Karlsruhe Institute of Technology

Transparent DAX Upgrades for CXL-based hybrid SSDs

Bachelor's Thesis
submitted by

Daniel John Dietzler

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Wolfgang Karl
Advisor:	Daniel Habicht, M.Sc.

June 13th, 2025 – October 13th, 2025

Abstract

Non-volatile memory is becoming of growing importance. There is great need for very fast and low latency, yet persistent memory. As flash storage cannot keep up with that, there has been rising research interest in Persistent Memory (PM). However, traditional PM is expensive and complex. Hybrid SSDs on the other hand combine traditional volatile memory with persistent flash storage. They are able to support both a block-granular and a fine-granular interface using Compute Express Link (CXL). The fine-granular interface allows the Operating System (OS) to map pages into the hybrid SSD's memory, which is persistent as it is backed by flash storage. This direct mapping allows bypassing the OS page cache and makes writeback superfluous as the data is already in a memory with persistence guarantees.

While there is already research on extending the concept of Direct Access (DAX) to make it work well with hybrid SSDs, up until now that required application support. This decreases usability and slows down adoption, as major companies tend to be slow with implementing new features, especially if there is no production hardware yet. On the other hand, sales of new hardware are expected to be low while there is not great use for them.

We introduce *Transparent DAX Upgrades*, a way to automatically upgrade relevant file ranges to hybrid SSDs. With our work we lift this requirement of needing application support by tracking file ranges and automatically upgrading those that we consider to have strong persistence requirements. Particularly, this means file ranges that are often written back due to system calls such as `fsync`. An upgrade hereby refers to migrating pages from the OS page cache to the memory on a hybrid SSD.

In our testing we measure a speedup of up to $24.47\times$ for RocksDB. We further benchmark Valkey and PostgreSQL and observe $7.29\times$ and $2.53\times$ speedups, respectively. By using our kernel patch, users can get large performance improvements for their applications without needing to configure anything.

Contents

Chapter 1

Introduction

With surging demand for High-Performance Computing, memory and storage requirements also increase [5]. There is not only a growing need for more memory, but also for *non-volatile memory*. Non-volatile memory combines the low access latencies of memory with the persistence properties of flash storage. Once written, data is retained, even in case of a power loss.

An illustrative use case for low latency persistent memory are database applications. Their access pattern typically consists of many writes immediately followed by `fsync` calls to ensure the data is persisted [26]. Without persistent, non-volatile memory, data would need to be flushed to the slow NAND flash storage on every writeback. Compared to flash storage, memory such as DRAM has significantly lower latencies and is byte addressable [5]. This has caused rising research interest in Persistent Memory (PM), a type of non-volatile memory [46, 36, 47]. Predominantly, Intel Optane quickly became popular, yet it reached end-of-life in 2024, with the most recent models being launched in 2023 [21]. PM is expensive compared to NAND, takes up a DIMM slot, requires extra NAND flash storage or more expensive special RAM, and is complex [5]. To address this, a new concept of *Hybrid SSDs* has emerged [5, 19, 55]. This special type of SSD included both a DRAM module and NAND flash storage [5, 55]. The DRAM is used as a buffer to enable fine-granular access [5, 55]. However, this imposes a new issue of needing to access both the fine-granular and block-level interface. This is made possible through a new standard called CXL.

Intel started working on a new protocol, built on the physical layer of PCIe, called Intel Accelerator Link (IAL). They open-sourced it as Compute Express Link (CXL) 1.0 in 2019 [8]. CXL provides asymmetric coherency; the host orchestrates coherency for all devices, so called *end points* [8]. Furthermore, CXL also promises backwards compatibility and vendor-independence, similar to PCIe which it shares the physical link layer with [8]. CXL exposes three interfaces; `CXL.io`, `CXL.mem`, and `CXL.cache` [7]. Devices supporting variations of these interfaces are differentiated into three classes; Type 1, Type 2, and Type 3 devices. Type 3 devices provide the `CXL.io` and `CXL.mem` interfaces which allow for both block- as well as fine-granular access. Hybrid SSDs are such type 3 devices.

Besides being able to access both interfaces of a hybrid SSD, it is also necessary to be able to memory map pages into the hybrid SSD's DRAM in order to properly use them. Traditional Direct Access (DAX) allows for direct access to storage, bypassing the host's caching layers [9]. DAX requires file system support and has been a feature of Linux for years. However, it only supports file-granular mappings and expects uniform low-latency memory access. Both are downsides that result in traditional DAX not being feasible for hybrid SSDs.

To solve these problems, Habicht et al. have introduced new interfaces to interact with CXL-based hybrid SSDs on an Operating System (OS) level, extending the idea of DAX [18]. Their DAX mappings can cope with Non-Uniform Memory Access (NUMA) and support page-granular mappings. However, while having implemented DAX mappings for hybrid SSDs [19], the current state of their work requires application support. Applications need to explicitly request DAX to access the hybrid SSD's persistent DRAM if they want persistent memory [18]. This thesis aims to fill that gap by introducing *Transparent DAX Upgrades for CXL-based Hybrid SSDs*. We propose a new way of tracking persistence requirements of file ranges and automatically upgrading file ranges with high persistence requirements. This enables the use of hybrid SSDs without needing to modify an application to explicitly request those DAX mappings. In our evaluation we are able to show an up to $25\times$ speedup, simply by using our kernel modifications and a hybrid SSD.

Chapter 2 discusses the underlying technologies of this work, specifically CXL, hybrid SSDs, traditional DAX, and hybrid DAX mappings. Then, we present alternative projects and works related to hybrid SSDs and PM in chapter 3. In chapter 4 we outline our approach, focusing on the tracking and policy we employed. Based on that, chapter 5 describes the specific implementation of transparent DAX upgrades. Afterwards, chapter 6 covers the results of our benchmark suite. We will also elaborate on the hardware our benchmarks ran on. After providing ideas for future work in chapter 7, we put our project into context in the conclusion in chapter 8.

Chapter 2

Background

2.1 CXL

Compute Express Link (CXL) [7] was introduced in 2019. In 2021, Intel and AMD announced CXL support for their upcoming enterprise processors [48, 31]. With Intel's release of their new lineup in 2023, both players had production CPUs supporting CXL [3]. By now, many major companies support and contribute to CXL, such as Intel and Samsung [20].

CXL provides three different modes:

- `CXL.io`
- `CXL.cache`
- `CXL.mem`

The block interface `CXL.io` is used for tasks such as device discovery and configuration, but can also be used for Direct Memory Access (DMA) [37, 7].
`CXL.cache` allows devices to access host memory coherently [7]. Specifically, a device can make a direct request to the host (D2H) targeting memory. The host can send *snoops* to the device (H2D) to ensure coherency [7]. The idea of `CXL.cache` is to coherently access the host's memory from a CXL device. Since we are interested in using a CXL device as a memory expander for the host, `CXL.cache` will not be relevant for our analysis. `CXL.mem` works the other way around as `CXL.cache`: `CXL.mem` enables the host to access memory on a CXL device as if it were the host's memory [37]. `CXL.mem` offers a load/store interface, similar to the one that is used for system memory.

Memory on a CXL device that is exposed through `CXL.mem` is called Host managed Device Memory (HDM) [8]. HDM can be further divided into host-only coherent (HDM-H) and device-managed coherent (HDM-D) memory [8]. In the following we will focus on HDM-H.

Habicht et al. found that their CXL memory device using a FPGA has $\sim 3\times$ the latency compared to DRAM in a SODIMM slot [19]. This is a similar latency to accessing memory of a remote socket [8]. As such there is a latency difference between different memory pages, however. Non-Uniform Memory Access (NUMA) allows OSs to deal with that difference [8].

2.1.1 Device Classes

A CXL device is categorized into a class based on the supported interfaces. Type 1 devices are accelerators. They implement `CXL.io` and `CXL.cache` [7]. Using the cache interface, they can coherently access host memory and do calculations on the working sets in there.

Type 2 devices implement all three protocols; `CXL.io`, `CXL.cache`, and `CXL.mem` [7]. That is, the CXL device offers HDM but can also coherently access the host's memory. Additionally, the HDM is also kept coherent with the host [7]. One example for devices that may benefit from CXL are general purpose GPUs [7]. GPUs also have onboard memory, but the cards cannot access the host memory and the onboard memory is not kept coherent with the rest of the system. There is a lot of communication overhead and data exchange to keep data in sync [7]. As a CXL type 2 device with all three CXL modes available, most of these issues could be improved or even resolved.

Lastly, type 3 devices provide `CXL.io` and `CXL.mem` [7]. These devices are memory expanders or Persistent Memory (PM). A type 3 device exposes the on-device memory to the host via the `CXL.mem` interface [7]. As such, the host has full access to it similar to any other Non-Uniform Memory Access (NUMA) node. Our work focuses on type 3 devices. We want to upgrade file ranges with strong persistence requirements to CXL PM.

2.1.2 History

CXL 2.0 introduces resource pooling [8]. This means devices can be shared between multiple hosts. Specifically, memory expanders can also be shared, allowing memory sharing across hosts [8]. Das Sharma, Blankenship, and Berger [8] argue that this can help mitigate the problem of *resource stranding*. Resource stranding describes unused resources (in this work, we care about memory) on a host for a given task. If memory cannot be shared, every host needs to have as much memory as the most intensive job requires. This results in overprovisioning and unused memory.

Google engineers, however, argue that CXL memory pooling in practice does not fix stranding as of 2023 [25]. According to Levis, Lin, and Tai [25] this is because CXL infrastructure (i.e., PCIe) is too expensive. To address this, it might be reasonable to share memory between compute nodes in a single rack. This could be a compromise compared to dedicated CXL memory pools. Furthermore, it adds software complexity according to the authors [25]. However, this may be mitigated with more work in the field.

CXL 2.0 also introduces Global Persistent Flush (GPF). This natively enables persistence guarantees for the memory on a CXL device accessed through `CXL.mem` [8].

In 2022, CXL 3.0 was released. It features support for multi-level switching, allowing hundreds of machines to connect to CXL devices [7]. CXL 3.0 also introduces Unordered I/O (UIO) and Back-Invalidate (BI) [7]. They can be used for a new device class combining HDM-H and HDM-D: *Host-managed Device Memory — Device-coherent with Back-invalidate support* (HDM-DB) [7]. We will not discuss HDM-DB devices in this research.

2.2 Hybrid SSDs

Byte- and Block-Addressable SSDs (2B-SSDs) [5] provide two different ways to access their storage. For one, they can still be accessed like any other NAND flash storage with a block interface, e.g., an NVMe drive. Additionally, they also offer fine-granular addressing on a per-byte level [5]. In the following, we will use the term "byte-level access" to stick to the naming of 2B-SSDs, although we are aware that in practice access is not as fine. 2B-SSDs offer cacheline-granularity access [5].

More generally, SSDs that offer fine granular access, supported by onboard DRAM, as well as block-level access, are referred to as *hybrid SSDs*. They are CXL type 2 devices and implement the respective interfaces `CXL.io` and `CXL.mem` [7]. While not necessary, hybrid SSDs typically provide persistence guarantees for their memory and can be used as Persistent Memory (PM) [55].

Fine-granular access offers significant improvements in latency [6]. Especially for small reads and writes, it is not necessary to fetch or write back entire blocks. This reduces traffic and thus yields lower latencies [5, 6, 15]. However, byte-level access is not ideal for high throughput. When accessing large chunks of data, fetching entire blocks at once and caching entire pages is significantly faster than byte-level access [6]. As a result, write bursts with many small writes also drastically reduce performance [5]. Write Combining (WC) aims to mitigate this by employing a WC buffer that accumulates writes [26]. However, as the WC buffer is not persistent [28], this is not a viable solution for hybrid SSDs that by themselves may offer PM [5, 26, 55]. The persistence domain does not even reach the SSD. Li et al. propose a *write-verify-read* operation to solve this problem [26]. After writing we immediately read 0B to ensure that the write operation is *non-posted*[1] [26].

In order to offer byte-level addressability, hybrid SSDs typically use the on-device DRAM to buffer requests [5]. However, there are also frameworks that do not require hybrid SSDs. Instead, they use the host's memory as a buffer and build byte-level access on top of that. Notable here is *Pipette*, which aims to improve fine-grained reads [6].

At the time of writing, there exists exactly one hybrid SSD offering byte- and block-level addressability [55]. The Samsung *CXL Memory Module Hybrid* (CMM-H) is a prototype and not currently freely available [55]. As the name suggests, the SSD supports CXL (`CXL.io` and `CXL.mem` to be precise) 1.0, and also implements GPF for crash consistency [55]. In recent benchmarks, CMM-H shows $1.2\times$ to $5.9\times$ higher latencies compared to local DDR5 when exceeding the on-device DRAM capacity [55]. Thus, it is not suitable for workloads requiring low-latency memory. As a result, Zeng et al. concluded that CMM-H requires additional software logic to perform well with applications requiring large amounts of memory [55]. Furthermore, they also found out that CMM-H works well for write-heavy applications, persisting small amounts of data at a time. Using hybrid SSDs as PM works especially well for Write-Ahead Logging (WAL) or file system journaling, workloads that require persistence guarantees [55]. Zeng et al. [55] measured an up to $4.59\times$ speedup in sequential deletes due to rendering WAL unnecessary. Typical real-world workloads employing WAL are databases such as PostgreSQL [2] and Redis/Valkey [33, 45].

2.3 DAX

For a long time Linux has had page caches accelerating reads and writes for applications [16]. However, Linux uses the host's DRAM to cache pages, which inherently is not persistent [16]. The *persistence domain* already breaks at this point. Even if PM was available, persistence guarantees could not be made. An application writing to PM may assume the data is immediately persisted. If, however, the data ends up being cached in Linux's page cache, there is a risk of data loss.

