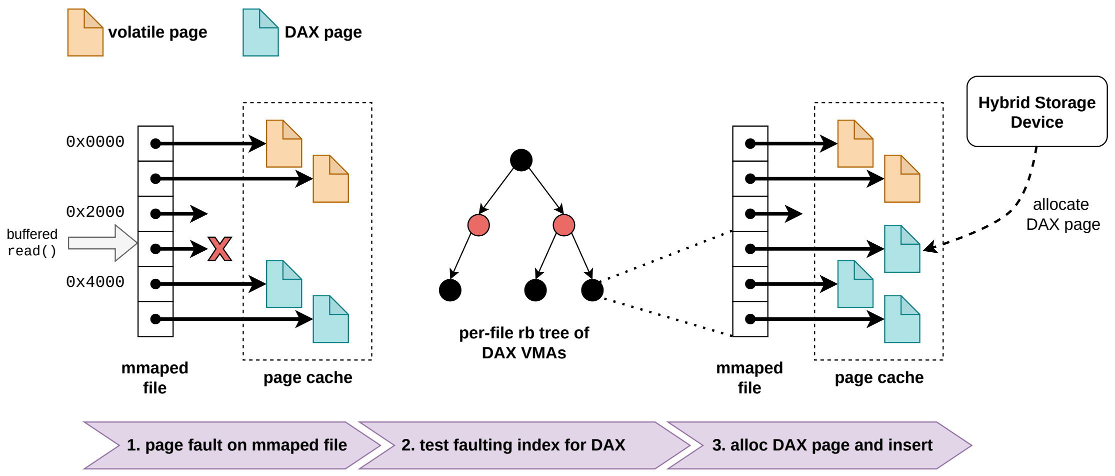
volatile page
DAX page
0x0000
0x2000
0x4000
buffered read()
mmaped file
page cache
per-file rb tree of DAX VMAs
Hybrid Storage Device
allocate DAX page
1. page fault on mmaped file
2. test faulting index for DAX
3. alloc DAX page and insert

Figure 4.1: Page fault handling on DAX region for memory-mapped file. The interval tree of DAX VMAs determines the page type for page cache allocations.

4.1.3 Persistence-Aware Page Cache

In order to decide if our persistence-aware page cache must allocate volatile or DAX pages, it first looks up whether the backing storage is a hybrid storage device. If this is the case, the persistence-aware page cache then goes on to check whether the faulting file range intersects with a DAX mapping using the DAX mapping interval tree (see Figure 4.1). As this interval tree is associated with the file object and not the current task, the intersecting DAX mapping can originate from any task. In other words, we look for intersections in the file address space and not the virtual address space of a particular task. If such an intersection exists, the persistence-aware page cache allocates the requested number of physically contiguous pages from the NUMA node that is set in the *hybrid device*.

From version 5.18 onwards, Linux's page cache does support inserting large *folios* as single page cache entries [66]. *Folios* are a new type for memory management that aims to replace compound pages[1] in the Linux kernel [64]. As a result of page cache entries potentially being larger than the page size, we might encounter the situation where the page cache allocates a large *folio*, i.e., with an order > 0, for file contents, but the file range is only partially covered by DAX mappings. Since our current implementation only requires an intersection with a DAX mapping, the persistent cache will have to fulfill the entire allocation regardless.

[1]In Linux *compound pages* of order n are a set of physically contiguous pages with 2^n pages.

While this results in some subpages being unnecessarily placed in the persistent cache, we argue that this is an acceptable trade-off. For one, we expect that partial overlappings rarely occur because all of the following three conditions must be met: **(1)** the file system[2] must support large *folios* [66, PATCH 71/75]. **(2)** the page cache must allocate a large *folio*. This might happen during *readahead* [66, PATCH 71/75] or on huge page mappings [66, PATCH 74/75]. And **(3)**, the file range covered by the large *folio* must be strictly partially covered by DAX mappings. As the page cache limits the maximum order used for allocating large *folios* (2 MiB[3] on x86 with Transparent Huge Page (THP) support[4]), we think that the impact on the allocation granularity of our approach is manageable for few partial overlappings.

Apart from the expected small impact on the persistent cache, there are also advantages to not having to break up large *folio* allocations so that there are no partial overlappings. For one, it keeps our implementation simple as we do not have to modify the logic for large *folios*. Secondly, our implementation only has to find a single intersection, which is algorithmically less costly than calculating the exact overlappings. Finally, the page cache benefits from a decrease in individual page cache entries which should translate to better performance.

In order to differentiate between volatile and DAX pages efficiently, we introduce an additional page flag, namely `PG_hybrid_cache`, to the per-page metadata, i.e., `struct page` [56]. As pages in the page cache might be mapped through DAX and non-DAX VMAs or even no VMAs at all, the `VM_DAX` vm flag of VMAs is not suitable for this task. While we could use the NUMA node ID embedded in the per-page metadata in combination with the *hybrid device* for this differentiation, we eliminate frequent lookups of the *hybrid device* from hot code paths by using a dedicated page flag.

For writeback, Linux uses the `struct writeback_control` to communicate what file range needs writeback and various flags that control how writeback is done (e.g., is the writeback synchronous) [56]. To optimize the synchronous writeback path for DAX pages, we introduce the `skip_nvm` flag to the `writeback_control`. When this flag is set for a writeback operation, file system code must use a *lightweight sync*, as described in Section 3.4, over a *full sync* for DAX pages. Since we assume that the hybrid storage device's persistence domain covers the CPU caches, the *lightweight sync* breaks down to a *noop*. For our implementation, we leave the dirty state of the page and the associated page cache entry intact so that future asynchronous writebacks pick up on them. This means that our current implementation does not differentiate between *out-of-sync* and *dirty* pages (cf. dirty states in Section 3.4, Page 26).

[2]As of Linux version 6.6, we are only aware of large *folio* support in afs, erofs, xfs, and shmemfs (see users of `mapping_set_large_folios()` [56]).

[3]The allocation size is determined by `MAX_PAGECACHE_ORDER` in `include/pagemap.h` [56]

[4]Large *folio* support currently relies on THP (see `mapping_large_folio_support()` [56]).

As users of the `writeback_control` ensure that unspecified flags default to zero [56], the default behavior is not to optimize writeback of DAX pages. Instead of making all writebacks use the optimized DAX path and specifying exceptions (e.g., for periodic asynchronous writeback), we opt for selectively choosing writeback code paths to optimize. We argue that this approach is safer because our writeback optimizations have side effects that caller should be aware of, e.g., the file range is guaranteed to persist but remains in the dirty state.

Our implementation currently optimizes three writeback code paths: **(1)** the code path of `fsync()` and `fdatasync()` [35] for the ext2 and the ext4 file system drivers, i.e., `generic_buffers_fsync()` for ext2 and ext4 without journaling and `ext4_-sync_file()` for ext4 with journaling. `msync()` [39] use the same kernel-internal code path as `fsync()` but restricts writeback to a given range. **(2)** the `sync_file_-range()` and `sync_file_range2()` system calls [41]. And **(3)** the direct I/O code path for users of Linux's *iomap* [33] facilities, namely `iomap_dio_rw()` [56]. Linux's *iomap* facilities provide a mapping layer abstraction that aims to replace *buffer heads*. For more context regarding *buffer heads* and the efforts behind replacing them, we refer to Corbet's *"A kernel without buffer heads"* [12].

As direct I/O aims to bypass the volatile page cache, Linux must carefully maintain coherence between buffered and direct I/O. For direct I/O reads, Linux flushes dirty pages in the file range before serving the direct I/O request [33]. For writes, Linux not only flushes dirty pages, but also invalidates the file range from the page cache before and after the I/O request [33]. In order to avoid slow storage I/O due to writebacks and unnecessary page cache invalidations, we modify the direct I/O path to skip these operations on DAX pages. For this, the file system must signal hybrid storage support to `iomap_dio_rw()` through the newly added `IOMAP_DIO_HYBRID` direct I/O flag. When this flag is set, the behavior of direct I/O changes as follows: first, direct I/O handles writebacks of dirty pages like in the case of `fsync()`. Secondly, direct I/O skips the invalidation of page cache entries backed by DAX pages. And finally, direct I/O serves I/O requests that are backed by DAX pages on the page cache, meaning that I/O is implemented as `memcpy` on the CPU. In order to serve the I/O request from the page cache where possible, the file system must split the requested range into chunks that are either fully backed by DAX pages or not backed by any DAX page. While those chunks that are backed by DAX pages can be served from the page cache, all other chunks are served through the block interface.

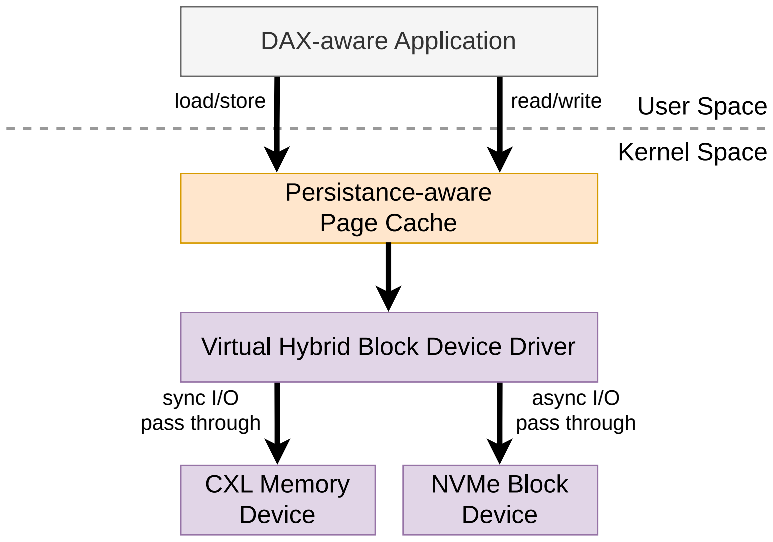

Figure 4.2: Architecture with virtual hybrid storage device backed by NVMe block device and CXL-attached memory for emulating the persistent cache. The virtual device dispatches I/O to the appropriate backing devices depending on the I/O interface used.

4.2 Hybrid Storage Device Emulation

At the time of writing, there is no commercially available storage device that fits our hybrid storage model. Therefore, we decide to emulate a hybrid storage device for the evaluation of our implementation. For this, we implement a hybrid block device driver, namely `vhybrid`, that allows users to create a virtual hybrid storage device from an existing block device and a (memory-only) NUMA node for emulating the persistent cache. Figure 4.2 shows the conceptual I/O flow for our emulated devices. Depending on the I/O interface used, the virtual hybrid storage device dispatches the request to the appropriate backing device.

As virtual devices provide a simple wrapper around a block device and a memory node, we cannot move storage pages into the emulated persistent cache without using Linux for I/O. This is a fundamental limitation of emulating hybrid storage devices and limits our ability to faithfully model performance characteristics of real-world hardware.

To create a virtual hybrid device, our kernel module accepts a block device and a NUMA node ID as optional parameters. While our kernel module does support the creation and deletion of multiple virtual hybrid storage devices, we currently do not export this interface to user space. In the future, we intend to support virtual hybrid device management through a special character device that offers device management through an `ioctl`-based interface. To ease the use of this interface, we also intend to add a user space component that allows to access the management interface through the command line. The `vhybrid` block device driver accounts for 320 LoC.