To solve this, Linux introduced Direct Access (DAX) [9]. With Linux Kernel 6.14.0, `ext2`, `ext4`, `xfs`, `virtiofs`, and `erofs` support DAX [9]. DAX extends the persistence domain by bypassing page caches [9]. It allows applications to

[1]Write operations can be broken up into *posted* and *non-posted* operations. Non-posted write operations require a success response before continuing [8].

directly access (external) memory devices [9]. This enables writes to have a higher chance of being truly persisted, if the respective device offers PM.

DAX requires explicit support. A DAX file must be explicitly mapped as such in order to use DAX. Linux provides specific flags on a per-file and per-directory basis [9]. Alternatively, a mount option can also force all mappings to be DAX mappings [9].

2.4 Hybrid SSD Support

Habicht et al. [18] found that the concept of traditional DAX does not work for hybrid SSDs. One major drawback is DAX's lack of fine-granular mapping. It only supports a per-inode flag and as such only allows mapping entire files, not individual file ranges [18]. Additionally, DAX expects low-latency uniform storage access, which cannot be guaranteed with CXL-based hybrid SSDs either [18]. As a solution, Habicht et al. [18] introduce a new concept of *DAX mappings*. It supports page-granular mappings and is purpose-built for hybrid SSDs.

Hybrid DAX mappings establish a new *hybrid interval tree* that only contains DAX Virtual Memory Area (VMA) structs [18]. This is to avoid having to search the big interval tree Linux puts every VMA in. It allows to efficiently figure out if a file has any DAX mappings and iterate over them [18]. It makes a file's DAX mappings easily and efficiently accessible. When a DAX mapping is requested, the respective VMA is put in both interval trees.

Furthermore, Habicht et al. [18] introduce a *persistence-aware page cache*. It allows taking properties of the underlying memory technology into account, especially PM. An `fsync` for a page that is volatile memory-backed requires a full writeback. However, if the page is backed by non-volatile memory, the majority of the writeback can be skipped. Habicht et al. [18] call this a *lightweight sync*. It is comparatively much cheaper [18]. They further optimize the performance of `fsync` by keeping data in a dirty state where possible [18]. At this point the data is out of sync with the backing storage, but it is persisted and safe. The async writeback will later clean this up, bringing everything back in sync [18]. Cleaning up the dirty pages allows the page allocator to quickly evict them later if need be.

Habicht et al.'s [18] implementation does not support migrating pages from the volatile system memory to the hybrid device. Instead, their approach currently relies on busting the page cache for the upgrade. Similarly, the downgrade also requires flushing the then-non-volatile pages [18].

To make the functionality available in user space, the implementation exposes an `mmap` flag: `MAP_DAX` [18]. This allows applications to explicitly memory map DAX pages.

Our research aims to remove this need for requiring explicit mappings. We will do so by transparently upgrading files to use the PM of hybrid devices if we recognize that they have strong persistence requirements. This affects files with many writes that are often followed by `fsync` calls shortly after. As already mentioned, typical applications showing such access patterns are databases [55, 2, 33, 45].

2.5 XArray

An *XArray* is an expandable data structure in the Linux Kernel [51]. In short, it combines the benefits of a resizable array and a doubly-linked list. The XArray allows efficient index lookups, accessing the previous and next elements cache-efficiently, and extending the array without needing to copy over all data to a new array [51]. The data structure allows storing arbitrary 4-byte aligned pointers. If a pointer is not required, any arbitrary unsigned long smaller than `LONG_MAX` can also be stored in the array instead [51]. For this the API offers dedicated methods `xa_mk_value` and `xa_to_value` which take care of the proper alignment.

Additionally, an XArray allows setting marks on its entries. It supports three marks `XA_MARK_0`, `XA_MARK_1`, and `XA_MARK_2` that can be set or unset [51]. With these each entry can have up to $2^3 = 8$ states. The API already provides interfaces to iterate over the structure, looking only for entries with a specific mark set [51].

Furthermore, the advanced XArray API enables unmanaged locking [51]. This allows for efficient iterations over the array in linear time [51]. Lastly, the advanced API also brings multi-index entries. Thus, multiple indices (aligned to a power of two) can be put together. Accessing either of those indices returns the same entry [51].

All of those features make it a good data structure for storing pages, which is already an important use case of XArrays [51]. Given that our tracking will sit alongside the page caches, XArrays are the best solution for our work as well.

Chapter 3

Related Work

Besides databases, another area with high persistence demands is file system journaling [54]. We will present *Directory-based Journaling File System (DJFS)* in section 3.1 which shows how to improve journaling performance if writes cannot be sped up.

More closely related to our work, there are many file systems aiming to improve the usage of byte- and block-addressable SSDs. We will discuss *Pipette* [6], *CoinPurse* [52], *RomeFS* [56], and *ByteFS* [26] as examples in section 3.2, section 3.3, section 3.4, and section 3.5, respectively.

Lastly, there is a novel approach updating existing NAND such that bulk byte-level write throughput can be significantly improved. With this, block-level addressing may potentially become irrelevant. As such, *srNAND* [24] offers a true alternative to all of the examples mentioned above. We will discuss *srNAND* in section 3.6.

3.1 DJFS

Journaling is employed by many file systems such as ext4 to ensure crash consistency [22]. It works by appending the operation the file systems wants to do to a *journal* first, before then actually executing it. Appending to the journal is a faster operation, reducing the risk of something happening to the system before the journal has been persisted [22]. There are inherent challenges to journaling such as lock contention or transactions locking up [54]. Especially the latter becomes more of an issue with finer journaling (i.e., on a per-file level or even more granular), given there are more transactions over all [54]. Directory-based Journaling File System (DJFS)[1] tries to solve this by extending the journaling

[1]It is worth noting that the paper is funded by Samsung [54]. Given the timing it would be possible that Samsung is interested in this for their CMM-H SSD.

scope [54]. In DJFS, journaling is done on a per-directory basis [54]. The respective transaction will be chosen by *path-based transaction selection* [54]. For a given file `/path/to/my/file` for instance, the corresponding transaction will be on `/path/to/my`.

Typically, a file may have multiple paths pointing to it, and as such also multiple parent directories. In traditional approaches the parent directory often gets the transaction, which may lead to ambiguity if there are multiple parent directories [54]. Thanks to the path-based transaction selection, DJFS solves this problem.

While their approach is effective, Yoo et al. show that `fsync` latencies are non-negligible, although comparatively low at $0.2\,\mathrm{ms}$ thanks to the small transaction size [54]. We find it likely that our work would improve latencies here, given we can dynamically upgrade the journal to use DAX mappings on a PM.

3.2 Pipette

Pipette is a file system optimized for reads with spatial locality [6]. Similar to all the other file systems we discuss here, Pipette tries to leverage a fine-grained as well as a block-level interface, optimizing for every use case [6]. Part of Pipette's work is a *Read Engine* extension to the SSD's firmware atop the flash memory [6]. Based on this, Pipette has a kernel implementation with access to both interfaces, choosing the best one for every use case.

Latency of fine-grained interfaces is already very low compared to block-level flash storage access. However, throughput is bad in comparison [6]. Bai et al. [6] improve the throughput of using the byte-addressable interface. They do so by implementing a new, dedicated read path on top of the fine-grained interface and adding a caching layer [6]. Bai et al. are able to show $> 30\%$ I/O throughput improvements in real-world benchmarks with their implementation [6]. However, their work is focused on improving read throughput [6], whereas we try to optimize writes with persistence requirements.

3.3 CoinPurse

Contrary to Pipette, *CoinPurse* focuses on writes, especially small, i.e., <block size, updates [52]. Besides the unmodified block interface, CoinPurse offers a new partial update interface which uses the byte addressable interface of the SSD.

CoinPurse keeps track of dirty update ranges [52]. When persisting data, the host determines the interface (block or partial update) based on the size of the dirty ranges. If the request should go to the partial update interface, the host writes an update record to the DRAM part of the hybrid SSD [52]. This update record only contains the parts that actually changed [52]. This way, CoinPurse makes good use of the usually rather small DRAM available on the device while not exhausting it.

CoinPurse ensures ordering consistency for writes intertwined on both interfaces [52]. It also provides crash consistency when given appropriate hardware support of the hybrid SSDs to persist the DRAM in case of a power failure.

In their benchmarks, Yang et al. [52] find that CoinPurse performs significantly better than F2FS, ext4, and XFS in writes [52]. Specifically, it outperforms ext4 by $5.93\times$ in artificial writes, which is also directly reflected in performance improvements of write-heavy applications [52]. In Filebench's `Varmail` [13] and other write-intensive real world applications they found $\sim 2\times$ improvements compared to ext4 [52]. However, CoinPurse has degraded performance for delete operations compared to ext4 due to checkpointing of `fsync`'d delete operations [52].

3.4 RomeFS

RomeFS also employs byte-granular addressing as well as block addressing [56]. To access the respective interfaces, RomeFS uses `CXL.mem` and `CXL.io` and as such requires a hybrid SSD with appropriate support [56].

Zhan et al. [56] identify one significant issue with current approaches: Data can end up being scattered across the CXL memory and flash storage. RomeFS tries to prevent this by adding merge-on-write (MOW) and a per-block data log write-back mechanism (LWB) [56]. As the name suggests, MOW will merge new data log writes with existing ones if the addresses overlap [56]. LWB will write data logs back to the entire blocks when MOW fails or the device runs out of memory [56]. These mechanisms help with reducing fragmentation and free up memory.

In order to cope with data scattered across both regions, Zhan et al. also introduce a new hybrid parallel file indexing mechanism [56]. Additionally, RomeFS offers metadata journaling with dual-path transactions [56]. That is, `CXL.mem` and `CXL.io` transactions are independent; multiple writes on both interfaces can be in-flight at the same time [56].

Furthermore, the file system also implements write request splitting. It sends block-aligned writes to the `CXL.io` interface, and residual data is then sent on the `CXL.mem` interface [56]. This way RomeFS provides the best of both worlds.

In their benchmarks, Zhan et al. show that RomeFS performs very well [56]. The real-world application benchmark GridGraph [42] yield improvements of up to $3.28\times$ compared to ext4 [56]. It also does not exhaust the CXL memory, given large blocks will always be written to the flash storage with `CXL.io` [56].

3.5 ByteFS

Li et al. analyze typical file system operations and their ideal interface before implementing *ByteFS* [26]. This yields important characteristics of file systems (ext4 and F2FS in this case) and allows them to build a framework with those characteristics in mind [26]. There are a couple of findings that stand out due to their impact: Firstly, inode reads can contribute to up to 82% of the traffic for metadata-intensive workloads [26]. Secondly, journaling makes up a significant part of the overall traffic. In ext4, 30.7% of the traffic is dedicated to journaling [26]. As a result Li et al. [26] conclude that writes $< 512\,\mathrm{B}$ should be written in cachelines. Larger chunks of data should use the block interface, persisting entire 4KB pages. This ensures optimal latencies and throughput according to their benchmarks [26].

ByteFS is a dual interface file system as well, so it also supports byte-level and block-level access [26]. Contrary to other approaches however, it does not only expose the external device's DRAM for byte-level access and the flash storage for block addressing. ByteFS involves an SSD firmware extension, reorganizing the DRAM on the device as a log-structured write buffer [26]. As such, the entire SSD presents itself as both a `CXL.mem` and a `CXL.io` device with the entire capacity [26]. The file system then automatically chooses the most appropriate interface based on the access pattern and size [26].

Lastly, ByteFS also ensures crash consistency, assuming hardware support in the form of sufficient capacitors on the SSD. Besides write ordering and atomicity, the firmware updates incorporated in ByteFS take care of flushing data in the DRAM before a power failure [26].

ByteFS has a similar result to what we want to achieve with transparent DAX upgrades: application-transparent usage of hybrid SSDs using the CXL interfaces. Contrary to our work however, ByteFS requires SSD firmware modifications. It is also inherently incompatible with other file systems, given it is a dedicated file system by itself. Our approach however has chances to become file system agnostic, assuming a set of features to be present in the FS.

3.6 srNAND

Currently, NAND read throughput suffers for many small, random accesses [24]. This is in part due to general overhead, but mostly because NAND chips inherently fetch entire blocks [24]. Even when only fetching a single $64\,\mathrm{B}$ blob, the underlying flash will fetch an entire block. In fact, NAND chips will fetch an entire block for *every* $64\,\mathrm{B}$ request, even when those $64\,\mathrm{B}$ ranges are next to each other and already included in a prior block [24]. According to Lee et al. [24], a major reason for NAND chips fetching larger blocks is the ECC overhead [24].

With more data, the parity bits become less significant compared to the overall payload. For *srNAND*, the authors found that fetching 256 B data words is a good middle ground. On top of those 256 B, there are 30 B of parity bits [24].

The data combined with the parity bits is called a code vector (*syndrome*). That code vector is calculated on the SSD with the firmware changes involved in srNAND [24]. This enables Lee et al. to only send the relevant part of the vector the application is actually interested in; i.e., 64 B of data + 30 B parity [24]. Thus, the overall communication overhead is significantly reduced and the drive fetches smaller amounts of data in the first place.

Additionally, srNAND also uses *Request Merge Optimization*. It merges multiple successive read requests into a single larger one [24]. In many cases the drive can fulfill these in the same time frame, given it fetched 256 B in the first place.

Under optimal circumstances, Lee et al. measured $4.3\times$ improvements with pure reads [24]. With increasingly more writes in the mix the performance gains decrease however [24].

3.7 Transparent DAX Mappings

Khalil et al. [23] also build on top of the hybrid storage support mentioned in section 2.4. They introduce *Transparent DAX Mappings* (TDMs), a way to bypass the kernel for I/O requests, writing directly to the hybrid SSD's DRAM [23]. Khalil et al. [23] employ a *Transparent DAX Mappings Manager* that resides in the kernel and notifies `libc` about a potential upgrade decision via shared memory. If the file should be upgraded to DAX mappings, the TDM manager makes sure future access goes directly to the hybrid SSD's DRAM. The TDM manager also takes care of the profiling and deciding which files to upgrade, however that part has not been implemented yet [23].

In their testing they show significant improvements. With YCSB [53] on Valkey [43], TDMs yield latency improvements of 67.5% in the `fsync=always` mode [23] In terms of throughput, they manage to match the throughput of the `fsync=no` mode, illustrating how much bypassing the kernel helps [23].

In principle, their goal is the same as ours and they manage to get better results by avoiding kernel invocations. However, TDMs still have a couple of limitations. For one they only allow promoting entire files to the hybrid SSD [23]. Contrary, our implementation allows for much finer upgrades; theoretically down to page-granularity. Furthermore, Khalil et al. [23] do not have the actual policy and metric collection implemented yet. Meanwhile, our implementation focuses on the tracking aspect and trying to make good decisions on when to upgrade or downgrade a file. As such, tracking insights from our work may be helpful for TDMs, which prove to ultimately show better performance.

Chapter 4

Approach

The goal of this work is to transparently upgrade file ranges with strong persistence requirements. Furthermore, upgraded file ranges that do not show behavior requiring Persistent Memory (PM) anymore should be downgraded again to relieve memory pressure on the PM. Primarily, we need to find policies for deciding when to upgrade or downgrade a file range, alongside with metrics to base the policies on. The assumption here is that an application's behavior in the past and present also likely represents its future behavior. Under this assumption, we will make our decision based on observed behavior in the past.

Fundamentally, the approach requires four aspects; (1) tracking, (2) decision making, (3) upgrading, and (4) downgrading. For the upgrade and downgrade we can leverage Habicht et al.'s [18] approach. Their work already provides most of the bits and pieces, requiring us to only implement some technical details we will discuss in chapter 5.

For (2), we employ a simple policy based around the writeback-to-writes ratio. We use an upgrade threshold and downgrade threshold to evaluate whether we should upgrade, downgrade, or do nothing.

Finally, the most involved part is (1) the tracking. We want our policy to be able to decide whether a given file range has strong persistence requirements. We will discuss what strong persistence requirements mean to us in section 4.1. In order to avoid an oscillating behavior, we consider both the writeback count as well as the write count for each file range. Its ratio will give us a more stable metric over time. Furthermore, we want our approach to respect temporal locality. Specifically, a file range that slowly accumulates writebacks over a longer period of time is not considered to have strong persistence requirements by us. For this, we introduce a decay to both the writeback and write counts. Section 5.1 will explore the tracking, our metric, and the decay in more detail.

4.1 Persistence Requirements

For some applications it is especially important that the written data is immediately safe. It may not be at risk in case of a power loss. Thus, these applications cannot rely on volatile memory. Popular examples of applications that have such requirements are databases and file systems. Databases typically deal with important data, often in a transaction executing multiple queries. They require *crash consistency*.

Crash consistency fundamentally requires two properties: (1) persistence and (2) being able to recover from any permutation of persisted data. Just because all application data that has been written is guaranteed to survive a crash does not necessarily mean an application can fully recover from that. Metadata necessary to correctly interpret the data may still be lost. (2) requires applications to keep this in mind and build measures that help ensuring crash consistency. A typical solution to this is the employment of Write-Ahead Logging (WAL). As the name suggests, WAL writes the operations it wants to do to a file first. This allows replaying operations if any of them failed. WAL is comparable to the idea of file system journaling. To address (1), an application may call a syscall like `fsync` or `fdatasync`. This triggers synchronous writeback of dirty pages [14, 11].

We consider an application to have *strong persistence requirements* if it triggers manual writeback of its data often. Those are the applications for which it is especially important that the written data is immediately persisted. We directly transfer the concept of (strong) persistence requirements to file ranges. If a file range is being written back often it has strong persistence requirements.

4.2 Tracking

We need to track metrics to make a decision to upgrade, downgrade, or not do anything, based on the policy. While the metrics should not lead to hasty decisions, the approach should still be reactive to behavior changes of an application. If an application suddenly changes into a pattern where it shows strong persistence requirements from now on, affected file ranges should be upgraded quickly to improve performance of the many `fsync` calls. Furthermore, we also do not want an oscillating behavior where files ranges are permanently toggled between upgrading and downgrading. Primarily, because Habicht et al.'s [18] implementation entails flushing the pages from cache with every upgrade or downgrade. These frequent cache misses would cause significant performance problems. An exemplary graph with oscillating behavior is shown in Figure 4.1. The upper × mark upgrade decisions, while the lower × mark downgrade decisions.

Lastly, we want to take *temporal locality* of the file access pattern into account. A file range slowly accumulating writebacks over a longer period of time does not make it a file range with strong persistence requirements. Most file ranges will eventually exceed any threshold in that sense. Thus, we also need our policy to respect temporal locality.

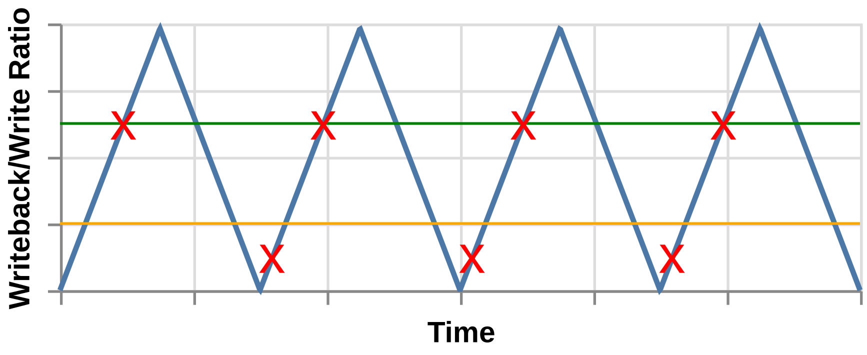

Figure 4.1: Schematic oscillating behavior of an application

We aim to address all these inflicted requirements by tracking multiple metrics and thresholds. First, we define our thresholds. We introduce three thresholds; an upgrade threshold U, a downgrade threshold D, and a write threshold F. In Figure 4.1, the upgrade threshold is the upper green line, and the downgrade threshold the lower orange line. Contrary to the upgrade and downgrade thresholds, the write threshold is an absolute value. While the upgrade and downgrade thresholds are compared against the ratio we introduce in subsection 4.2.1, the write threshold is compared against the write count. We require at least F writes on any given file range before evaluating the metric and considering the range for an upgrade. This write threshold should help prevent the approach from getting into an oscillating behavior: For small write and writeback counts, a minor change would have significant impact. Furthermore, introducing dedicated upgrade and downgrade thresholds is an additional measure against such oscillation. In general, the goal is to try to prevent a small change in our measured values to have a big impact on the metric we consider for our policy.

4.2.1 Writebacks/Write Ratio

For the metric $\mathcal{M}$ we use the ratio between writes and writebacks

$$\mathcal{M} = \frac{\text{<writeback count>}}{\text{<write count>}} \cdot 100 \in [0, 100]^{1} \tag{4.1}$$

with $[\mathcal{M}] = \%$ which leads to the policy for our decision

$$\mathcal{D} = \begin{cases} \text{Upgrade,} & \text{if } \mathcal{M} > U \\ \text{Downgrade,} & \text{if } \mathcal{M} < D \\ \text{Do nothing,} & \text{otherwise.} \end{cases} \tag{4.2}$$

$\mathcal{M}$ being a ratio further helps with smoothing the metric with an increasing write count. It also helps reduce the aforementioned hasty decisions. An application that triggers many writebacks in rare bursts but overall has a permanent flow of writes should not be upgraded. Given the majority of writes do not need to be persisted immediately, it might be better for the application to use the volatile page cache. Although, at the expense of suffering through those bursts. Furthermore, upgrading at bursts and potentially downgrading between them also causes many page cache flushes. This, again, is oscillating behavior with the downsides as mentioned in the beginning. Unfortunately, the tested workloads do not show such behavior and we lack insight into real world server workloads. Thus, we are unable to provide comparisons here showing what the performance difference looks like between upgrading file ranges at every burst or not. It certainly also depends on the frequency of those bursts, which our approach can be adapted to by altering the decay as explained next.

4.2.2 Decaying

For considering *temporal locality*, we add a decay to the writeback count. Every increment of the writeback counter will decay after a fixed amount of time. A small decay value should make the algorithm more reactive by requiring stronger temporal locality and more quickly yielding a downgrade decision again. Similarly, a high value should leave more time between writebacks for the application to exceed the upgrade threshold. It also takes longer for $\mathcal{M}$ to drop below the downgrade threshold again. Generally, the algorithm becomes more inert.

The second question to consider is whether to decay the write count as well. If the write count does not decay at all, the algorithm is less reactive, but also more stable. If a file range does not have many writebacks for a long time, the write

[1]Naturally, a file range will always have more writes than writebacks. It may have more `fsync` calls, but if there was no write there is nothing to write back.

count will eventually become so large that even a burst of writebacks will not lift the ratio over the upgrade threshold anymore. At this point the file range is locked in the downgraded state.

The other extreme would be decaying the write count at the same rate as the writeback count, or even quicker. This would have the same effect as only tracking the writeback. The write count would not provide any smoothing over time. It would not help gain stability, but the algorithm would be very reactive. In fact, it would be susceptible to bursty behavior and potentially run into oscillating behavior, too.

Unfortunately, as already mentioned, we were not able to discover such bursty behavior in any of our real-world benchmarks. As a compromise, we opted for a conservative approach: Adding a decay to the write count, but with a significantly higher delay than the writeback decay has. With this, we are hoping to still prevent oscillating behavior caused by bursts. Long-living processes that have a slow but steady writeback over time do not fit our conception of applications with strong persistence requirements. Decaying the write count more slowly than the writeback count should build resilience against these processes and prevents them from being inappropriately upgraded. However, we are unable to verify that due to the aforementioned lack of production-insight.

Neither of our benchmarks are negatively impacted by this decay, while also being robust against oscillation. If, however, there are real world applications that show writebacks in bursts and would benefit from being upgrade during those bursts – or stay upgraded the whole time – there is a case for changing the delay of either decay. The implementation shall make that easily possible.

Alternatively to tweaking the time it takes for a writeback count to decay, it is also possible to employ a *multiplicative increase, additive decrease*-like algorithm. With this, a fixed-time periodic decay could gradually decrease the writeback count. The result however would be the same: varying the time it takes to decay a writeback count. Either solution delays the decay. We have opted to stick to the additive increase and decrease. Its implementation is simpler and less work items need to be queued.

4.2.3 Tracking Granularity

In testing we found that tracking writebacks and writes on a per-page basis is too granular. Specifically, individual pages may still hit the threshold and get upgraded, however by the time they were upgraded, the application hard already started writing to the next pages. The approach was always too slow to react.

To address this, there is a tracking granularity order constant $\mathcal{G}$ that can be modified at build time. $2^{\mathcal{G}}$ pages, aligned to a power of two, will be combined and all have the same writeback and write counter. We consider this a *file range* to track and upgrade. Logically, a smaller granularity allows for finer upgrades. The amount of pages that do not have strong persistence requirements but may still get upgraded is lower. Thus, less PM is wasted. However, as already mentioned, if the tracking is too fine the asynchronous upgrade may happen by the time the file range is not of interest anymore. Similarly, a higher granularity leads to coarser upgrades. The approach is more likely to track and upgrade uninteresting pages. Primarily, the tracking aspect may make it harder for a file range to be upgraded. Pages with low persistence requirements might affect the tracking and pull down the overall ratio.

That being said, if many pages, and especially ones at the beginning of a file range, show strong persistence requirements, we are likely to upgrade the file range. With many successive pages being upgraded to the PM, performance is likely to be improved significantly.

Chapter 5

Implementation

In the following we will elaborate on our implementation. It is compatible with the Linux Kernel 6.6 and based on Habicht et al.'s [18] patch set. As such, we only support ext2 and ext4. The implementation is device-agnostic as long as the device supports the `CXL.io` and `CXL.mem` interfaces. As motivated initially, our work does not require application support, i.e., user space modifications.

First, we will explain how we track file ranges and which properties we take into account. We will also discuss how the tracking implementation evolved and why we ended up with this approach in section 5.1. Then, section 5.2 will delve into the process of making the upgrading/downgrading decision and performing the transition. For the latter we primarily leverage the existing work by Habicht et al. [19].

5.1 Tracking

The goal of this work is to intelligently detect and upgrade file ranges that have strong persistence requirements. As our metrics we track two properties per file range.

1. `writeback_count`
2. `write_count`

`writeback_count` tracks how often a specific range of a file is written back. `write_count` tracks how often a specific range is written to in the first place. With these two values every given file range's persistence requirements are known: The ratio of writebacks to writes, defined as $\mathcal{M}$ in subsection 4.2.1.

The counters are managed using XArrays [51]. For one, they provide the benefits of a resizable array as well as a doubly-linked list: fast indexing and iterations. They also allow setting marks at entries, which we will use in the following. An entry in both of those arrays represents $4\,\mathrm{KiB}$ page-size granularity given the kernel version we use does not support huge pages for ext4. Lastly, we also benefit from their multi-index entries.

A constant `GRANULARITY_ORDER` allows changing the granularity of the file range by using multi-index entries. `GRANULARITY_ORDER` is an integer and is interpreted as combining $2^{GRANULARITY\_ORDER}$ indices in the array. For instance, to achieve a $2\,\mathrm{MiB}$ granularity, we need to combine $\frac{2\,\mathrm{MiB}}{4\,\mathrm{KiB}} = \frac{2^{21}}{2^{12}} = 2^9$ indices. Hence, `GRANULARITY_ORDER` $= \log_2(2^9) = 9$.

Further, we add a decay to the writeback count. Whenever we increment a field in `writeback_count`, we also queue a delayed `Workqueue` [49] job. After a specified amount of *jiffies*[1] [38], `writeback_count` is decremented again, effectively undoing the increment.