4.3 File System Support for Hybrid Storage

While most of our modifications concern generic memory management code that is used throughout the entire kernel, hybrid storage devices also require small changes to file systems. For our implementation, we currently support the ext2 and ext4 file system drivers. The scope of our current implementation is limited to ext4 without journaling. Therefore, journaling does not benefit from any of our DAX optimizations. As some of our modifications change kernel-internal APIs, various other file systems that use them require small changes, i.e., less than 5 lines of code, in order to compile. These other file systems do not use the synchronous interface of the hybrid storage device and rely on the block interface entirely.

Our modifications to the ext2 and ext4 driver follow a similar pattern. For both, we add a new mount option, i.e., `-o hybrid`, that signals the driver to use the kernel's hybrid storage support. This mount option requires the underlying device to be a hybrid storage device and the file system's block size to match the page size. The requirement on the file system's block size simplifies certain aspects of the implementation. Linux's DAX subsystem currently suffers from the same limitation [32]. For future work, we plan to support block sizes that are a multiple of the page size. In addition to the mount option, we also add the `MAP_DAX` mmap flag to the bit mask of mmap flags supported by the file system.

Both file system drivers tag dirty pages for synchronous writeback, e.g., `fsync()` or `msync()` with `MS_SYNC`. For this, the file system iterates over all page cache entries marked dirty and tags them with a `TOWRITE` mark. This approach to tagging dirty page cache entries for writeback is a mechanism for livelock avoidance [26]. When iterating over dirty page cache entries, we modify the file system to check whether a page cache entry is backed by DAX pages. For this, we check the `PG_hybrid_cache` page flag of the *folio* associated with the entry, as introduced in Section 4.1.3, and skip tagging the page for writeback if the flag is set.

The most intricate modification of each file system is the optimization of direct I/O on DAX pages. As the generic direct I/O code can only serve file ranges from the page cache that are fully backed by DAX pages, the file system must split the requested file range. For file systems that use the iomap framework, the `iomap_begin()` function obtains the largest file system mapping that the file system can create for the requested file range [33]. Since file ranges backed by DAX pages require a different I/O backend than all other file ranges (`memcpy` vs. block I/O), the file system must ensure that each mapping handed out by `iomap_begin()` has an unambiguous I/O backend. In order to meet this requirement, we modify `iomap_begin()` so that it further breaks up the I/O range depending on DAX pages.

Algorithm 1: Pseudocode for splitting direct I/O request into chunks served through the page cache and chunks served through the block layer.

```
// range in interval tree
struct Span
    $is\_hole$,  // if span is not covered by intervals
    $start$,     // start of this span
    $end$,       // end of this span (inclusive)

// returns the largest possible span in interval_tree
// that fulfills $span.start = start \wedge span.end \in [start, end]$.
// If start is not covered by any interval in the
// interval tree, is_hole is set and the returned
// span represents the hole.
fn GetSpan($interval\_tree$, $start$, $end$) $\rightarrow Span$

// find the first buffered page of $file$ in $[start, end]$
fn FilemapFindPage($file$, $start$, $end$) $\rightarrow Page \mid \perp$

// count the number of consecutive pages buffered for
// $file$ in the file range $[start, end]$ starting at $start$
fn FilemapCountContig($file$, $start$, $end$) $\rightarrow Num$

Input: The $file$ under I/O, the index of the first page in the I/O range $start$, and
       the last possible page index in the I/O range $max\_end$.
fn NextChunk($file$, $start$, $max\_end$)
    $span :=$ GetSpan($file.dax\_vma\_tree$, $start$, $max\_end$)
    $chunk\_end := span.end$
    $from\_pagecache := false$
    if $\neg span.is\_hole$ then
        $first :=$ FilemapFindPage($file$, $start$, $span.end$)
        if $first \neq \perp$ then
            if $first.index \neq start$ then
                $chunk\_end = first.index - 1$
            else
                $cnt :=$ FilemapCountContig($file$, $start$, $span.end$)
                $chunk\_end = start + cnt - 1$
                $from\_pagecache = true$
    return ($start$, $chunk\_end$, $from\_pagecache$)
```

Algorithm 1 outlines our splitting logic for the `iomap_begin()` function in pseudocode. Here, we assume that the file system block size matches the page size of the host. For each file system mapping, we calculate the largest possible prefix[5] p of a file range that is either fully backed by DAX VMAs or only non-DAX VMAs (Line 17). If only non-DAX VMAs cover p, the file system tries to obtain a mapping for p and serves it through the block layer. If, however, p is covered by DAX VMAs (Line 20), p may be backed by DAX pages. Because the existence of DAX VMAs does not imply that any DAX pages have been faulted in, we search for the first buffered page in p (Line 21). If no such page exists in p, we continue as with the non-DAX interval before. If such a page exists but p does not start with it, the file system continues with the file range just up to this page (Line 24). When p does start with a DAX page, we count the number of consecutive DAX pages (Line 26) and pass this sub-range of p to the file system. Only in this case, we can serve I/O through the page cache.

4.4 Evaluation Targets

For our evaluation, we modify the flexible I/O tester `fio` [3] and the key-value datastore Valkey [57], a fork of the popular in-memory key-value store Redis [51]. For Valkey, we implement a new backend for Valkey's Append Only File (AOF) [59] that leverages our hybrid storage API for improving the cost of frequent calls to `fsync()` (§ 4.4.1). We use Valkey version 7.2.5 as basis for our modifications (800 LoC). For `fio`, we implement a new DAX option for the mmap I/O engine that uses our hybrid storage API (§ 4.4.2). As basis for our implementation, we build upon `fio` version 3.37. Our changes to `fio` account for 150 LoC.

4.4.1 DAX-aware Key-Value Store

Even though Valkey is an in-memory datastore [57], it provides two mechanisms for persisting data [59]. The first option is RDB persistence which performs periodic snapshots of the dataset [59]. The second option is the AOF persistence mode which writes out all operations required for reconstructing the dataset, i.e., write operations, into AOF files [59]. For our hybrid storage optimizations of Valkey, we solely focus on AOF persistence.

For AOF persistence, Valkey uses two types of AOF files [59]: One is the *base file* that contains all write operations required for reconstructing the dataset at the time of the last AOF rewrite. The second type are *incremental files* that contain write operations that occurred after the last base AOF file was written. As the AOF grows with each write operation, Valkey performs AOF rewrites in a forked-off child process when the AOF

[5]For each closed interval $r := [r_1, r_2] \subset \mathbb{R}$, we call $s := [r_1, x] \subset \mathbb{R}$ prefix of $r \Leftrightarrow x \leq r_2$

size grows too large [59]. During these rewrites, Valkey writes a new AOF base file that contains a minimal set of operations that result in the datastore state at the time the rewrite started [59].

To ensure that data written to a AOF file reaches persistent storage, Valkey uses `fsync()`[6] on the AOF file. To control when Valkey calls `fsync()`, Valkey offers three AOF `fsync()` policies: `appendfsync=always` calls `fsync()` after each write to the AOF, `appendfsync=everysec` calls `fsync()` in intervals of one second, and `appendfsync=no` never calls `fsync()` on the AOF [59]. For the latter, Valkey relies on the OS's periodic writeback of dirty pages. Valkey only writes to the AOF before sending out replies and waiting for new events to process (see `beforeSleep()` in `server.c` [57]). In each iteration of its event loop, Valkey might process commands from multiple clients. A single write to the AOF might contain commands from multiple clients or even multiple commands from a single client (pipelined execution [60]) [59].

Since Valkey uses a single thread for processing commands, long-blocking calls to `fsync()` reduce Valkey's performance. While Valkey's documentation suggests that the *everysec* policy provides a good trade off between persistence and performance, it describes the *always* policy as "Very very slow, very safe" [59]. In order to reduce the overhead of frequent `fsync()` calls, we use our hybrid storage API to eliminate slow I/O from Valkey's command processing.

As the name of the AOF suggests, Valkey opens AOF files in append mode (i.e., `O_APPEND` [40]) and uses regular `write()` system calls for appending data. Since our hybrid storage API requires a DAX mapping for the AOF to exist in order to benefit from our `fsync()` optimizations, we use `mmap()` on the AOF files. While we could just memory-map the AOF files and leave the use of `write()` unchanged, thus forcing `write()` to access DAX pages, we implement a new mmap backend for AOF persistence instead. By using memory-mapped I/O on hybrid storage devices for writing to the AOF, Valkey can not only benefit from our `fsync()` optimizations but also from the elimination of the syscall overhead introduced by writes to the AOF.

When Valkey uses the AOF mmap backend, it creates a fixed-sized memory-mapping for the file. We currently use a $1\,\mathrm{TiB}$ large sparse mapping so that we do not have to resize the mapping when the AOF grows. The general idea of the mmap backend is to bring only a small part of the AOF into the page cache just before it is needed. As the AOF exclusively grows downwards and never re-reads data during normal operation, Valkey can evict parts of the AOF from the page cache after they have been written. Figure 4.3 outlines this approach. Here, `cur` points to the current write position in the AOF. To write to the AOF, our mmap backend copies the request to the current write position and increases `cur`. `end` marks the End of File (EOF). All bytes between `cur` and `end` must be zero. Should `cur` ever reach the EOF, Valkey cannot write new operations to the AOF. Therefore, the mmap backend initiates a resize of the AOF file

[6]On Linux, Valkey uses `fdatasync()` [35] in favor of regular `fsync()` (see `config.h` [57]).

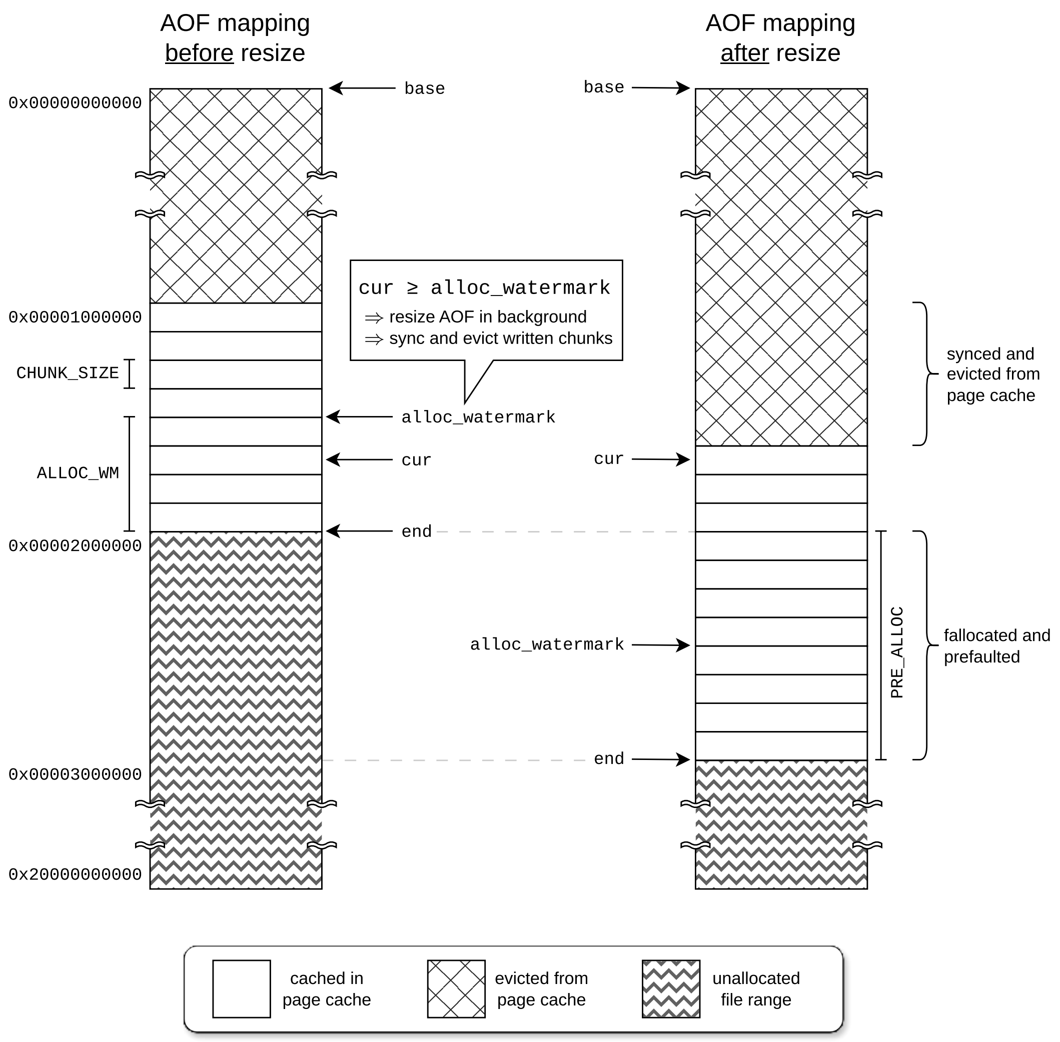

Figure 4.3: AOF resize operation for the AOF mmap backend.

as soon as `cur` reaches the `alloc_watermark`. We offload this resize operation to a background thread[7] to not block the main thread from processing events. Only when `cur` reaches the EOF before the resize completes, the main thread must be blocked.