With this we introduce temporal locality to our tracking. If an application only rarely writes back data, the `writeback_count` entry will always be zero or close to zero. In that case, we will not decide to upgrade that file range.

5.1.1 Marks

We use the `writeback_count xarray` for state tracking. There, we introduce three marks:

- `TRANSPARENT_DAX_TO_DOWNGRADE`
- `TRANSPARENT_DAX_TO_UPGRADE`
- `TRANSPARENT_DAX_UPGRADED`

We use these to represent six different states. This tracking allows us to know exactly which state a file range is currently in. Knowing this helps avoid queueing contradictory work items or duplicating items in the queue. It also helps ensuring that any given file range is only operated on once by providing some implicit locking.

The entire lifecycle can be seen in Figure 5.1. Initially a regular file range that is in the typical page caches on volatile memory. If the policy yields the decision to upgrade, the mark `TRANSPARENT_DAX_TO_UPGRADE` is added. As soon as a `Workqueue` job gets to the file range and starts upgrading it, the flag `TRANSPARENT_DAX_UPGRADED` is set. Once the upgrade is complete, the flag `TRANSPARENT_DAX_TO_UPGRADE` is removed again.

[1] A *jiffy* is a time unit on Unix systems based on the kernel constant *HZ*. On modern Linux Kernels a jiffy defaults to 4ms [38].

Later, the metric may evaluate to a downgrade decision. The first step is setting `TRANSPARENT_DAX_TO_DOWNGRADE`, indicating that the file range should be downgraded. Once a background job gets to the file range, it removes the flag `TRANSPARENT_DAX_UPGRADED`, marking the file range as in a state of currently being downgraded. If successful, `TRANSPARENT_DAX_TO_DOWNGRADE` is also removed, leaving the file range at a blank slate with no marks set.

In either case, if the upgrade or downgrade is unsuccessful, the respective mark again is added or removed again. This indicates that a background job should attempt to upgrade or downgrade this file range again the next time it looks for tasks.

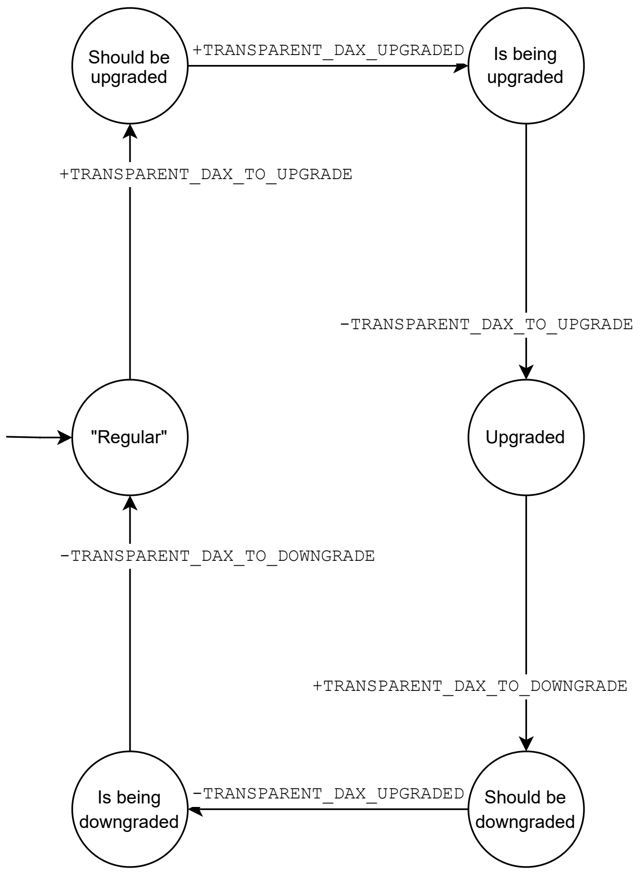
Should be upgraded
+TRANSPARENT_DAX_UPGRADED
Is being upgraded
+TRANSPARENT_DAX_TO_UPGRADE
-TRANSPARENT_DAX_TO_UPGRADE
"Regular"
Upgraded
-TRANSPARENT_DAX_TO_DOWNGRADE
+TRANSPARENT_DAX_TO_DOWNGRADE
Is being downgraded
-TRANSPARENT_DAX_UPGRADED
Should be downgraded

Figure 5.1: The lifecycle of a file range's marks

5.2 Upgrading and Downgrading

In the following section we first provide some context how writeback works in the Linux Kernel. This is done in subsection 5.2.1. Then, we explain how we track our metrics and apply the chosen policy in subsection 5.2.2. Lastly, subsection 5.2.3 and subsection 5.2.4 explain how we upgrade and downgrade a specific file range, respectively.

5.2.1 Linux Kernel Writeback

Fundamentally, a writeback describes writing back data from the OS's page cache to the disk. While only living in the page cache, data on volatile memory will get lost in case of a power failure or similar hardware error. Only if data is written back to the drive, and the page is not in a *dirty* state anymore, is it considered safe. Writeback can be triggered manually, e.g., by calling `fsync`, but it is also run periodically.

In the Linux Kernel, the writeback is split up into two tasks. First, respective pages are tagged for writeback. There, pages tagged as `DIRTY` will get their tag updated to a special `TOWRITE` tag instead. This point in the code also turns out to be a great place for our tracking to hook into. The actual writeback is done in a second job. Here, the kernel iterates over all pages with the `TOWRITE` tag set and writes them back to disk.

Discovery and writeback are separated to prevent livelocks [29]. A livelock may happen if an application creates dirty pages more quickly than the writeback executes. By moving the slow writeback into an async job, the likelihood of a livelock decreases significantly, since the pure tagging of pages is very fast. Essentially, the kernel splits the writeback up into a fast and slow part, so that other applications only race with the fast part.

5.2.2 Decision Making

As already mentioned, we hook our tracking into the page tagging. We first get the current writeback count and increment it. We also immediately queue a decrement of the specific writeback count again, in Listing 5.1, line 2, in order to implement the decay. Then, we calculate the ratio between writeback count and write count in percent, i.e., our metric $\mathcal{M}$ from subsection 4.2.1. If this ratio exceeds the `upgrade_threshold` and the file range is currently not upgraded or in the process of being upgraded, the flags are set accordingly to queue it for an upgrade.

Alternatively, if the ratio is below `downgrade_threshold` and the file range is currently upgraded, the flags for the downgrade are set accordingly.

The gist of the implementation can be seen in Listing 5.1, excluding locking.

Listing 5.1: `mm/page-writeback.c#__tag_pages_for_writeback; simplified`

```
writeback_count[index]++;
queue_decrement(mapping, index);

int ratio = 100 * writeback_count[index] /
    (write_count[index] - file_write_threshold);
bool is_upgraded = get_mark(&writeback_count,
        TRANSPARENT_DAX_UPGRADED);

if (ratio > upgrade_threshold && !is_upgraded) {
  set_mark(&writeback_count,
        TRANSPARENT_DAX_TO_UPGRADE);
  clear_mark(&writeback_count,
        TRANSPARENT_DAX_TO_DOWNGRADE);
}

if (ratio < downgrade_threshold && is_upgraded) {
  set_mark(&writeback_count,
        TRANSPARENT_DAX_TO_DOWNGRADE);
  clear_mark(&writeback_count,
        TRANSPARENT_DAX_TO_UPGRADE);
}
```

After the bookkeeping of file ranges we queue our transparent DAX jobs for the given file. For the upgrade this works by going over all `writeback_count` entries that are marked with the flag `TRANSPARENT_DAX_TO_UPGRADE` and do not have the flag `TRANSPARENT_DAX_UPGRADED` set. Entries also marked with `TRANSPARENT_DAX_UPGRADED` are currently in the process of being upgraded and should not be queued again. Finally, each matching entry is marked with `TRANSPARENT_DAX_UPGRADED` and a `Workqueue` [49] item is queued for upgrading.

Analogously, file ranges to downgrade are found by going over all entries with `TRANSPARENT_DAX_TO_DOWNGRADE` and `TRANSPARENT_DAX_UPGRADED` set. For each match, `TRANSPARENT_DAX_UPGRADED` is cleared – marking the entry as in-process – and the respective work item gets queued.

Eventually, a `Workqueue` work item will be scheduled to run. It infers the corresponding mapping and index and performs the upgrade or downgrade.

5.2.3 Upgrading

We use the custom hybrid interval tree by Habicht et al. [18] we discussed in section 2.4. Besides the hybrid implementation no other parts of the kernel use that tree. If a hybrid DAX mapping is requested, a Virtual Memory Area (VMA) struct will be put in that tree as well as the kernel's default VMA tree [18]. This enables the hybrid implementation to quickly find hybrid VMAs for a file and iterate over them, if there are any. Given we work with pages that already have a corresponding VMA in the kernel's interval tree, we only need to add it to the hybrid tree. We are not allocating new pages.

We first try to allocate memory for a new `vm_area_struct` that we can insert into the hybrid VMA tree. That VMA makes the hybrid implementation aware of file ranges we upgraded, and also allows us to track which file ranges we upgraded. If for whatever reason that was unsuccessful, we unset `TRANSPARENT_DAX_UPGRADED`, indicating that the file range should be upgraded but is not currently being upgraded. We then exit the upgrade job.

Otherwise, we set the necessary fields on the VMA as can be seen in Listing 5.2 ll. 8-13. Notably this is not a VMA in the traditional sense and as such does not represent an actual mapping. Rather, this is solely used as a data struct to hold some necessary information. This information includes the start and end byte address of the file range as well as the page index. Furthermore, we also set a newly introduced `is_transparent_dax` field to true. This field indicates whether a given VMA is created transparently and can be downgraded if required. Lastly, we set the `VM_DAX` flag on the VMA as expected by Habicht et al.'s [18] hybrid SSD support.

This minimal fake VMA is inserted into the respective hybrid tree and the pages from the page cache are flushed using the existing hybrid interfaces by Habicht et al. [19]. Both are covered by the `hybrid_vma_insert` call in line 15 of Listing 5.2 that is exposed by the hybrid implementation already. Lastly, if successful, we clear the `TRANSPARENT_DAX_TO_UPGRADE` mark. Now, only `TRANSPARENT_DAX_UPGRADED` remains set and the upgrade is done. The gist of the code can be seen in Listing 5.2, all locking is omitted.

Listing 5.2: `transparent_dax_upgrades.c#__upgrade; simplified`

```
1 struct vm_area_struct *vma =
2   kmalloc(sizeof(struct vm_area_struct), GFP_ATOMIC);
3 if (!vma) {
4   clear_mark(&mapping->writeback_count, index,
5              TRANSPARENT_DAX_UPGRADED);
6   return;
7 }
```

```
vma->vm_start = index * PAGE_SIZE;
vma->vm_end =
  (index + (1 << GRANULARITY_ORDER)) * PAGE_SIZE;
vma->vm_pgoff = index;
vma->is_transparent_dax = true;
vm_flags_init(vma, VM_DAX);

hybrid_vma_insert(vma, mapping);
clear_mark(&mapping->writeback_count, index,
        TRANSPARENT_DAX_TO_UPGRADE);
```

5.2.4 Downgrading

For downgrading we first try to find the respective VMA at the given page index in the hybrid VMA tree. This can be seen in Listing 5.3, l.1. If that does not exist or if it is not a VMA that has been upgraded by us – the flag `is_transparent_dax` is set to `false` – we set `TRANSPARENT_DAX_UPGRADED` again and quit the upgrade. This again indicates that the file range is not being downgraded anymore, and that it should still be downgraded.

If we successfully found a VMA, the downgrade continues. All non-volatile pages in the file range are flushed (ll.9-10). Afterwards we delete the VMA from the hybrid VMA tree as can be seen in l.11 of Listing 5.3. We again use the existing interfaces of the hybrid implementation for this [18]. Lastly, we clear the `TRANSPARENT_DAX_TO_DOWNGRADE` mark, indicating that the downgrade is complete. The entry in the XArray now has no marks set anymore. The simplified downgrade function is listed in Listing 5.3, all locking is omitted.

Listing 5.3: `transparent_dax_upgrades.c#__downgrade; simplified`

```
 1 struct vm_area_struct *vma = vma_hybrid_it_iter_first(
 2   &mapping->i_hybrid, index, index);
 3 if (!vma || !vma->is_transparent_dax) {
 4   set_mark(&mapping->i_wb_dirty_count, index,
 5           TRANSPARENT_DAX_UPGRADED);
 6   return;
 7 }
 8
 9 hybrid_flush(mapping, vma->vm_pgoff, vma_pages(vma),
10           FILEMAP_SELECT_NVM);
11 hybrid_vma_remove(vma, mapping);
12 clear_mark(&mapping->i_wb_dirty_count, index,
13           TRANSPARENT_DAX_TO_DOWNGRADE);
```

Chapter 6

Evaluation

After having implemented transparent DAX upgrades we now go on to evaluating our approach. We evaluate our implementation with benchmarks on Valkey [43], PostgreSQL [17], and RocksDB [34]. The results of that are shown in section 6.2.

In our testing we found the overall best results with the following settings:

- Upgrade Threshold: 60%
- Downgrade Threshold: 30%
- Write Threshold: 5
- File Granularity Order: 9

6.1 Methodology & Evaluation Environment

Due to a lack of production hybrid SSDs, we need to use an emulated hybrid SSD. Habicht et al. [19] built software emulation for a hybrid SSD. It takes a NUMA node and backing storage. For the NUMA node, we use a CXL memory expander. It has $256\,\mathrm{GB}$ of memory, which notably is significantly larger than the $16\,\mathrm{GB}$ CMM-H has for instance [55]. However, none of our benchmarks require large amounts of memory. Thus, we do not gain any unintended benefits and the memory expander is comparable in that regard for our testing.

Furthermore, the OS-level emulation requires data flowing between the memory and storage to traverse the host. Contrary, on a true hybrid SSD, the on-device controller would take care of that and data would not leave the hybrid device. That is why the performance is not comparable and we expect higher latencies. However, we are only interested in memory-hit performance in this thesis anyways. Hence, performance of the memory $\leftrightarrow$ NAND storage path does not matter for our work.

We expect the performance of the memory expander to be at least as good as a hybrid SSD. Likely, due to a higher bandwidth of the expander, it may even be better, which in turn might have a positive effect on our results. For the Samsung CMM-H, recent analytics show latencies around 3 times as high as local DIMM. [55]. Meanwhile, Unal et al. [41] measure a latency twice as high as local DRAM with the same CXL memory expander we are using.

The evaluation setup is outlined in Table 6.1. All benchmarks were run on our modified version of the Linux Kernel 6.6.

Component	Specification
Mainboard	Supermicro X13SEI-F
CPU	Intel® Xeon® Silver 4410Y12c/24T @ 2 GHz
Host Memory	4×16 GiB DDR5 RDIMM @ $4800\,\mathrm{MT/s}$
Storage	Samsung 990 PRO 1 TB
CXL Memory Expander	SMART CXA-4F1W
CXL Memory	$4 \times 64\,\mathrm{GB}$ DDR5 RDIMM @ $5600\,\mathrm{MT/s}$
Hybrid SSD Storage	Samsung 970 PRO 1 TB

Table 6.1: Benchmark Hardware

For ease of use, we build a Docker [10] image. It contains all of the dependencies the benchmarks rely on. This allows us to have a reproducible, easy to set up environment that we can quickly move between hosts. Docker containers especially helped us with postgres. On Debian and Ubuntu there are utilities around cluster management, wrapping the `initdb` interface [40, 1]. These utilities make it easier to spin up various postgres instances on the host with data in different locations. Given our host runs Fedora 41 [12], we cannot use those natively and a Debian container is a great solution.

Li et al. [27] indicate that the I/O overhead of Docker is relatively large for writing single bytes. They measured an approximately $30\% - 40\%$ performance degradation compared to running it on the host. However, when writing entire blocks the performance hit becomes nearly negligible [27]. Considering that their work is from 2017, we expect the Docker I/O driver to have improved since then. Furthermore, our benchmarks do not exhibit such fine granular writes. As a result, we consider it viable to run the benchmarks in a Docker container, with the caveat that performance may be slightly better if run directly on the host.

All benchmarks are performed with cold caches. For each test, we run 5 iterations in the case of Valkey, and 3 iterations for PostgreSQL and RocksDB. Each iteration is done on a new process, with a new working set, and new files. This aims to avoid iterations bleeding into each other. Furthermore, all cores are locked at 2 GHz to additionally decrease variance.

6.2 Benchmarks

In order to get an impression of how our implementation works in the real world, we benchmark three databases. In subsection 6.2.2 we run benchmarks on the popular Key-Value (KV) store *Valkey* [43] on version 7.2.5, an open source fork of *Redis* [32]. For that we use the built-in `valkey_benchmark` [44] utility. It supports various tests and the option to pipeline requests, amongst others [44]. Habicht et al. [18] also tested their work on a patched version of Valkey using `valkey_benchmark`, so we can compare our numbers on a stock version of Valkey to their results.

We then move on to *PostgreSQL* (also known as *Postgres*) [17] version 17.6, an open source popular relational database that is gaining more traction and scales well [4]. We discuss the results we obtained using `pgbench` [30] in subsection 6.2.3. `pgbench` is the first party benchmark tool from PostgreSQL. It supports both synthetic as well as real-world oriented workloads. The benchmark defaults to a TPC-B-alike scenario [30]. TPC-B is an accounting workload designed to stress test relational databases [39]. While the original scenario is obsolete, postgres uses a similar workload aiming to simulate a real world use case. `pgbench` allows scaling the workload as necessary, making it strenuous even on modern hardware [30].

Finally, in subsection 6.2.4, we will look at Meta's open source, persistent KV store *RocksDB* [34] on version 10.5.1. RocksDB is interesting as it is used in production at huge scale and also because other related research projects tested with it as well [52, 26]. We benchmark RocksDB with the built-in `db_bench` [35] utility. Initially, we planned on testing RocksDB with the popular Yahoo! Cloud Serving Benchmark (YCSB) [53] workload. However, we found out that YCSB does not support synchronous writes with RocksDB out of the box. As such, the benchmarks cannot benefit from our transparent DAX upgrades, which we confirmed in testing.

For all tests, we distinguish between two types: `default` refers to the case of not having transparent DAX upgrades or any other speedups available. `hybrid` refers to transparent DAX upgrades being enabled with a hybrid SSD available that file ranges can be upgraded to.

6.2.1 Persistence

The databases we look at use either Write-Ahead Logging (WAL), or an Append-Only File (AOF) for persistence [2, 50, 45]. Both are similar in what they are trying to achieve: keeping track of operations to replay them later if necessary.

Valkey supports three modes for its AOF. While `no` disables it entirely, losing persistence guarantees in the first place, `everysec` and `always` append write operations to the AOF every second or with every write, respectively [45]. As a result, `everysec` may lose some writes if the system crashes within a second, whereas `always` is the most resilient persistence mode for the AOF in Valkey. Valkey always uses `FdataSync` to write back changes of the AOF [45].

Next up, PostgreSQL employs WAL [2]. We evaluate three configuration variants of Postgres' WAL. By default, Postgres will force the immediate write-back of changes in the WAL using an `fsync` (or equivalent) call. By setting `fsync=false` [2] this can be explicitly disabled, which will be our `NoFsync` case. However, that can result in unrecoverable data corruption in case of a system crash [2]. That is why `fsync` defaults to `true`, with the `wal_sync_method` `fdatasync` on Linux [2]. We call this case `FdataSync`. Lastly, we evaluate Postgres with setting the `wal_sync_method` to `fsync`. In our evaluation we refer to this by `Fsync`.

Lastly, RocksDB also uses WAL for its persistence guarantees [50]. For RocksDB we test sequential writes as well as synchronous writes to the database. Our `Default` case refers to the sequential writes executed by `db_bench` [35]. No writebacks are slowing down the benchmark in this case. Then, for the synchronous writes we test both the `fdatasync` and `fsync` writeback options again. Similar to the other databases, RocksDB also uses `fdatasync` by default [50], again we call this case `FdataSync`. When setting the database option `use_fsync` to `true`, it will use `fsync` calls. This scenario makes up our `Fsync` case.

6.2.2 Valkey

In the Valkey `SET` test we see significant improvements (up to $7.29\times$ speedup). The test exclusively executes `SET` operations on the Key-Value (KV) store, that is, many small writes. We run 1 million requests with a pipeline length of both 1 and 10, i.e., the amount of possible in-flight requests. Figure 6.1 shows an overview of different upgrade and downgrade thresholds at a fixed write threshold of 5 and a pipeline length of 1.

A file granularity order of 0, meaning tracking at page-granularity, is identical to the `default` baseline. We observe a maximum difference of $< 1\%$ which is well within margin of error. We see similar minimal variation in the other plots. We observe significant improvements with the `always` sync mode with both a file granularity order of 4 as well as 9. Furthermore, a file granularity order of 9 outperforms the file granularity order of 4. This is because this benchmark represents a consistent write stream. Ideally, every file range would be upgraded here. By upgrading a larger file range at once, we overall need to upgrade less often. That is helpful because every upgrade comes with some delay until the thresholds are hit and the file gets upgraded. While allowing the algorithm to be more reactive to access pattern changes, in a consistent stream of writes like in this benchmark, we will not ever make a different decision than to upgrade.

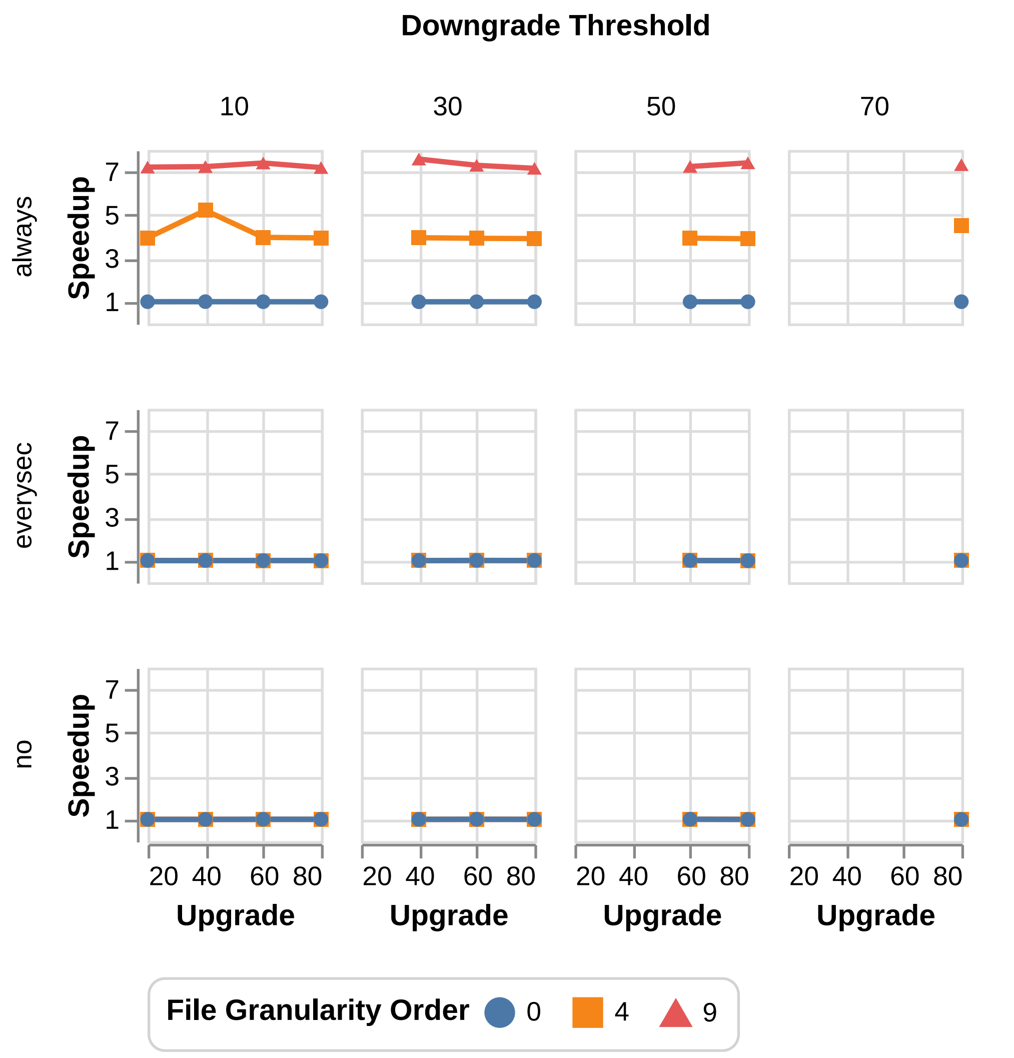

Figure 6.1: Overview of Valkey benchmark runs with a pipeline size of 1 and a write threshold of 5. The shown values are relative to the `default` case. The x- and y-axis represent the upgrade threshold and speedup, respectively. The plots are split vertically by downgrade threshold, and horizontally by AOF sync mode. Both, a file granularity order of 4 and 9, show significant improvements with the `always` sync mode. The modes `everysec` and `no` show no significant improvements.

Unexpectedly however, we do not see a significant impact of the downgrade threshold. That is because the workload, given it exclusively writes data that needs to be written back, does not require downgrades.

At an upgrade threshold of 60, a downgrade threshold of 30, and a write threshold of 5 we can see a $7.29\times$ speedup. Looking into that run more closely, we observe that performance of the `always` mode gets much closer to the `everysec` and `no` sync modes. Figure 6.2 shows that our implementation reaches over half of the other mode's requests per second, making it more feasible to use. We also measure Habicht et al.'s [18] modified Valkey version that uses explicit DAX mappings. We refer to that by the type `explicit`. With the AOF sync modes `everysec` and `no` performance is on par between the different types. Unexpectedly, `explicit` performs worse than our `hybrid` type with the `always` sync mode. We are able to reproduce the `explicit` performance on bare metal, ruling out that this is related to the Docker I/O driver. We assume this might be due to suboptimal performance of the Valkey `mmap` backend. However, we are not able to verify that due to a lack of time.

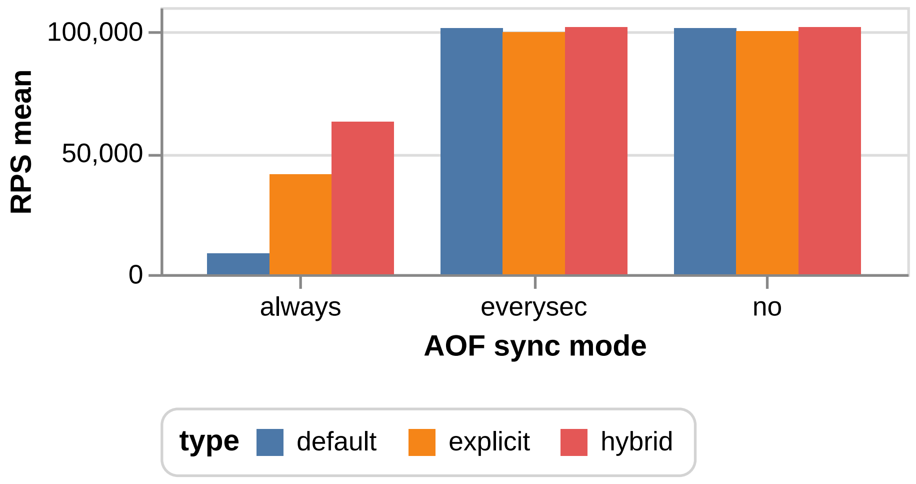

Figure 6.2: Valkey benchmark run with a pipeline size of 1, upgrade threshold: 60, downgrade threshold: 30, write threshold 5, file granularity order: 9. The `explicit` type uses a modified version of Valkey 7.2.5 that manually requests DAX mappings. We observe a $7.29\times$ speedup for the `always` sync mode in the `hybrid` case compared to `default`. `hybrid`, `explicit`, and `default` perform similarly in the other cases. `explicit` falls behind `hybrid` with the `always` AOF sync mode.