.

For resizing the AOF file, the background thread uses `fallocate()` [34]. As `fallocate()` requires file system support [34], we mandate `fallocate()` support for all users of our mmap backend. After `fallocate()` succeeds, the background thread pre-faults the newly allocated blocks for writing, calls `fsync()` to persist all previously written operations, and finally drops the AOF contents up to `cur` from the page cache. Here, we use the `MADV_DROPCACHE` advise, that we described in Section 3.3.2, for evicting AOF contents from page cache and `MADV_POPULATE_WRITE` [36] for pre-faulting new blocks. To complete the resize, the background thread sets `end` to the new EOF.

The mmap backend has three settings for managing the memory-mapped AOF file: **(1)** the `CHUNK_SIZE` determines the granularity we use for managing the memory-mapping of the AOF file. When resizing the file or when evicting AOF contents from the page cache, we always operate in multiples of the `CHUNK_SIZE`. **(2)** `ALLOC_WM` determines where the mmap backend should place the `alloc_watermark`. After each AOF resize, the mmap backend defines a new `alloc_watermark` as follows:

$$\texttt{alloc\_watermark}_{new} := \texttt{end}_{new} - \texttt{CHUNK\_SIZE} \times \texttt{ALLOC\_WM}$$

And **(3)**, the `PRE_ALLOC` setting determines the amount of chunks that the mmap backend allocates during resize. As we evict written chunks from the page cache during the resizing process that is only triggered after the allocation watermark is reached, we can obtain an upper limit on the number of buffered AOF pages in the page cache. This upper limit is given as follows:

$$\texttt{num\_live\_aof\_pages} \leq \frac{(2 \times \texttt{PRE\_ALLOC} + \texttt{ALLOC\_WM}) \times \texttt{CHUNK\_SIZE}}{\texttt{PAGE\_SIZE}}$$

Because we pre-fault new chunks before evicting written ones, the page cache might contain newly allocated chunks and to-be-evicted chunks simultaneously. For our evaluation, we define `CHUNK_SIZE`, `ALLOC_WM`, and `PRE_ALLOC` as follows:

$$\texttt{CHUNK\_SIZE} := 2\,\mathrm{MiB}, \qquad \texttt{ALLOC\_WM} := 4, \qquad \texttt{PRE\_ALLOC} := 8$$

[7]Valkey maintains a pool of worker threads for certain background operations (see `bio.c` [57])

During startup, Valkey rebuilds the dataset from the AOF files if they exist [59]. For this, Valkey reads the AOF files until reaching the EOF. As our new mmap backend fills unused parts of the AOF, i.e., the parts between `cur` and `end`, with null bytes, we need to modify the AOF parsing to stop when reaching trailing null bytes. Should any non-null bytes follow after a sequence of null bytes, the AOF is corrupted. We currently do not implement recovery of a corrupted AOF.

In order to leverage DAX mappings in Valkey, we add a DAX mode to our AOF mmap backend. When Valkey uses the AOF mmap backend in the DAX mode, AOF files are memory-mapped with the `MAP_DAX` mmap flag. Since this should significantly decrease the overhead of `fsync()` calls on AOF files, we expect this mode to be especially beneficial for the AOF `fsync()` policy that syncs after every write to the AOF.

While the use of DAX mappings help in removing slow storage I/O from the event processing, they cannot completely get rid of slow I/O but just defer it to a later point in time. The persistence-aware page cache writes back AOF contents either during periodic asynchronous writebacks or when the respective pages are evicted from the page cache. Since our AOF mmap backend moves the eviction of AOF contents from the page cache to the background thread that handles the resizing process, the slow I/O during writeback does not slow down event processing in the main thread. In other words, the majority of the original `fsync()` overhead is deferred to one of Valkey's background workers.

4.4.2 DAX-aware `fio` mmap I/O Engine

For benchmarking synthetic I/O workloads using our hybrid storage support, we add two new options to `fio`'s mmap I/O engine. This I/O engine memory-maps the file for the given `fio` job and uses `memcpy` for implementing read and write. When `fio` must sync data after writes, this I/O engine uses `msync()` on the memory-mapped file. The first option that we add is the `dax` option that sets the `MAP_DAX` mmap flag when memory-mapping the job file. The second option is `dax-pin` that allows to pin the memory-mapped file to the persistence cache by calling `mlock()`.

Chapter 5

Evaluation

To obtain a better understanding of the impact of our hybrid storage optimizations, we assess our implementation regarding performance, CPU overhead, and energy efficiency. For this thesis, we focus entirely on our writeback optimizations. For our energy efficiency evaluation, we only consider the active power consumption during write-heavy I/O workloads. As we are currently limited to our emulated hybrid storage device that uses a separate CXL memory expander for its persistent cache, we cannot faithfully model the idle power consumption of real-world hybrid storage devices.

First, we outline our evaluation setup and describe our testing methodology (§ 5.1). In order to isolate the impact of our writeback optimizations from other aspects of I/O, we evaluate a micro benchmark which stress tests synchronous writeback (§ 5.2). As this workload is hardly representative for real-world I/O, we evaluate a synthetic I/O workload that persists writes in a much coarser granularity in the next step (§ 5.3). Lastly, we evaluate the use of hybrid storage devices for persistence features in in-memory key-value datastores as one promising use case of hybrid storage devices (§ 5.4).

5.1 Methodology and Evaluation Setup

For evaluating the CPU and energy efficiency of our implementation on each workload, we adapt Werling et al.'s efficiency metrics for PM file systems [62]. As stated by Werling et al., their efficiency metrics are independent of the bandwidth [62]. Therefore, they allow us to decouple performance from efficiency considerations. Instead of using the authors' metrics for their intended purpose, namely comparisons between PM file systems, we use them to quantify the effectiveness of our `fsync()` optimizations for DAX pages.

Werling et al. define the *CPU cost* and the *energy cost* as follows [62]:

$$\mathtt{CPU\_cost} := \frac{\mathtt{CPU\_time}}{\mathtt{bytes\_written}}, \qquad \mathtt{energy\_cost} := \frac{\mathtt{energy\_used}}{\mathtt{bytes\_written}}$$

with the CPU cost being measured in seconds per gibibyte ($\mathrm{s/GiB}$) and the energy cost being measured in joules per gibibyte ($\mathrm{J/GiB}$). While these cost functions can be applied to our `fio` workload (§ 5.3) as is, they are not well-defined for the synchronous writeback micro benchmark (§ 5.2) and the key-value datastore benchmark (§ 5.4) since these benchmarks do not quantify progress in number of bytes written. For our synchronous writeback micro benchmark, we define the CPU cost and energy cost as

$$\mathtt{CPU\_cost}_{\mathtt{wb}} := \frac{\mathtt{CPU\_time}}{\mathtt{dirty\_bytes\_fsynced}},$$

$$\mathtt{energy\_cost}_{\mathtt{wb}} := \frac{\mathtt{energy\_used}}{\mathtt{dirty\_bytes\_fsynced}}$$

with `dirty_bytes_fsynced` including all bytes in a dirty page. Therefore, even if we write to a single byte in a page, the entire page will factor into the CPU and energy cost. We argue that this approach is more sensible as the OS will put the entire page under writeback, not just the bytes written. For the CPU and energy cost of the key-value datastore, we define them as

$$\mathtt{CPU\_cost}_{\mathtt{kv}} := \frac{\mathtt{CPU\_time}}{\mathtt{requests\_processed}}, \quad \mathtt{energy\_cost}_{\mathtt{kv}} := \frac{\mathtt{energy\_used}}{\mathtt{requests\_processed}}$$

For approximating the energy consumption of our benchmarks, we average the power consumption during the benchmark, subtract the average power consumption during idle, and multiply by the benchmark runtime. We measure the wall power for the entire system. Throughout our tests, we measure the power consumption multiple times per second and average all samples. By measuring the average power consumption over a time span of three minutes before running our tests, we recalibrate the idle power consumption to the current system.

Similar to Werling et al. [62], we obtain the CPU time consumed by our benchmarks from stats reported by Linux's CPU scheduler through `procfs` (`/proc/stat`). As I/O might cause activity on cores other than the ones running our benchmark, we consider all cores for this metric. Like in the case of our power measurements (e.g., interrupts of I/O devices), we subtract the CPU time consumed by an idle system from our measurements to isolate our benchmark from the rest of the system. The CPU time is defined as

$$\mathtt{t}_{\mathtt{cpu}} := (\mathtt{t}_{\mathtt{usr}} + \mathtt{t}_{\mathtt{sys}} + \mathtt{t}_{\mathtt{irq}} + \mathtt{t}_{\mathtt{softirq}})$$

with $\mathtt{t}_{\mathtt{usr}}$ as time in user mode, $\mathtt{t}_{\mathtt{sys}}$ as time in kernel mode, $\mathtt{t}_{\mathtt{irq}}$ as time servicing interrupts, and $\mathtt{t}_{\mathtt{softirq}}$ as time servicing softirqs.

One peculiarity of our approach to measuring CPU and energy efficiency for our `fio` benchmark, as well as the writeback micro benchmark, is that we do not capture the cost of writing out-of-sync pages to the backing storage. As our persistence-aware page cache will eventually write out-of-sync pages to the backing storage, the actual CPU and energy overhead should be larger. For the `fio` and writeback micro benchmark, we assume that they are long-running and that pages frequently synced remain in the page cache. Under these assumptions, we argue that the cost of the eventual writeback is negligible as its being amortized over the runtime. Our key-value store benchmark does not suffer from this problem as our DAX-aware Valkey implementation explicitly evicts pages from the page cache.

Component	**Specification**
Mainboard	Supermicro X13SEI-F
CPU	Intel® Xeon® Silver 4416+ (Sapphire Rapids)[*] 20 Cores @ 2 GHz
Main Memory	128 GiB DRAM 8 × 16 GiB DDR5 RDIMM @ 4800 MT/s
Memory Expander	16 GiB CXL-attached DRAM (CXL hard IP) 1 × 16 GiB DDR4 RDIMM @ 3200 MT/s
Storage	Samsung SSD 990 PRO 2 TB Samsung SSD 970 PRO 1 TB

[*] TurboBoost and Simultaneous Multi Threading (SMT) disabled

Table 5.1: Description of evaluation hardware.