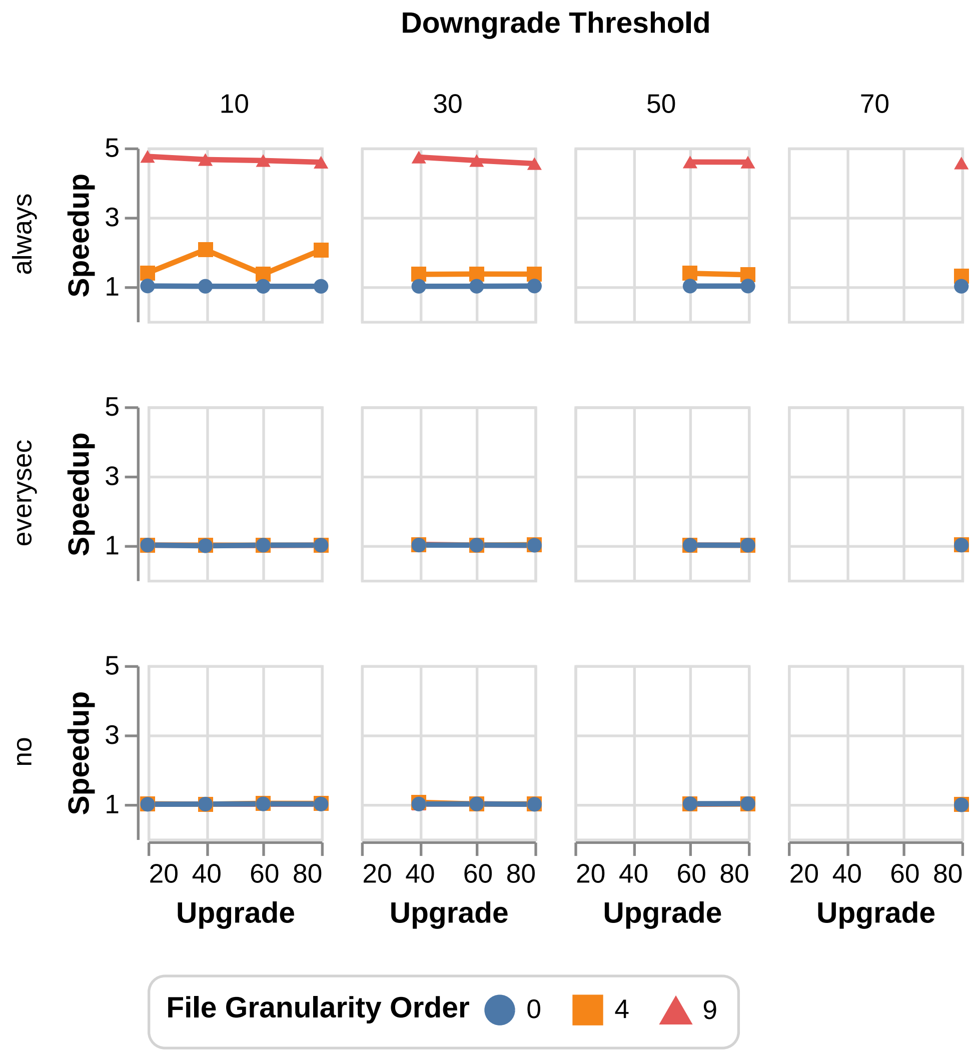

Figure 6.3: Overview of Valkey benchmark runs with a pipeline size of 10 and a write threshold of 5. The shown values are relative to the `default` case. The x- and y-axis represent the upgrade threshold and speedup, respectively. The plots are split vertically by downgrade threshold, and horizontally by AOF sync mode. Especially at a file granularity order of 9, significant improvements are noticeable across the board for the `always` sync mode. `everysec` and `no` do not show significant changes.

With a pipeline size of 10, individual file ranges are done so quickly that the asynchronous upgrades had comparatively smaller effects as can be seen in Figure 6.3. By the time they are done, Valkey is already about to write to the next file range of the AOF. Consequently, with a larger file granularity order results become better as can be seen in Figure 6.3. The larger file granularity causes more pages to be upgraded in advance, making it more likely to be upgraded when Valkey writes to it. In the AOF file sync modes `everysec` and `no` [45] there is no noticeable difference. Only with the most aggressive sync option (`always`) are improvements noticeable. Again, the file granularity order of 0 shows very minimal differences. At a file granularity order of 4, slight improvements are noticeable. However, compared to the improvements at a file granularity order of 9, the differences are still negligible. At a file range granularity of 9, we can observe a $4.62\times$ improvement with an upgrade threshold of 60, a downgrade threshold of 30, and a write threshold of 5 over the default case.

Nonetheless, the `always` mode still lacks behind `everysec` and `no` in terms of total throughput. This can be seen in the sample run in Figure 6.4. While `everysec` and `no` perform similarly, `always` with transparent DAX upgrades manages to reach about 60% of the requests per second of the `no` sync mode. Compared to the `default` case, reaching 60% of the maximal throughput makes the `alawys` significantly more usable.

However, `explicit` again comes out second, even though we would expect it to perform best. The pattern is identical to the one we observe in Figure 6.2, suggesting that the Valkey storage backend is at responsibility for both cases.

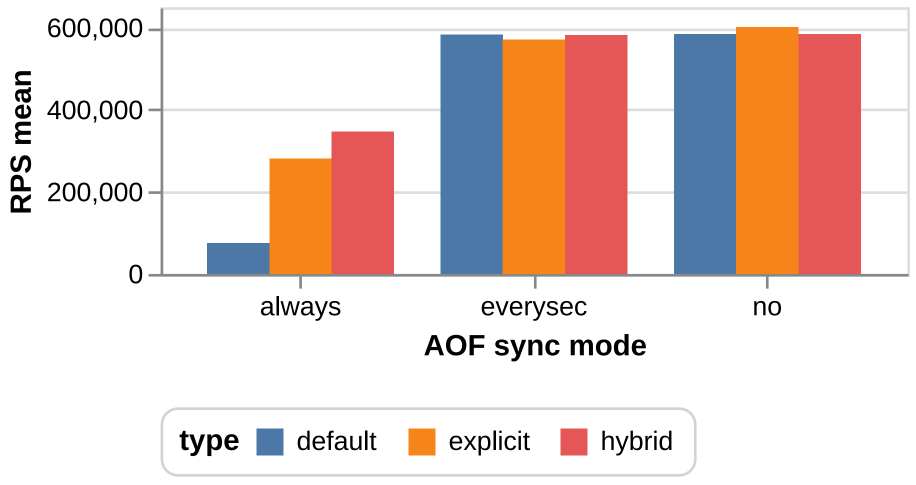

Figure 6.4: Valkey benchmark run with a pipeline size of 10, upgrade threshold: 60, downgrade threshold: 30, write threshold: 5, file granularity order: 9. In the `always` sync mode there is a $4.62\times$ speedup of the `hybrid` case compared to the `default` case. `explicit` falls behind `hybrid` by about 20%. All three similarly with the `everysec` and `no` sync modes.

The reason we do not see any significant difference when using `everysec` with either pipeline size is in the definition of Valkey's persistence method. Primarily, `fsync` calls are too rare compared to the writes when only writing back every second. That alone causes a file range to rarely reach the upgrade threshold. Additionally, due to the nature of an AOF, upgrading earlier file ranges does not make a difference anymore if the process is already writing many blocks ahead.

Thus, we need to upgrade the file range we are still writing to in order to see meaningful improvements. This can only be ensured by big file ranges and/or aggressive `fsync` calls like the `always` mode that synchronizes after every write.

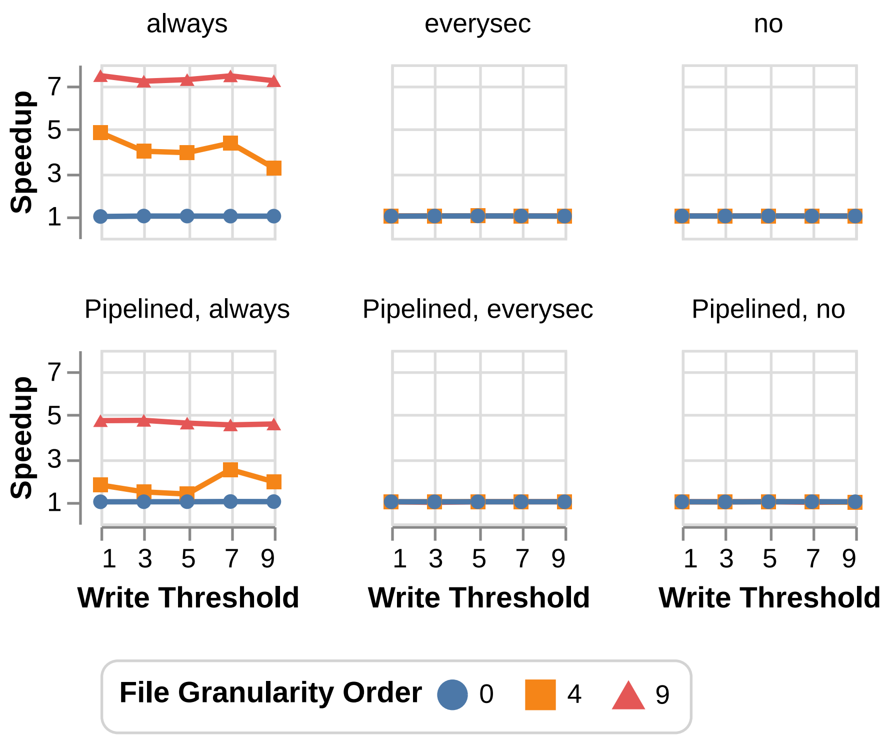

Figure 6.5: Overview of Valkey benchmark runs with an upgrade threshold of 60, downgrade threshold of 30, and a write threshold of 5. The shown values are relative to the `default` case. The x- and y-axis represent the upgrade threshold and speedup, respectively. The plots are split vertically by downgrade threshold, and horizontally by AOF sync mode. With the `always` sync mode and a pipeline size of 1 the file granularity order 4 decreases with an increasing write threshold. In all other cases the speedups are consistent across all write thresholds.

Figure 6.5 shows how varying the write threshold affects the throughput. The runs are done with an upgrade threshold of 60 and a downgrade threshold of 30. In the case with a pipeline length of 1, a downwards trend is noticeable with a file granularity order of 4. Especially a write threshold greater than 7 causes the speedup to drop by a factor of 2. This again illustrates that, especially in this test, we need to upgrade file ranges, and thus reach the upgrade threshold, as early as possible. `everysec` and `no`, again, show no difference with either a pipeline size of 1 or 10.

6.2.3 Postgres

We test PostgreSQL with `pgbench` [30] and the TPC-B-alike scenario. Notably, this benchmark is a mixed workload with reads and writes, contrary to the Valkey one. We run the benchmarks with a scaling factor of 70 which results in roughly $7,000,000$ rows. Furthermore, `pgbench` is split up into an initialization phase and the actual benchmark. During the initialization phase the tables get created and pre-filled. As a result this is pretty write-heavy and benefits from transparent DAX upgrades. However, that access pattern is similar to what we investigate with RocksDB, so we decided to leave out those results here. Thus, we only concentrate on the real-world workload, ignoring pre-seeding.

We execute the benchmarks with three different postgres configurations: `wal_sync_method={fdatasync,fsync}` and `fsync=off`. Figure 6.6 provides an overview over postgres test runs with a write threshold of 5.

As expected there are no differences with `fsync=false`. In the `Fsync` case there are no relevant changes noticeable either. Presumably that is because the amount of metadata that needs to be written back and does not benefit from transparent DAX upgrades is too big. The syscall `fsync` flushes the data pages to the disk [14]. Additionally, it also flushes all modified file metadata, which in turn may trigger file system journal writebacks if journaling is enabled [14, 22]. The journal and file system metadata are not on our hybrid SSD in theses tests.

The syscall `fdatasync` on the other hand flushes data pages and only the file metadata that is absolutely necessary for future reads [11]. As such, a larger part of the data that is flushed benefits from being upgraded to our hybrid SSD and the overall operation becomes faster. Especially in the mixed workload we have here, this appears to make a significant difference. The metadata writeback likely becomes the limiting factor, rendering the performance improvements of the data writeback irrelevant. Unfortunately, we did not have time to validate this.

Finally, `FdataSync` shows noticeable improvements. Similar to Valkey, a file granularity order of 0 does not show changes here either. At a file granularity order of 4 we can observe a roughly $2\times$ speedup. Notably, the results are the best with a downgrade threshold of 30, although with a small difference. Further, it falls to about a 50% improvement with the larger downgrade thresholds 50 and 70. This might indicate that downgrading too early has a negative impact. With a file granularity order of 9, results are overall improved.