As we do not have a storage device which fits our hybrid storage model available for our evaluation, we must emulate the hybrid storage device. During our testing, we encountered performance inconsistencies with our block device driver implementation described in Section 4.2. Instead of wrapping a block device in a virtual hybrid storage device, we register the *hybrid device* directly with the backing storage. We use a Samsung SSD 970 PRO 1 TB for the backing storage and 16 GiB CXL-attached DRAM for the persistent cache. A custom memory expander (CXL type 3 device) based on an Intel Agilex 7 I-Series FPGA [18] provides the CXL-attached DRAM for our device. The memory expander is connected via 16 PCIe 5 lanes to the host system and features 2 × 8 GiB DDR4 RDIMMs @ 3200 MT/s.

On our evaluation setup, we measure read latencies of approximately 350 ns using Intel's MLC [21] for the memory expander. This is 3.3 times higher than for local DRAM ($\sim$105 ns) on our evaluation platform. We compare the performance of our emulated hybrid storage device against the raw NVMe SSD that provides the backing storage for the emulated device. In this chapter, we denote the measurements using the raw NVMe SSD with “conventional I/O”. We denote measurements that use our hybrid storage optimizations with “hybrid storage”.

Table 5.1 describes our evaluation hardware. Our host platform supports CXL 1.1.We run Fedora 39 Server on our evaluation setup. For all tests, we use our modified Linux kernel described in Section 4.1. Benchmarks using `fio` use the modified mmap I/O engine described in Section 4.4.2. For benchmarking Valkey, we use our modifications described in Section 4.4.1. Apart from the measurements in Section 3.1 that do not require a file system, we use our modified ext4 for all other tests. Further, we disable journaling due to the limited scope of our implementation. In order to minimize noise on our measurements, we disable TurboBoost as well as SMT. In addition, we reserve physical cores by using Linux’s `isolcpus` kernel command-line option and pin benchmark tasks to the reserved set of cores. Lastly, we also disable THP for the entire system.

5.2 Synchronous Writeback DAX Optimization

In order to better understand the cost of synchronous writeback[1], we implement a micro benchmark that measures the latency of `fdatasync()` as a function of the amount of dirty pages d and the amount of virtual memory pages m. Here, we are only interested in the overhead of writeback for data pages as our optimizations do not affect file system metadata. For each latency measurement, our micro benchmark dirties the first d out of m pages by writing a single byte. After dirtying d pages, our benchmark calls `fdatasync()` and measures the latency of this call. For each value of d and m, we repeat this measurement at least 1000 times. Before taking measurements, we run 10 iterations as warm-up.

First, we assess performance for our writeback optimizations as a function of the mapping size. On the next page, Figure 5.1 shows the writeback bandwidth of dirty pages as well as the CPU and energy efficiency of the writeback. The “clear dirty” variant provides a performance projection for three-state dirty track and will be discussed in more detail below. While our current implementation provides up to $37.1\times$ higher average writeback bandwidth than conventional I/O ($7.2\,\mathrm{GiB/s}$ vs. $198.6\,\mathrm{MiB/s}$ at $2\,\mathrm{MiB}$ virtual memory), the writeback performance for hybrid storage unexpectedly drops off on large memory mappings. For memory mappings larger than $2\,\mathrm{GiB}$, the writeback bandwidth, as well as CPU and energy efficiency, fall behind conventional storage I/O. The writeback performance of conventional I/O, however, remains stable at a level of approximately $750\,\mathrm{MiB/s}$ for a virtual memory size larger $4\,\mathrm{MiB}$.

[1]not to be confused with writeback through the synchronous I/O interface.

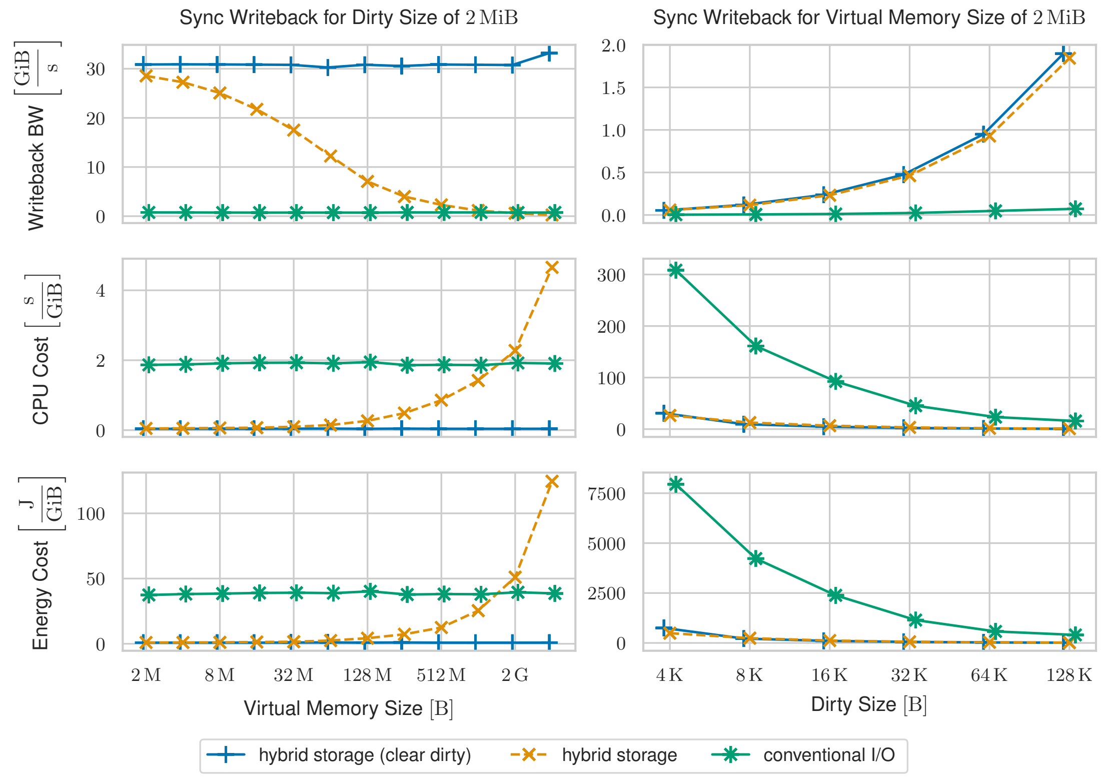

Figure 5.1: Synchronous writeback bandwidth, CPU efficiency, and energy efficiency for the synchronous writeback micro benchmark as a function of the virtual memory size of the memory-mapping (left column) and the amount of dirty pages (right column).

By profiling the writeback path for hybrid storage, we observe that the time for iterating over all dirty page cache entries dramatically increases with the memory mapping size. This is caused by the dirty state tracking of our implementation. As mentioned in Section 4.1.3, we currently do not differentiate between *out-of-sync* and *dirty* pages and leave DAX pages marked dirty during synchronous writeback. Our micro benchmark, however, pre-faults the entire memory mapping for write access, thus causing all pages in the mapping to be marked dirty initially. While the first call to `fsync()` during the warm-up phase clears this dirty state for conventional I/O, all pages remain dirty for our implementation. During synchronous writeback on hybrid storage, this causes the CPU overhead caused by iterating over all dirty pages to explode. This behavior can also be observed in the plot of the CPU and energy cost.

In order to predict the performance of a hybrid storage implementation that does not suffer from this limitation, we implement a writeback mode that clears the dirty mark on DAX pages during synchronous writeback instead of skipping these pages entirely. While this approach loses the dirty state required for writing DAX pages back to the underlying storage, it allows the dirty set to shrink during synchronous writeback, thus decreasing the CPU overhead required for iterating over dirty pages on consecutive calls to `fdatasync()`. Since an implementation that does use the *out-of-sync* state should also help in reducing the number of dirty pages during synchronous writeback, we argue that the writeback performance of the "clear dirty" variant offers a realistic prediction. Figure 5.1 shows that the modified dirty state tracking solves the performance shortcomings of our implementation. In Section 6.1, we discuss potential changes to our implementation that help in reducing the CPU overhead during synchronous writeback. For the rest of this thesis, we report measurements with and without this modified dirty state tracking.

Next, we assess how the dirty size impacts the writeback performance. While the writeback throughput significantly increases with the dirty size for hybrid storage (from 59.5 MiB/s to 1887.7 MiB/s), conventional I/O only shows a modest increase from 3.6 MiB/s to 187.3 MiB/s. At a dirty size of 2 MiB, hybrid storage reaches the approximately 30 GiB/s shown in the left column of Figure 5.1. Our measurements suggest that the overhead of writeback to storage dominates the cost of `fdatasync()` on conventional storage. As the memory mapping size of 2 MiB is relatively small, we do not see a significant difference between the dirty state tracking variants for hybrid storage.

5.3 Synthetic I/O Performance

While the micro benchmark in Section 5.2 shows a drastic increase in synchronous writeback performance for our hybrid storage optimizations, this workload is hardly representative for most I/O workloads. For our next test, we model an I/O-heavy workload that calls `fsync()` after writing a large block of memory. We use `fio` [3] with the mmap I/O engine for generating random writes on a single memory-mapped file. We configure `fio` to sync each written block using `fsync()`. Before running this test, we pre-fault the entire file into the page cache. As our current evaluation hardware does not support the hardware-assisted data movement between backing storage and persistent cache (see Section 3.2), we do not expect a measurable performance difference for page cache misses. Similar to our previous test, we measure the throughput as well as the CPU and energy efficiency. To simulate different loads, we vary the number of I/O threads and the block size.