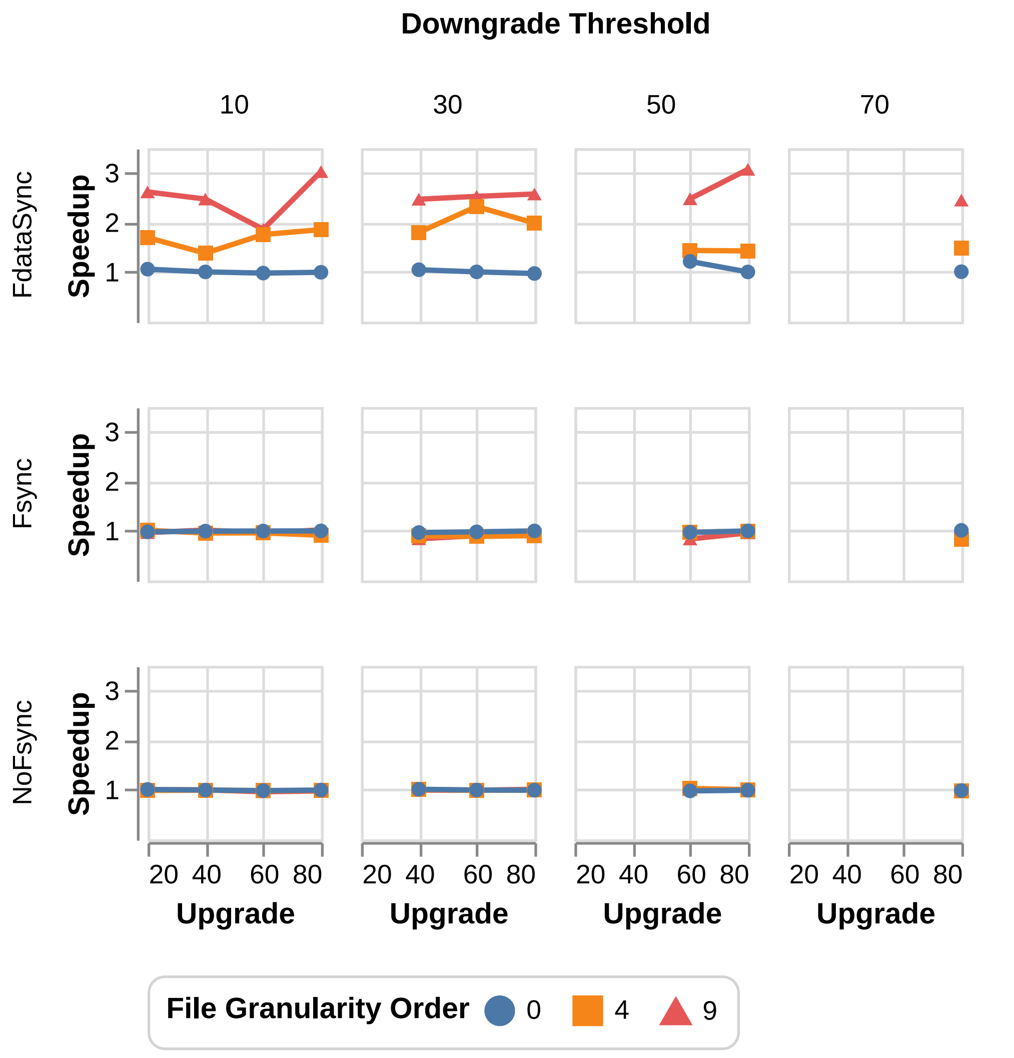

Figure 6.6: Overview of Postgres benchmark runs with a write threshold of 5. The shown values are relative to the `default` case. The x- and y-axis represent the upgrade threshold and speedup, respectively. The plots are split vertically by downgrade threshold, and horizontally by Postgres sync variants. `NoFsync` does not show any changes. `Fsync` shows minor differences in both a positive and negative direction. `FdataSync` also shows slowdowns with a write threshold of 30%. Only with the highest upgrade threshold of 80% there is a minor improvement noticeable across all downgrade thresholds but 30%.

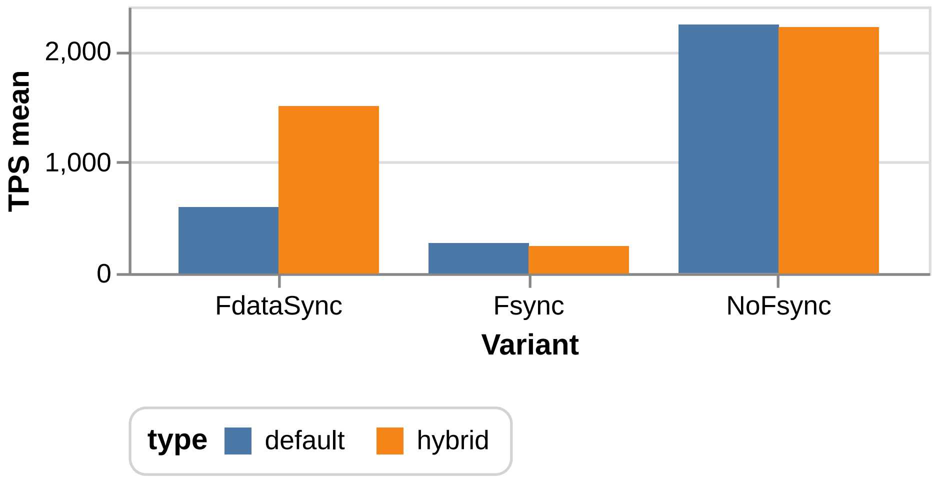

Figure 6.7: Postgres benchmark run with an upgrade threshold of 60, downgrade threshold of 30, write threshold of 5, and a file granularity order of 9. A significant speedup in the `FdataSync` case is noticeable. It reaches about 67% of the performance of the `NoFsync` case.

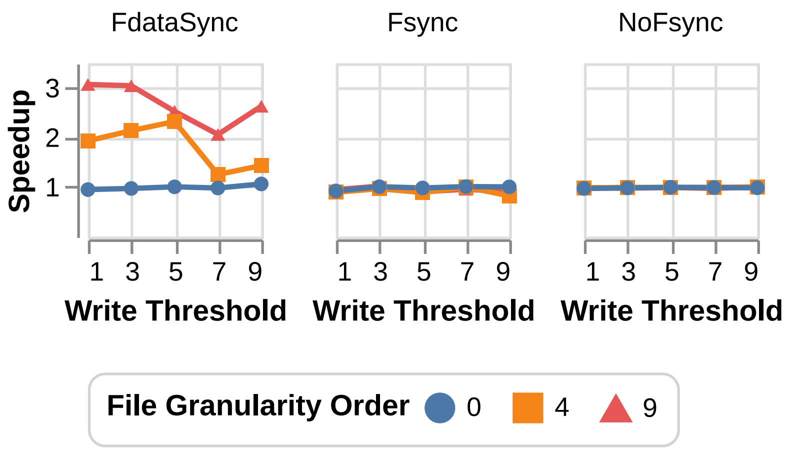

Figure 6.8: Postgres benchmark run with an upgrade threshold of 60 and a downgrade threshold of 30. `Fsync` and `NoFsync` do not any differences. `FdataSync` shows a drop for write thresholds greater than 5.

The run with an upgrade threshold of 60%, a downgrade threshold of 30%, and a file granularity order of 9 this time yields a $2.53\times$ speedup for `FdataSync`. It is shown in detail in Figure 6.7. `FdataSync` with transparent DAX upgrades manages to achieve about 67% of the performance of the `NoFsync` run. Furthermore it can be seen how even the `default` `FdataSync` case is significantly faster than doing entire `fsync` calls.

Lastly, looking at the write thresholds in Figure 6.8 we observe similar patterns. Across all write thresholds the `Fsync` and `NoFsync` variants do not show any changes. In `FdataSync` we observe a drop in throughput for write thresholds greater than 5. Presumably this results from the structure of the workload. Contrary to Valkey, the PostgreSQL benchmark we use is a mixed workload including both reads and writes, contrary to an exclusive write workload. This may cause individual file ranges to not reach as many writes in total. Consequently, less file ranges that may benefit from being on a hybrid SSD get upgraded.

6.2.4 RocksDB

Initially, we planned on testing RocksDB with the popular Yahoo! Cloud Serving Benchmark (YCSB) [53]. However, we found out that YCSB does not support synchronous writes with RocksDB out of the box. As such, the benchmarks cannot benefit from our transparent DAX upgrades, which we confirmed in testing. Thus, we use the built-in benchmarks from the `db_bench` utility shipped with RocksDB.

We use the `fillseq` and `fillsync` benchmarks that use asynchronous and synchronous writes, respectively, to fill up a database. We consider the `fillseq` performance the best possible we can achieve, similar to Valkey's `no` sync mode.

RocksDB allows to configure `use_fsync`. It defaults to `false`, meaning it will use `fdatasync` instead of `fsync`. We run benchmarks with `fillseq`, `fillsync`, and `fillsync` combined with `use_fsync=true`, which we refer to by `Default`, `FdataSync`, and `Fsync`, respectively.

Figure 6.9 gives an overview of our test runs on RocksDB with a write threshold set to 5. Unsurprisingly, the asynchronous writes of the *Default* case result in no changes with our implementation. Similar to the other two benchmarks, we also see significant improvements in the *FdataSync* case here. In fact, the speedup is the greatest out of all the benchmarks we run. With an upgrade threshold of 60, a downgrade threshold of 30, and a file granularity order of 9, we observe a $24.47\times$ speedup. Furthermore, we notice that with a file granularity order of 0 we again see essentially zero difference. A file granularity order of 9 also ends significantly above the order of 4, doubling the speedup with a downgrade threshold of 10%. Interestingly, `Fsync` does not show any significant difference ($< 1\%$). The responsible flag `use_fsync` is only being passed to the writeback functions and does not result in RocksDB taking any other branches [34]. Thus, we assume that the *Fsync* mode is as slow because there is dirty metadata that also needs to be written back and does not end on the hybrid SSD, resulting in slow writebacks. In subsection 6.2.3 we saw similar results for Postgres. The performance difference can likely be attributed to the specific data both applications write back. However, we did not find the time to confirm nor deny this hypothesis with I/O analysis.

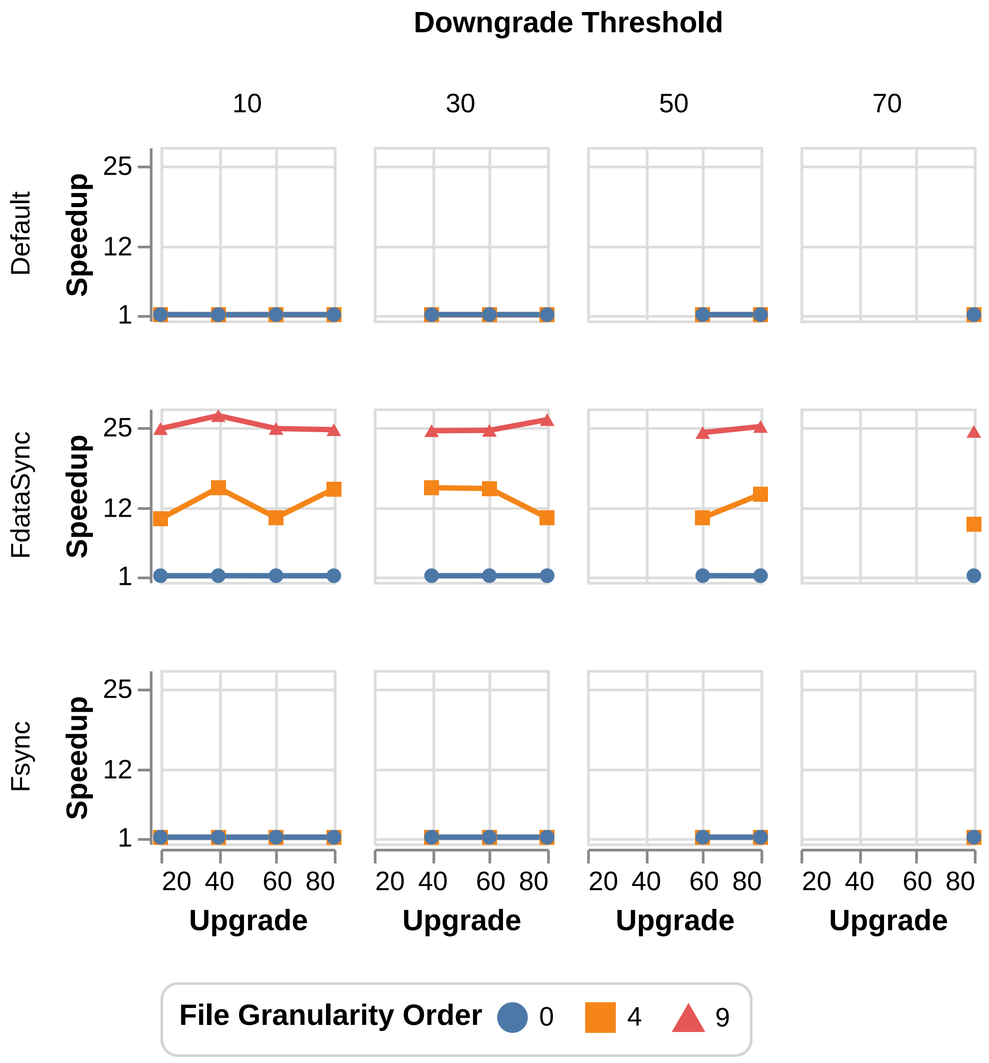

Figure 6.9: Overview of RocksDB benchmark runs with a write threshold of 5. The shown values are relative to the `default` case. The x- and y-axis represent the upgrade threshold and speedup, respectively. The plots are split vertically by downgrade threshold, and horizontally by variant. The *Default* and *Fsync* cases do not show any significant speedups. In the *Fdatasync* case there are major improvements for the file granularity orders 4 and 9. With a file granularity order of 9 we reach a $24.47\times$ speedup.