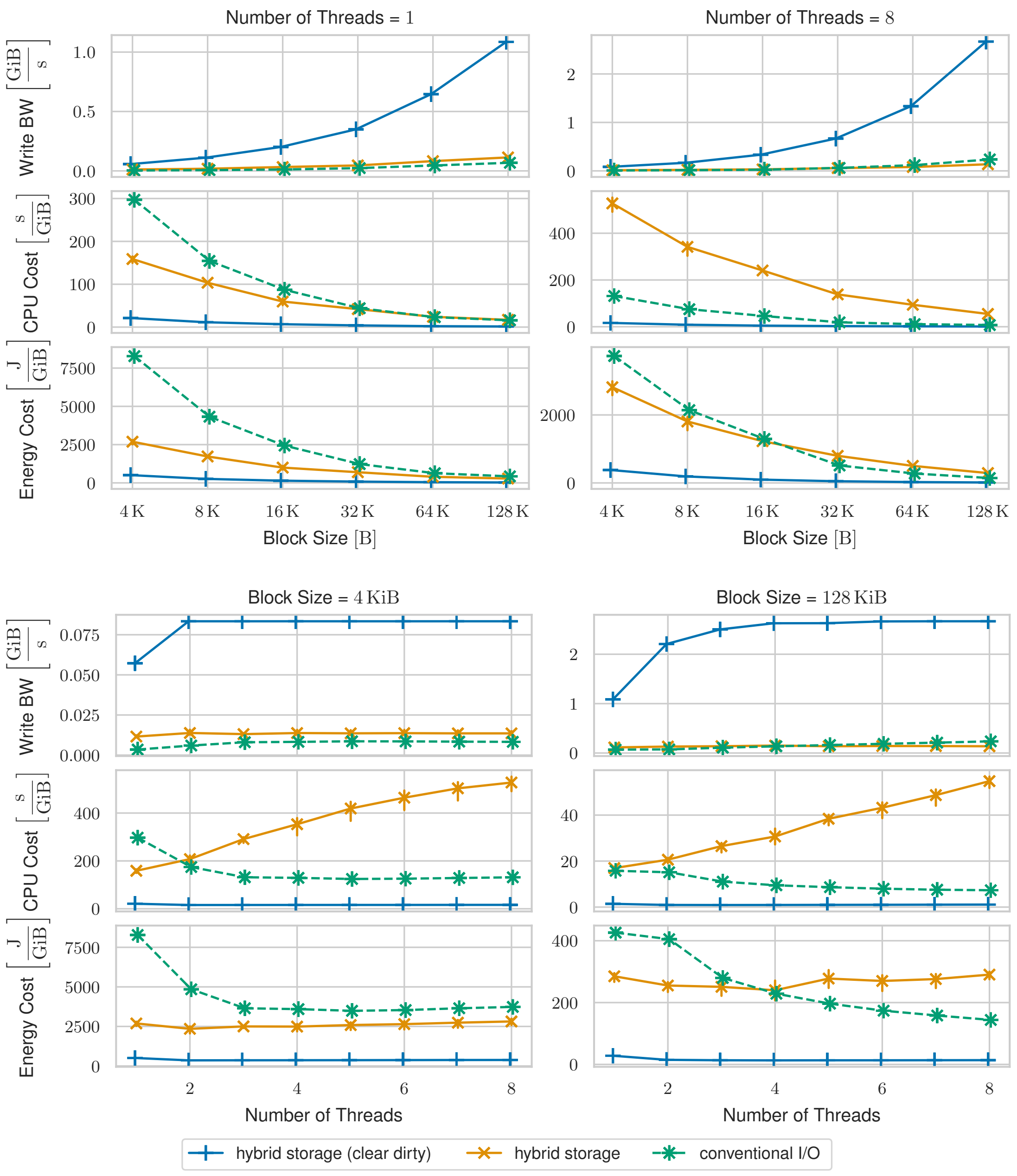

Figure 5.2: Write bandwidth, CPU efficiency, and energy efficiency for random write I/O workload that syncs after each written block. The left column shows measurements for a single I/O thread (top left) and a block size of 4 KiB (bottom left). The right column shows measurements for eight I/O threads (top right) and a block size of 128 KiB (bottom right).

Figure 5.2 shows the write bandwidth, CPU cost, and energy cost as a function of either the number of I/O threads or the block size. Similar to our previous test, the hybrid storage with modified dirty state tracking (i.e., "clear dirty") performs best in all metrics. At a block size of 128 KiB, the projected hybrid storage's write bandwidth is 11 to 30 times higher than for conventional storage while also causing only 6 % to 15 % of its CPU overhead and using 4 % to 10 % of its energy. The write bandwidth plateaus at 2.7 GiB/s. Given that our writeback micro benchmark showed that the maximum synchronous writeback throughput is just under 2 GiB, we suspect that `fsync()` overhead is limiting the write bandwidth in this test. The fact that the I/O performance does not increase with the number of I/O threads, even though we measured write bandwidths of approximately 22.6 GiB/s for our memory expander in Section 3.1, leads us to the hypothesis that lock contention on the `fsync()` path limits further scaling.

Without the modifications to the dirty state tracking, hybrid storage only provides between 0.6 to 3.3 times the write bandwidth of conventional storage. Similar to the previous test, our implementation suffers from the dirty set steadily increasing when using DAX mappings. This causes the synchronous writeback code, that iterates over all pages, to consume much more CPU time (up to 8.4 times). However, the impact of the growing dirty set is less severe on the energy consumption. We suspect that the CPU does not consume as much energy when contending for locks since this type of workload is not compute-heavy. As the dirty set increases with the ongoing I/O workload, the CPU overhead steadily grows. Consequently, the write bandwidth continuously drops over the course of the workload.

As the block size increases, the ratio of `fsync()` calls to bytes written decreases. Since each invocation of `fsync()` implies a switch between user and kernel space as well as a call into the file system, we can control this fixed cost through the block size. Due to the large gap in CPU and energy consumption for small block sizes between hybrid storage with modified dirty state tracking and conventional I/O, we argue that this fixed cost is negligible for conventional I/O. Here, the overhead of storage I/O through the asynchronous block interfaces dominates the I/O performance.

5.4 Real-World Application Performance

To showcase the potential benefit of hybrid storage devices and our approach to managing hybrid storage in the OS, we assess our implementation for the in-memory key-value datastore Valkey [57]. This last test focuses on our AOF mmap backend (§ 4.4.1) for improving the performance of Valkey's AOF persistence mode. We compare four different configurations for the AOF backend:

1. mmap backend with hybrid storage,
2. mmap backend with hybrid storage and modified dirty state tracking,
3. mmap backend with conventional I/O,
4. default AOF backend using `write()` with conventional I/O.

We pin Valkey's main thread and all background worker threads each to a reserved CPU core. In addition, we disable AOF rewrites. For benchmarking Valkey, we use the `valkey-benchmark` utility [58]. As only write operations cause writes to the AOF, we use workloads that only issue write commands to the server, namely the `SET`, `HSET`, and `INCR` tests included in `valkey-benchmark`. We found that all tested write benchmarks provide similar results. Therefore, we only report results for the `SET` benchmark. We configure `valkey-benchmark` to send one million requests per run. We repeat each test five times and average the results over all runs.

In order to achieve high throughput with Valkey, clients can use pipelining. Pipelining allows clients to send multiple commands to the Valkey server at once [60]. The server, on the other hand, can read multiple commands from the network socket with a single call to `read()`, process the entire batch of commands, and respond with a single call to `write()` [60]. This helps to reduce the number of context switches between user and kernel space and eliminates the time required waiting for a response before sending the next command [60]. When processing a batch of pipelined commands, Valkey issues a single AOF write for the entire batch [59].

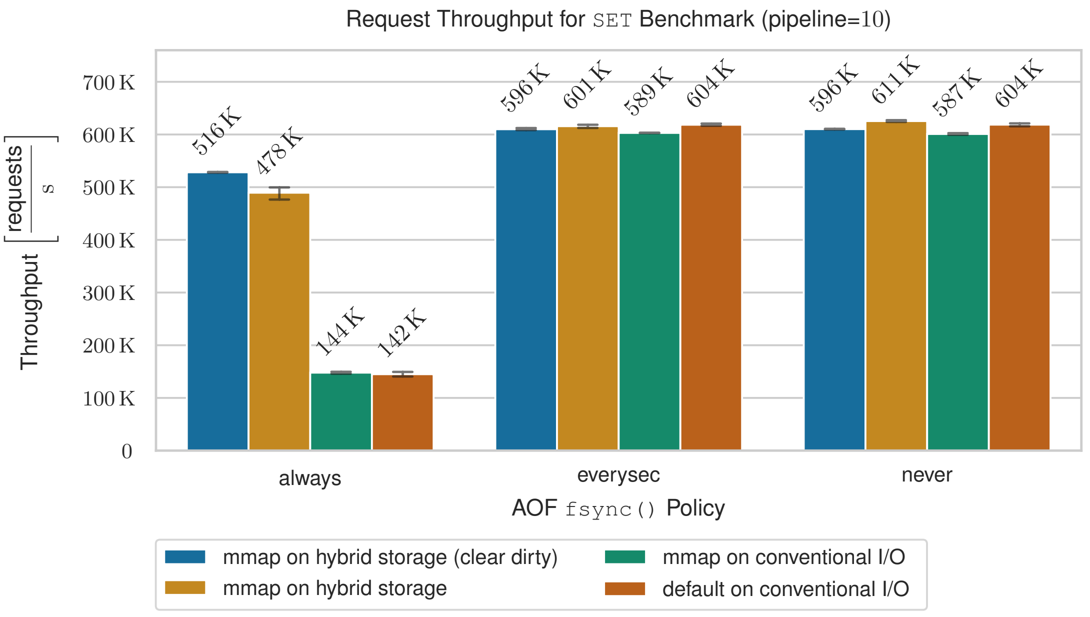

Figure 5.3: Request throughput on `SET` benchmark for all AOF `fsync()`. The *always* and *never* policies offer similar performance. Hybrid storage mitigates the impact of the *always* policy.

Figure 5.3 shows Valkey's request throughput on the `SET` benchmark for all AOF `fsync()` policies. When not using hybrid storage optimizations, our AOF mmap backend provides similar, albeit slightly lower, performance than the default AOF backend that uses `write` system calls for appending the AOF. While the AOF `fsync()` policy to periodically sync, namely *everysec*, and the policy to never sync show identical performance for all configurations, performance for the *always* policy takes a steep hit for conventional I/O (approx. $25\,\%$ of *never*'s throughput). Hybrid storage helps in reducing the performance gap between the `fsync()` policies. It provides up to 3.64 times higher throughput than conventional storage I/O ($87\,\%$ of *never* for the "clear dirty" variant).

Compared to our previous tests using `fio` and our writeback micro benchmark, our hybrid storage optimizations on Valkey are not susceptible to the lack of proper dirty state tracking in our implementation. Across all Valkey benchmarks, the modified dirty state tracking performs at most $10\,\%$ better than our current implementation. This behavior, however, is expected as our Valkey mmap backend keeps the number of AOF pages in the page cache small. For our parameters of `CHUNK_SIZE`, `ALLOC_WM`, and `PRE_ALLOC` (§ 4.4.1), we obtain an upper bound of $40\,\text{MiB}$ of buffered AOF contents in the page cache. Therefore, the dirty set size of the AOF file also has an upper bound of $40\,\text{MiB}$, which is small enough to not impact the `fsync()` performance too much.

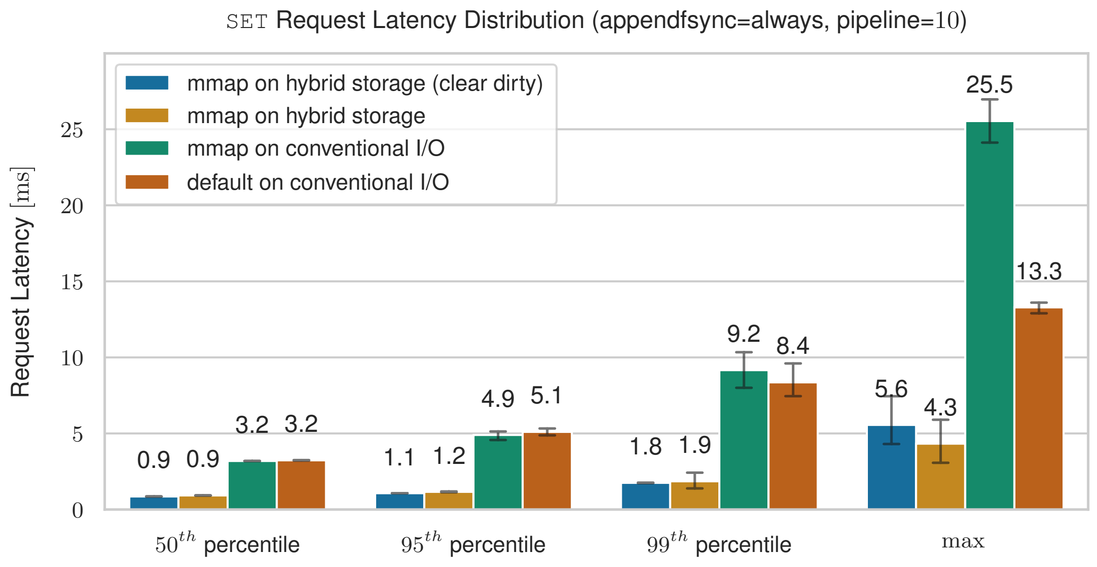

Figure 5.4: Request Latency Distribution for SET benchmark. Hybrid storage reduces request latencies significantly. The mmap AOF backend suffers from high worst case latencies.

Figure 5.4 shows the request latency distribution for the *always* AOF `fsync()` policy on the `SET` benchmark. As hybrid storage allows us to defer the expensive part of `fsync()` to a background worker thread, the request latency for hybrid storage is significantly shorter. For the 99^{th} percentile latency, hybrid storage achieves a latency reduction of $77.8\,\%$ compared to the default AOF backend on conventional storage. Further, hybrid storage not only reduces the latency but also provides more stable request latency.

One striking property of the request latency distribution is the worst-case latency for the mmap backend with conventional storage I/O. While the request latency is similar between the mmap backend and the default AOF backend for the 99^{th} percentile, there is a large difference of $12.2\,\mathrm{ms}$ for the maximum latency. For hybrid storage, the relative latency gap between the $99th$ percentile and the maximum latency is even more pronounced. We suspect that a long-running `fsync()` call during a rewrite in a background worker causes the main thread to also block on a `fsync()` call for flushing the AOF. This causes the `fsync()` call in the main thread to sync more than just the previously written AOF content. If our suspicion is correct, we should be able to mitigate this problem by using more fine-granular writeback in the main thread, e.g., use `msync()` on the AOF range that we want to persist. Investigating this possibility, however, remains as future work.

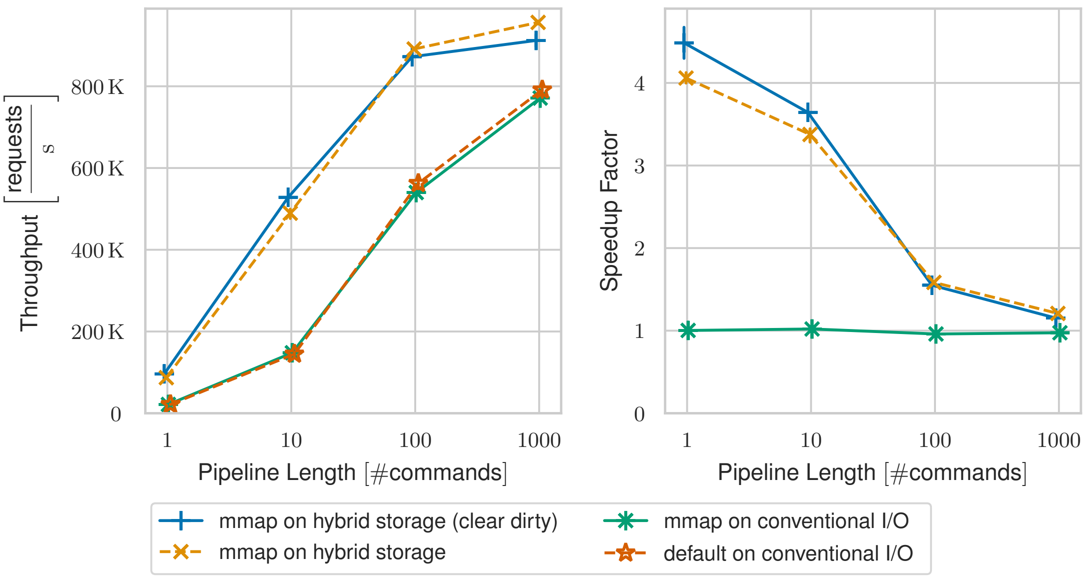

Figure 5.5: Throughput and speedup as a function of the pipeline length on `SET` benchmark. This test uses the *always* AOF `fsync()` policy. The speedup is relative to the default AOF backend on conventional storage. The request throughput speedup falls off for large pipeline lengths.

Figure 5.5 shows the request throughput and the resulting speedup relative to the default AOF backend with conventional storage as a function of the pipeline length. Here, we can observe that our hybrid storage optimization provides the most benefit for short pipeline lengths. This is expected because long command pipelines cause fewer AOF writes, and thus also fewer `fsync()` calls for the *always* policy. For 1000 pipelined `SET` commands, hybrid storage only provides a marginal speed up of $21\,\%$ over conventional I/O as command processing dominates the request. At a pipeline length of 1, hybrid storage reaches a speedup of up to 4.5 with modified dirty state tracking and 4.1 with the limited dirty state tracking of our current implementation.

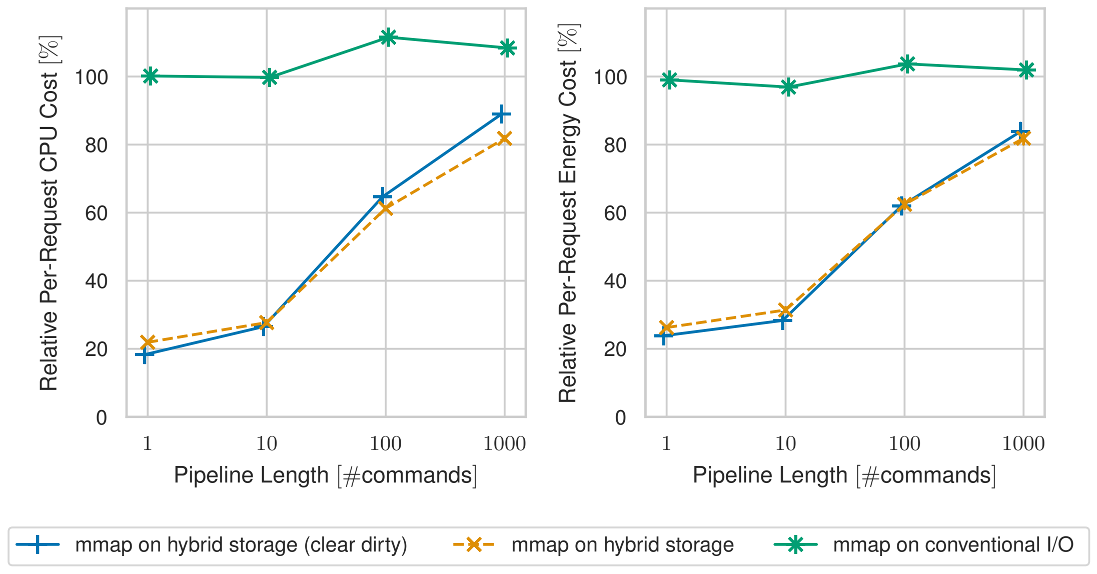

Figure 5.6: Per-`SET`-request CPU and energy cost relative to the default AOF backend on conventional storage. This test uses the *always* AOF `fsync()` policy. CPU and energy cost reductions fall off for large pipeline lengths.

As shown in Figure 5.6, the per-request CPU overhead and energy consumption relative to the default AOF backend on conventional I/O increase for a larger number of pipelined commands. Since our mmap backend keeps the number of dirty pages small, there is no large difference between the CPU and energy savings of our hybrid storage variants. Even though the “clear dirty” variant should cause no I/O on the backing storage at all, it does not provide better energy efficiency than our implementation which does access the backing storage during AOF resizing. This shows us that, in this workload, the energy consumption is not dominated by I/O caused by the AOF resize, but by the main thread.

Chapter 6

Discussion

In this chapter we discuss the results of our evaluation and outline future work. First, we discuss shortcomings of our dirty state tracking found during our evaluation (§ 6.1). Next, we examine data persistence guarantees of the persistence-aware page cache (§ 6.2). Finally, we outline cache resource management, transparent use of hybrid storage, and hardware-assisted data movement as subjects of future work (§ 6.3).

6.1 Dirty State Tracking

As shown in Chapter 5, our current implementation suffers from high CPU overhead on workloads where the number of dirty pages in the page cache grows too large. Like previously mentioned, this has to do with how our persistence-aware page cache handles the dirty state of buffered pages. In Section 3.4, we introduced the notion of *out-of-sync* pages. This dirty state describes a page whose up-to-date contents are only present in the persistent cache and not the backing storage. As our hybrid storage model assumes that the persistent cache guarantees all writes to persist a crash, the *out-of-sync* state is different from the *dirty* state in regard to persistence. Namely, all writes to a dirty page that were not synced to the backing storage, will be lost upon a crash.

Our current implementation does not differentiate between out-of-sync and dirty pages. Instead, it uses a generic dirty state that describes both types of pages whose up-to-date content is only present in the page cache. Since our optimizations to synchronous writeback aim to avoid writes to the backing storage, DAX pages under synchronous writeback must remain in this generic dirty state. The page cache cannot mark them clean as this would violate the invariant of the clean state, namely that the page contents in the page cache and on the backing storage are both up-to-date and in-sync.

When an application, like in the case of the `fio` benchmark (§ 5.3) or the writeback micro benchmark (§ 5.2), writes to a large file chunk and frequently calls `fsync()` between writes, the dirty set grows over time. While an unmodified writeback would clean all previously dirtied pages, thus reducing the number of pages that must be considered during the next writeback, our implementation must consider all dirty pages. Only in the case where the OS asynchronously writes dirty DAX pages back to storage or where the application explicitly manages the page cache, e.g., `MADV_DROPCACHE`, the dirty set shrinks.

To understand why having many dirty pages in the page cache causes performance problems, we need to take a closer look at how file systems implement `fsync()` and Linux's page cache internals. During synchronous writeback, file systems like ext4 tag all dirty pages buffered for the file under writeback with a `TOWRITE` mark. Like previously mentioned, this writeback tagging aims to avoid livelocks [26]. For implementing writeback, the file system iterates over all page cache entries in the file range that is under writeback and sets the `TOWRITE` mark if the page cache entry is dirty. Linux's XArray [65, 63] data structure that is, among other things, used for mapping file chunks to page cache entries [56], allows for efficiently iterating over dirty pages even when the page cache is large. For implementing our writeback optimizations, we skip DAX pages during writeback tagging. We have to iterate over all dirty pages in the file range regardless.

Under normal circumstances, the CPU does not spend much time in this loop, as typically only a small portion of buffered pages are marked dirty. Even if a large share of buffered pages is dirty, this increases the tagging cost just for the next writeback. Since our modifications lead to a steadily increasing dirty set size under certain conditions, the overhead of writeback page tagging becomes increasingly noticeable. This is further aggravated by having multiple threads doing writeback on the same file because the file system holds a spinlock that protects the XArray during writeback tagging. Combined with the already high cost for writeback tagging on our implementation, this leads to a highly contended spinlock that further wastes CPU time and thus also energy.

To reduce the cost of frequent writeback tagging on DAX-mapped files, our implementation must provide a mechanism for efficiently iterating over just the pages in the page cache that are either in the *dirty* or the *out-of-sync* dirty state. Here, we could potentially leverage Linux's XArrays similar to how iteration over dirty pages is implemented. Each entry in Xarrays contain a small number of bits, so called *search marks*, that associate the entry with a number of groups (one per bit) [65]. These search marks allow for efficient iteration over a specific group of entries in the Xarray [65]. Currently, XArrays offer three marks that users of the data structure can use [65]. The page cache, however, already uses all three search marks for tagging page cache entries as: **(1)** dirty, **(2)** under writeback, and **(3)** marked for writeback (i.e., `TOWRITE`) [56].

While we are not aware of a limitation that prevents additional search marks, more search marks will lead to increased memory consumption for all users of XArrays. On 64-bit systems, adding a fourth search mark increases the memory consumption of the XArray's internal tree nodes by approximately $1\,\%$. Given that XArrays have many users throughout the kernel, we speculate that even a small increase in memory consumption might have unforeseen regressions regarding the overall system performance and memory consumption. We intend to investigate this moving forward.

6.2 Data Persistence

Currently, our implementation limits persistence to the block device-level, meaning that all writes to DAX-mapped storage will reach the device and survive a crash[1]. What our implementation does not guarantee, however, is data persistence on the file system–level. Here, we define data persistence on the file system–level to guarantee the persistence of all writes to DAX-mapped files. While it might seem like persistence on the device-level implies persistence on the file system–level, this is not the case because the latter not only requires the data to persist but also file system metadata required for retrieving the data written.

This can, for example, lead to data loss on sparse files. When an application writes to a hole[2] in the sparse file by using a DAX mapping, the file system allocates a free block on the storage device and maps it into the process. While writes to this mapping will persist in that they reach the storage device, the metadata that maps this file range to the newly allocated block might not. If, however, metadata is lost after a crash, the written data is also lost from the viewpoint of the application.

The problem of achieving file system–level data persistence on storage exposed through a `load`/`store` interface is not unique to hybrid storage devices. Linux's DAX subsystem solves this problem through synchronous page faults [27] which are accessible through Linux's `MAP_SYNC` [38] mmap flag. When a page fault occurs due to a write on a memory mapping established with this flag, the DAX fault handler checks if there is any file system metadata that needs to be flushed in order to guarantee persistence of writes [27]. Should this be the case, the DAX fault handler does not insert the page into the page table but returns the faulting page with the additional information that the file system needs to flush metadata [27]. The file system's fault handler is then responsible for flushing required metadata, i.e., similar to how `fdatasync()` [35] works, for the faulting range, and finally inserting the DAX page writable into the page table [27]. This approach guarantees that a DAX page cannot be inserted writable into a `MAP_SYNC` mapping while there is unflushed metadata required for persisting the associated file range.

[1]Our emulated hybrid storage device does not guarantee persistence.

[2]A part of the sparse file without any backing storage blocks.

As of now, synchronous page faults are only supported for Linux's DAX subsystem. For implementing file system–level data persistence for hybrid storage, we intend to reuse Linux's synchronous page fault mechanism. This requires modifications to the page fault handling of supported file systems and the page cache itself. Analogous to the DAX subsystem, users would be required to use the `MAP_SYNC` flag in conjunction with our `MAP_DAX` flag to get the persistence guarantees of synchronous page faults.

6.3 Future Work

While our evaluation in Chapter 5 shows promising improvements of up to $37.1\times$ for synchronous writeback and up to $4.1\times$ more throughput for Valkey's AOF persistence, there are more facets to hybrid storage than what we can cover in this thesis. Next up, we discuss challenges and opportunities of our hybrid storage approach that we need to tackle moving forward.

Cache Resource Management

One of the most important challenges in using hybrid storage to its full potential is resource management. For our approach, we assume a hybrid storage device that offers only few gigabytes of persistent cache for terabytes of backing storage. As we cannot access the backing storage directly through `load` and `store` instructions but must go through the persistent cache, the OS has to carefully manage the cache capacity in order to optimize performance and provide fairness. The use of fine-granular DAX mappings and pinned DAX pages are two concepts used in our approach that aim to tackle this problem.

One area that we think needs further investigation is thrashing on the persistent cache. This occurs when the size of DAX-mapped file contents that are accessed in a short time span exceeds the persistent cache capacity. When thrashing occurs, we expect degraded I/O performance and increased CPU overhead as a significant amount of time will be spent with paging data between the persistent cache and the backing storage. Currently, our implementation does not offer any mechanism to counteract thrashing on the persistent cache. We leave it up to future work to explore thrashing mitigation techniques that fit our hybrid storage approach.

Managing the capacity of the persistent cache is only one piece of a holistic solution. The second dimension of resource management for hybrid storage that we need to consider is bandwidth management. As demonstrated in Section 3.1, the CPU efficiency for the synchronous `load`/`store` interface drops off when saturating the bandwidth in write-heavy workloads. Therefore, we argue that hybrid storage support in the kernel should strive to keep the bandwidth on the hybrid device's `load`/`store` interface below this critical level. Further, our measurements on our custom CXL memory expander show a decrease in bandwidth when too much pressure is applied. Prior work from Yang et al. [67] shows similar behavior on Optane PM. To what extent these observations on Optane PM translate to upcoming CXL hardware is currently unclear. If this behavior is also present in upcoming commercial CXL memory expanders and hybrid storage devices, we have an even stronger case for introducing bandwidth management to hybrid storage.

For managing bandwidth for hybrid storage, we plan to investigate per-process bandwidth monitoring. We argue that bandwidth management on process granularity helps in increasing fairness between tasks and allows the implementation of Quality of Service (QoS) features. This, for example, could be helpful for guaranteeing low tail latencies for performance-critical services like key-value stores.

Transparent Use of Hybrid Storage

Even though our writeback optimizations require little modifications in applications, they do require DAX mappings, which our current implementation does not employ transparently. To ease the adoption of hybrid storage, we intend to investigate transparent use of hybrid storage so that a wide range of unmodified applications can benefit. In the following, we want to outline one approach for transparently using our writeback optimizations.

First, the OS needs to identify file ranges that can benefit from fast synchronous writeback. Here, we expect all sufficiently small file ranges that show heavy use of `fsync()` to be suitable candidates. For identifying such file ranges, the OS accounts calls to `fsync()` on a coarse granularity (e.g., 2 MiB). In order to maintain acceptable memory overhead, we propose to only engage this accounting for files that exceed a static limit of `fsync()` calls in a predefined time frame. As soon as a file chunk exceeds a static limit of `fsync()` calls, the OS checks if there is enough free persistent cache capacity available. If this is the case, the OS engages the DAX upgrade mechanism proposed in Section 3.4. In addition, the OS must ensure that future page cache allocations for this file range allocate DAX pages. In order to revert this DAX upgrade when the hybrid device runs out of free DAX pages, the OS must track transparently upgraded file ranges. Before evicting DAX pages of applications that make explicit use of our hybrid storage API, the OS reclaims DAX pages from file ranges that use DAX transparently.

Hardware-Assisted Data Movement

As discussed in Section 3.2, hardware-assisted data movement between the persistent cache and the backing storage is one crucial feature of our hybrid storage model that distinguishes it from using a PM product in combination with a high-end NVMe SSD. Our evaluation, however, is limited to emulated hybrid storage that does not offer this feature. To showcase the full potential of our approach to hybrid storage, the access to a fully-capable hybrid storage device is essential. For future work, we set out to investigate the use of memory expansion hybrids, like Samsung's CMM-H [48] or IBM's Hybrid Memory Subsystem (HMS) [24], for our approach. Additionally, we intend to explore the hardware design space of hybrid storage with a FPGA-based prototype.

Chapter 7

Conclusion

Emerging hybrid storage devices path the way for bringing cost-effective and byte-addressable storage to the masses. By combining a persistent cache accessed through `load`/`store` semantics with cost-effective Flash storage, they enable low-latency storage I/O with strong persistence guarantees, something that was previously only attainable through expensive PM technologies such as Intel Optane.

In this thesis, we described our OS-centric approach for supporting emerging hybrid storage devices. Based on our analysis of both storage interfaces regarding performance, CPU overhead, and energy efficiency, we proposed a high-level hybrid storage model that enables persistent cache management through the OS. Further, we have shown that this device model cannot be supported by Linux's DAX subsystem without making significant compromises. To address the problem of resource management for hybrid storage in Linux, we introduced the persistence-aware page cache that focuses on fine-granular control over mappings of cache memory. The DAX and resource management features of our design are exposed to user space through a slim extension to the POSIX API. Applications only require minimal changes to benefit from direct access to storage and fast synchronous writeback when already using memory-mapped I/O.

We evaluated our Linux-based prototype on writeback-heavy micro benchmarks and a modified build of the key-value datastore Valkey with AOF persistence. Due to the lack of available hybrid storage hardware, we emulated the device using CXL-attached DRAM and a NVMe SSD. For `fsync()`, we achieved up to 37 $\times$ higher throughput. Using our new AOF mmap backend, we translated the speedup on `fsync()` into $4.1\times$ higher request throughput on Valkey for non-pipelined write requests. In addition, tail latencies (99^{th} percentile) are reduced by up to $77.8\,\%$, and CPU and energy overhead by $78\,\%$ and $74\,\%$. While our current implementation suffers from high CPU overhead in workloads with large working sets, we showcased the potential of improved dirty state tracking to solve this problem and further improve performance.

List of Acronyms

Notation	Description
AOF	Append Only File.
BAR	Base Address Register.
CXL	Compute Express Link.
DAX	Direct Access.
DMA	Direct Memory Access.
DRAM	Dynamic Random Access Memory.
eADR	extended Asynchronous DRAM Refresh.
EOF	End of File.
FPGA	Field Programmable Gate Array.
GPF	Global Persistent Flush.
ISA	Instruction Set Architecture.
LoC	Lines of Code.
MMIO	Memory-Mapped I/O.
MRAM	Magnetoresistive Random Access Memory.
NUMA	Non-Uniform Memory Access.
NVMe	Non-volatile Memory Express.
OS	Operating System.
PCIe	Peripheral Component Interconnect Express.
PM	Persistent Memory.
QoS	Quality of Service.
SMT	Simultaneous Multi Threading.
SSD	Solid State Drive.
TCO	Total Cost of Ownership.
THP	Transparent Huge Page.
VMA	Virtual Memory Area.

Bibliography

[1] A. Abulila, V. S. Mailthody, Z. Qureshi, J. Huang, N. S. Kim, J. Xiong, and W.-m. Hwu. "FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy." In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, Apr. 4, 2019, pp. 971–985.

[2] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.10. Arpaci-Dusseau Books, Nov. 2023.

[3] J. Axboe. *Flexible I/O Tester*. Version 3.37. 2022. URL: https://github.com/axboe/fio.

[4] J. Axboe. *Per-Bdi Writeback Flusher Threads*. Patch Series, v20. Linux Kernel Mailing List (LKML). Sept. 11, 2009. URL: https://lore.kernel.org/all/1252654450-25721-1-git-send-email-jens.axboe@oracle.com/.

[5] M. C. Baca. *CXL Picks Up Steam In Data Centers*. Semiconductor Engineering. Jan. 26, 2023. URL: https://semiengineering.com/cxl-picks-up-steam-in-data-centers/ (visited on July 17, 2024).

[6] D.-H. Bae, I. Jo, Y. A. Choi, J.-Y. Hwang, S. Cho, D.-G. Lee, and J. Jeong. "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)*. 2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA). Los Angeles, CA: IEEE, June 2018, pp. 425–438.

[7] Compute Express Link Consortium, Inc. *Compute Express Link (CXL) Specification 3.0*. Version 1.0. 2022.

[8] Compute Express Link Consortium, Inc. *Compute Express Link Consortium, Inc. and CCIX Consortium, Inc. Announce Agreement for Consortium to Receive CCIX Consortium Specifications and Other CCIX Consortium Assets*. Press Release. Aug. 6, 2023. URL: https://computeexpresslink.org/blog/compute-express-link-consortium-inc-and-ccix-consortium-

inc-announce-agreement-for-consortium-to-receive-ccix-consortium-specifications-and-other-ccix-consortium-assets-1052/ (visited on July 17, 2024).

[9] Compute Express Link Consortium, Inc. *CXL Consortium and OpenCAPI Consortium Sign Letter of Intent to Transfer OpenCAPI Specifications to CXL*. Press Release. Aug. 1, 2022. URL: https://computeexpresslink.org/wp-content/uploads/2024/01/OCC_CXL-Announcement_FINAL.pdf (visited on July 17, 2024).

[10] Compute Express Link Consortium, Inc. *CXL Consortium Announces Compute Express Link 3.1 Specification Release*. Press Release. Nov. 14, 2023. URL: https://computeexpresslink.org/wp-content/uploads/2024/01/CXL_3.1-Specification-Release_FINAL.pdf (visited on July 17, 2024).

[11] Compute Express Link Consortium, Inc. *CXL Consortium Signs Agreement with Gen-Z Consortium to Accept Transfer of Gen-Z Specifications and Assets*. Press Release. Feb. 10, 2022. URL: https://computeexpresslink.org/wp-content/uploads/2024/01/CXL_GenZ-Agreement-Release_v3.pdf (visited on July 17, 2024).

[12] J. Corbet. “A Kernel without Buffer Heads.” In: *LWN.net* (May 1, 2023). URL: https://lwn.net/Articles/930173/ (visited on Apr. 3, 2024).

[13] D. Das Sharma, R. Blankenship, and D. Berger. “An Introduction to the Compute Express Link (CXL) Interconnect.” In: *ACM Computing Surveys* 56.11 (July 2024).

[14] P. Esmaili-Dokht, F. Sgherzi, V. S. Girelli, I. Boixaderas, M. Carmin, A. Momeni, A. Armejach, E. Mercadal, G. Llort, P. Radojkovic, M. Moreto, J. Gimenez, X. Martorell, E. Ayguade, J. Labarta, E. Confalonieri, R. Dubey, and J. Adlard. *A Mess of Memory System Benchmarking, Simulation and Application Profiling*. Version 1. May 16, 2024. arXiv: 2405.10170 [cs]. Pre-published.

[15] B. Gervasi and S. Chang. *NVMe Over CXL™ Defines Memory Class Storage to Improve System Performance*. White Paper. 2024.

[16] J. Handy. *Intel’s Optane DIMM Price Model - The Memory Guy Blog*. May 31, 2019. URL: https://TheMemoryGuy.com/intels-optane-dimm-price-model/ (visited on July 14, 2024).

[17] IEEE and The Open Group. *The Open Group Base Specifications Issue 7, 2018 edition*. Jan. 2018. URL: https://pubs.opengroup.org/onlinepubs/9699919799/mindex.html.

[18] Intel Corporation. *Agilex™ 7 FPGA and SoC FPGA I-Series*. URL: https://www.intel.com/content/www/us/en/products/details/fpga/agilex/7/i-series.html (visited on July 4, 2024).

[19] Intel Corporation. *daxctl*. Version 78. URL: https://github.com/pmem/ndctl.

[20] Intel Corporation. *eADR: New Opportunities for Persistent Memory Applications*. Jan. 15, 2021. URL: https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html (visited on July 16, 2024).

[21] Intel Corporation. *Intel® Memory Latency Checker*. Version 3.11a. URL: https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

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

[23] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. *Basic Performance Measurements of the Intel Optane DC Persistent Memory Module*. 2019. arXiv: 1903.05714 [cs.DC].

[24] D. Jamsek and A. McPadden. “An Innovative Persistent Memory Solution with Today’s Memory.” Conference session. Presented at SC’19. Nov. 20, 2019. URL: https://sc19.supercomputing.org/presentation/index-id=exforum148&sess=sess373.html.

[25] M. Jung. “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*. HotStorage ’22. New York, NY, USA: Association for Computing Machinery, June 27, 2022, pp. 45–51.

[26] J. Kara. *Implement Writeback Livelock Avoidance Using Page Tagging*. Patch. Linux Kernel Mailing List (LKML). Feb. 12, 2010. URL: https://lore.kernel.org/all/1265929584-5080-3-git-send-email-jack@suse.cz/.

[27] J. Kara. *Synchronous Page Faults*. Patch Series, v6. Linux Kernel Mailing List (LKML). Nov. 1, 2017. URL: https://lore.kernel.org/all/20171101153648.30166-1-jack@suse.cz/.

[28] J. Kim, Y. J. Soh, J. Izraelevitz, J. Zhao, and S. Swanson. “SubZero: Zero-Copy IO for Persistent Main Memory File Systems.” In: *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems*. APSys ’20. New York, NY, USA: Association for Computing Machinery, Aug. 24, 2020, pp. 1–8.

[29] C. Lameter and M. Kim. “Page Migration.” In: *The Linux Kernel Documentation*. v6.6.0. URL: `https://www.kernel.org/doc/html/v6.6/mm/page_migration.html`.

[30] R. Landley. “Red-Black Trees (Rbtree) in Linux.” In: *The Linux Kernel Documentation*. v6.6.0. URL: `https://docs.kernel.org/6.6/core-api/rbtree.html`.

[31] G. Lee, S. Shin, W. Song, T. J. Ham, J. W. Lee, and J. Jeong. “Asynchronous I/O Stack: A Low-latency Kernel I/O Stack for Ultra-Low Latency SSDs.” In: 2019 USENIX Annual Technical Conference (USENIX ATC 19). 2019, pp. 603–616.

[32] Linux kernel contributors. “Direct Access for Files.” In: *The Linux Kernel Documentation*. v6.6.0. URL: `https://docs.kernel.org/6.6/filesystems/dax.html`.

[33] Linux kernel contributors. “VFS iomap Documentation.” In: *The Linux Kernel Documentation*. next-20240621. June 21, 2024. URL: `https://www.kernel.org/doc/html/next/filesystems/iomap/index.html` (visited on June 24, 2024).

[34] Linux man-pages project. *fallocate(2) — Linux System Call Manual*. Version 6.8. May 2, 2024.

[35] Linux man-pages project. *fsync(2) — Linux System Call Manual*. Version 6.8. May 2, 2024.

[36] Linux man-pages project. *madvise(2) — Linux System Call Manual*. Version 6.8. May 2, 2024.

[37] Linux man-pages project. *mlock(2) — Linux System Call Manual*. Version 6.8. May 2, 2024.

[38] Linux man-pages project. *mmap(2) — Linux System Call Manual*. Version 6.8. May 2, 2024.

[39] Linux man-pages project. *msync(2) — Linux System Call Manual*. Version 6.8. May 2, 2024.

[40] Linux man-pages project. *open(2) — Linux System Call Manual*. Version 6.8. May 2, 2024.

[41] Linux man-pages project. *sync_file_range(2) — Linux System Call Manual*. Version 6.8. May 2, 2024.

[42] Microsoft. *Win32 File Mapping Documentation*. URL: https://learn.microsoft.com/en-us/windows/win32/memory/file-mapping (visited on June 3, 2024).

[43] NVM Express, Inc. *NVM Express® Base Specification, Revision 2.0c*. 2022.

[44] PCI-SIG®. *PCI Express® Base Specification Revision 4.0 Version 1.0*. Version 1.0. 2017.

[45] *PCPartPicker*. URL: https://pcpartpicker.com (visited on June 12, 2024).

[46] R. van Riel. "Page Replacement in Linux 2.4 Memory Management." In: 2001 USENIX Annual Technical Conference (USENIX ATC 01). 2001.

[47] C. Robinson. *Kioxia CXL and BiCS Flash SSD Shown at FMS 2023*. ServeTheHome. Aug. 10, 2023. URL: https://www.servethehome.com/kioxia-cxl-and-bics-flash-ssd-shown-at-fms-2023/ (visited on July 14, 2024).

[48] Samsung. *CXL Memory Module Hybrid (CMM-H)*. 2023. URL: https://samsungmsl.com/cmmh/ (visited on Mar. 23, 2024).

[49] Samsung. *Samsung CXL Solutions – CMM-H*. July 9, 2024. URL: https://semiconductor.samsung.com/news-events/tech-blog/samsung-cxl-solutions-cmm-h (visited on July 18, 2024).

[50] Samsung. *Ultra-Low Latency with Samsung Z-NAND SSD*. 2017. URL: https://download.semiconductor.samsung.com/resources/brochure/Ultra-Low%20Latency%20with%20Samsung%20Z-NAND%20SSD.pdf (visited on July 19, 2024).

[51] S. Sanfilippo. *Redis*. URL: https://github.com/redis/redis.

[52] C. Siebenmann. *What the Unified Buffer Cache Is Unifying*. July 2, 2007. URL: https://utcc.utoronto.ca/~cks/space/blog/unix/UnifiedBufferCache (visited on July 18, 2024).

[53] J. Stuecheli. "Open Coherent Accelerator Processor Interface (OpenCAPI) for Advanced Storage." Presented at SNIA Developer Conference 2018. Sept. 26, 2018. URL: https://www.snia.org/sites/default/files/SDC/2018/presentations/General_Session/Jeff_Stuechelli_OpenCAPI.pdf (visited on July 18, 2024).

[54] Y. Sun, Y. Yuan, Z. Yu, R. Kuper, C. Song, J. Huang, H. Ji, S. Agarwal, J. Lou, I. Jeong, R. Wang, J. H. Ahn, T. Xu, and N. S. Kim. "Demystifying CXL Memory with Genuine CXL-ready Systems and Devices." In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. Micro '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 105–121.

[55] T. Talpey. “Persistent Memory in Windows Server 2016.” Persistent Memory Summit 2017. Presented at Persistent Memory Summit 2017. 2017. URL: https://www.snia.org/sites/default/files/PM-Summit/2017/presentations/Tom_Talpey_Persistent_Memory_in_Windows_Server_2016.pdf.

[56] L. Torvalds. *Linux Kernel*. Version 6.6. URL: https://www.kernel.org/.

[57] Valkey contributors. *Valkey*. Version 7.2.5. 2024. URL: https://valkey.io/.

[58] Valkey contributors. *Valkey Benchmark*. URL: https://valkey.io/docs/topics/benchmark/ (visited on July 7, 2024).

[59] Valkey contributors. *Valkey Persistence*. URL: https://valkey.io/docs/topics/persistence/ (visited on June 28, 2024).

[60] Valkey contributors. *Valkey Pipelining*. URL: https://valkey.io/docs/topics/pipelining/ (visited on July 7, 2024).

[61] K. Vättö. *Power Loss Protection: How SSDs Are Protecting Data Integrity*. White Paper. Mar. 22, 2016. URL: https://insights.samsung.com/2016/03/22/power-loss-protection-how-ssds-are-protecting-data-integrity-white-paper/ (visited on July 14, 2024).

[62] L. Werling, Y. Khalil, P. Maucher, T. Gröninger, and F. Bellosa. “Analyzing and Improving CPU and Energy Efficiency of PM File Systems.” In: *Proceedings of the 1st Workshop on Disruptive Memory Systems*. DIMES ’23. New York, NY, USA: Association for Computing Machinery, Oct. 2023, pp. 31–37.

[63] M. Wilcox. *Introducing the eXtensible Array (Xarray)*. Patch Series. Linux Kernel Mailing List (LKML). Feb. 28, 2017. URL: https://lore.kernel.org/all/20170228181343.16588-1-willy@infradead.org/.

[64] M. Wilcox. *Memory Folios*. Patch Series. Linux Kernel Mailing List (LKML). July 15, 2021. URL: https://lore.kernel.org/all/20210715033704.692967-1-willy@infradead.org/.

[65] M. Wilcox. “XArray.” In: *The Linux Kernel Documentation*. v6.6.0. URL: https://docs.kernel.org/6.6/core-api/xarray.html.

[66] M. Wilcox and W. Kucharski. *MM Folio Patches for 5.18*. Patch Series. Linux Kernel Mailing List (LKML). Feb. 4, 2022. URL: https://lore.kernel.org/linux-mm/20220204195852.1751729-1-willy@infradead.org/.

[67] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. “An Empirical Guide to the Behavior and Use of Scalable Persistent Memory.” In: 18th USENIX Conference on File and Storage Technologies (FAST 20). 2020, pp. 169–182.