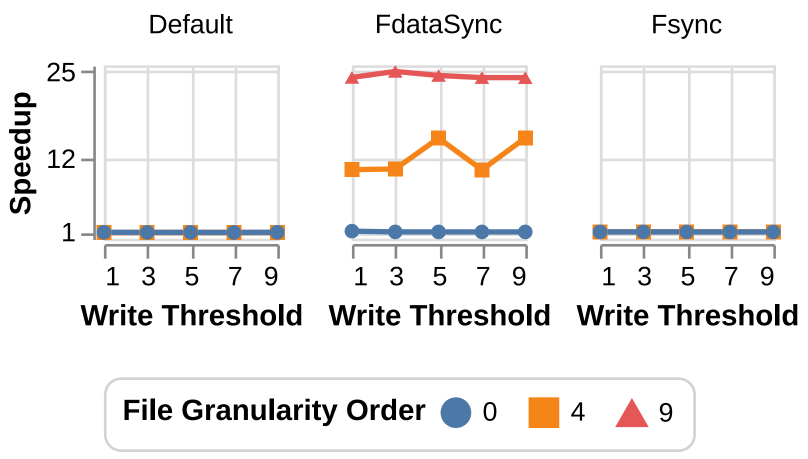

Figure 6.10: Overview of RocksDB benchmark runs with an upgrade threshold of 60 and a downgrade threshold of 30. The *Default* and *Fsync* cases do not show any significant speedups. In the *Fdatasync* case the file granularity order 9 is stable across write thresholds, while the order 4 shows small variance.

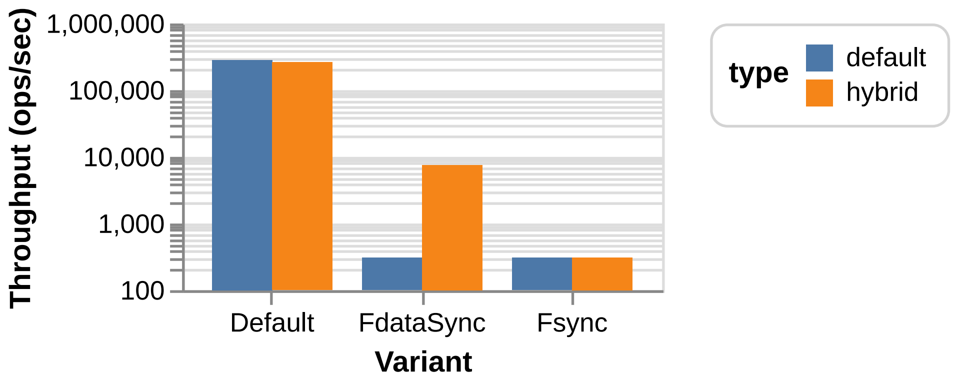

Figure 6.11: RocksDB benchmark run with an upgrade threshold of 60, downgrade threshold of 30, write threshold of 5, and a file granularity order of 9. Although the speedup of `hybrid` in the `FdataSync` case is significant, the roughly $7,500$ operations per second do not come near the roughly $280,000$ operations per second the asynchronous writes in the `Default` case achieve.

Looking at the write threshold development in Figure 6.10, we unsurprisingly do not see any differences for *Default*. For *FdataSync*, the file granularity order 9 is fairly stable, hovering at the $25\times$ speedup. A file granularity order of 4 shows minor variance. *Fsync* does not show significant changes either. Overall, a write threshold of 5 or 7 appears to be an appropriate choice here, too.

Finally, Figure 6.11 shows the absolute throughput of the various variants. The asynchronous writes in the *Default* case greatly outperform the synchronous writes, even with our hybrid implementation. With FdataSync we reach a throughput of approximately 7,500 operations per second in the hybrid case. Contrarily, with asynchronous writes RocksDB reaches a throughput of over 280,000 operations per second.

6.3 Discussion

We have seen how our implementation behaves under various workloads. Overall, we are able to improve performance of applications with strong persistence requirements across all our benchmarks. Specifically, the changes range from an approximately $2.53\times$ improvement in postgres via a $7.29\times$ speedup in the Valkey workload up to a $24.47\times$ speedup in RocksDB.

The benchmarks reveal a couple of insights into the configuration options we allow setting. First, the write threshold helps with algorithm stability but it also should not be set too high. We assumed that it would be helpful since we avoid dividing small numbers, where a single increment can significantly impact the ratio. The plots show that a large write threshold leads to degraded performance in both Valkey and PostgreSQL.

Second, the upgrade and downgrade thresholds show unexpectedly little impact in our workloads. While it becomes apparent that very small upgrade thresholds may upgrade file ranges prematurely, the specific value is not as important. A value around 60% works well for our workloads. For the downgrade thresholds, we also could not observe a clear sweet spot. This is likely to, in parts, be due to the write-focus of most of our benchmarks. In Postgres a downgrade threshold of approx. 30% works fine in combination with the 60% upgrade threshold and a write threshold of 5.

Lastly, the file granularity order shows significant impact. In all benchmarks we see that a file granularity order of 0 shows little to no effect. Page-granular tracking and upgrading is too fine. With the processing time and delay of the asynchronous job we queue, the upgrade happens too late – if the individual page even reaches the upgrade threshold. At a page granularity order of 4, improvements start to become apparent. A page granularity order of 4 implies a file range size of $2^{12}\ \mathrm{B} \cdot 2^4 = 2^{16}\ \mathrm{B} = 64\,\mathrm{KiB}$, assuming $2^{12}\ \mathrm{B} = 4\,\mathrm{KiB}$ pages. Setting the page granularity order to 9 yields even better results, especially in Valkey. The Valkey workload is a consistent write stream, so ideally all pages are already upgraded. Upgrading more pages in advance helps with performance here. Furthermore, the other two benchmarks also show better performance compared to an order of 4.

Overall, our policy works well with the workloads we tested. However, the upgrade and downgrade thresholds show unexpectedly little difference. This might indicate that our chosen metric is not the most precise. Either it exceeds all upgrade thresholds greatly, or it does not even hit the lowest. While that is to be expected for the Valkey and RocksDB workloads given their access patterns, we would have expected greater variation in Postgres. Having a more sophisticated metric might be helpful and yield better results, especially for Postgres.

Chapter 7

Future Work

Our approach is rather flexible and provides lots of options to expand and modify it. Configuring these options dynamically at runtime could be interesting and will be briefly discussed in section 7.1. Furthermore, we do not consider storage pressure at the moment and none of our benchmarks cover that. Lastly, section 7.3 touches on our policy and talks about more sophisticated policies and evaluating those.

7.1 Dynamic Thresholds

In our work we expose these three parameters through `sysfs`.

- Write threshold
- Upgrade threshold
- Downgrade threshold

They can all be dynamically altered at runtime. As of now, however, they are all treated as being static. It would be interesting to explore dynamically modifying the thresholds in user space, depending on the current workload. Assuming workloads on a system vary over time, this could be a way to use more optimal settings for each of those workloads. This would allow to maximize the performance of applications if specific patterns are recognized. For instance, if a stream of writes is detected, it can be assumed that all pages should be upgraded and as such thresholds could be lowered. Similarly, if a mixed workload is recognized, as we have seen with PostgreSQL, more conservative thresholds could be chosen to avoid upgrading file ranges that would suffer from being upgraded. In addition, this could also be further enhanced to have different thresholds for different memory regions. By enabling that, different workloads could benefit from optimal thresholds at the same time.

Furthermore, space utilization could be incorporated in the analysis. If the hybrid SSD's DRAM is nearly full, it may make sense to raise thresholds. This would cause fewer file ranges to be upgraded, leaving room on the hybrid device for explicit DAX mappings.

We also have static parameters set at compile time. Specifically, there is a delay until the write and writeback counts decay. We did not have the time to evaluate our delay parameters, but instead tried to pick values that seemed sensible. It would be interesting to evaluate different delays with different workloads to figure out which delays work best. Potentially they could also be dynamically configurable and as such adapted in user space during run time if the workload changes.

7.2 Storage Pressure

Related to explicit DAX mappings, our implementation currently does not prioritize those. That is, our implementation could fill up the entire hybrid SSD with transparent DAX mappings, causing explicit mappings to fail. It arguably makes sense that explicit DAX mappings should get prioritized, as those can be assumed to have guaranteed strong persistence requirements.

One mitigation strategy for this would be to allow transparent DAX mappings to only use up a part of the entire available storage. By preserving a safe area for explicit mappings, the chance of transparent DAX upgrades taking up all storage becomes less likely. However, in theory that reserved space could be filled up by explicit mappings and the rest by transparent ones.

To also accommodate for that issue, it would be necessary to downgrade transparent DAX upgrades if an explicit mapping cannot be established due to space constraints. For that an eviction policy would be necessary. Trivially, oldest upgraded file range could be downgraded. However, that would potentially be a suboptimal decision. Instead, it would be more interesting to consider the metric for the upgrade decision and based on that evaluate the file range with the weakest persistence requirement.

7.3 Policy

In this thesis we are proposing a rather simple policy made up of the ratio between the write and writeback count per file range. It would be worthwhile to explore different, possibly more complex, policies that may also include other metrics.

Specifically, we saw that our implementation does not always make the best decision, or takes too long to realize a pattern. Possibly, a more sophisticated policy will be able to perform better in those cases. It might be able to more cleanly distinguish various access pattern such as a mixed read/write workload in Postgres and a consistent write stream in Valkey. Additionally, a different policy may allow using a smaller file granularity order. This would allow the implementation to more granularly upgrade specific pages.

Furthermore, it would be interesting to benchmark the reactivity of our policy and implementation. This would split up into (1) measuring how long it takes our metric to pick up on, e.g., a consistent write stream, and (2) how long the upgrade itself takes, once decided. By knowing its reactivity, we might be able to pick better delay parameters for the decay. If we would realize that the policy is extremely slow to react, it might also make sense to pick a different Workqueue in the hopes that jobs get queued earlier.

Chapter 8

Conclusion

In a world of omnipresent compute, high-speed, low-latency storage becomes of increasing relevance. Historically, PM such as Intel Optane provides low latencies known from DRAM with persistence properties. However, this special memory is expensive and takes up SODIMM slots for comparatively little storage. As a solution, CXL-based hybrid SSDs have evolved. They provide both a fine-granular interface, supported by on-device DRAM that is backed by capacitors, as well as a block-granular interface. Up until now, its usefulness required explicit application support by mapping files to that CXL device.

Our work lifts this requirement, enabling *Transparent DAX Upgrades* for any application. On a configurable file range we track writebacks and writes. Based on these metrics, we implemented a policy for deciding when to upgrade or downgrade a file range. We consider the ratio between the writeback and write count, and compare that against an upgrade threshold and downgrade threshold. To make our algorithm temporal locality-aware, we add a decay to both counters. This avoids file ranges with a slow but consistent stream of writebacks eventually getting improperly upgraded.

We tested our implementation with benchmarks of Valkey, PostgreSQL, and RocksDB. Due to a lack of production hybrid SSD, we tested our implementation with a CXL memory expander and an emulated hybrid SSD using that CXL memory and a regular NVMe SSD. Across the entire suite we saw significant improvements. In Valkey, for instance, we measured a $7.29\times$ speedup from using our kernel patch. No application modifications were necessary. In RocksDB, we even achieved a $24.47\times$ speedup.

We conclude that our work yields significant improvements for data center typical workloads such as databases. Without the need to modify the applications or change any config, our kernel patch immediately results in performance improvements when using a hybrid SSD. In the case of Valkey for instance, our approach might actually make the `always` AOF sync mode more feasible. With our solution the difference to the rarer sync modes shrinks to less than 50%. However, we also note that Khalil et al. [23] are able to show even more impressive results with Transparent DAX Mappings by bypassing the kernel entirely. At the same time, our implementation is simpler and easier to reason about, while still exposing relevant config options to the user space to dynamically affect upgrading behavior, if necessary.

Bibliography

[1] *18.2. Creating a Database Cluster*. Aug. 2025. URL: https://www.postgresql.org/docs/17/creating-cluster.html (visited on 08/27/2025).

[2] *19.5. Write Ahead Log*. May 2025. URL: https://www.postgresql.org/docs/17/runtime-config-wal.html (visited on 05/24/2025).

[3] *4th Gen Intel Xeon Scalable Processors Product Specifications*. URL: https://www.intel.com/content/www/us/en/products/sku/series/228622/4th-gen-intel-xeon-scalable-processors.html (visited on 09/04/2025).

[4] *Announcing PlanetScale for Postgres — PlanetScale*. July 2025. URL: https://planetscale.com/blog/planetscale-for-postgres (visited on 08/13/2025).

[5] Duck-Ho Bae et al. „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)*. June 2018, pp. 425–438. DOI: 10.1109/ISCA.2018.00043. URL: https://ieeexplore.ieee.org/document/8416845 (visited on 05/15/2025).

[6] Shuhan Bai et al. „Pipette: Efficient Fine-Grained Reads for SSDs." In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. San Francisco California: ACM, July 2022, pp. 385–390. ISBN: 978-1-4503-9142-9. DOI: 10.1145/3489517.3530467. URL: https://dl.acm.org/doi/10.1145/3489517.3530467 (visited on 05/15/2025).

[7] *CXL® Specification - Compute Express Link*. Sept. 2023. URL: https://computeexpresslink.org/wp-content/uploads/2024/02/CXL-3.1-Specification.pdf (visited on 05/15/2025).

[8] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. „An Introduction to the Compute Express Link (CXL) Interconnect." In: *ACM Comput. Surv.* 56.11 (July 2024), 290:1–290:37. ISSN: 0360-0300. DOI:

10.1145/3669900. URL: https://dl.acm.org/doi/10.1145/3669900 (visited on 05/15/2025).

[9] *Direct Access for Files — The Linux Kernel Documentation*. URL: https://docs.kernel.org/6.14/filesystems/dax.html (visited on 05/24/2025).

[10] *Docker: Accelerated Container Application Development*. Apr. 2025. URL: https://www.docker.com/ (visited on 08/27/2025).

[11] *Fdatasync(2) - Linux Man Page*. URL: https://linux.die.net/man/2/fdatasync (visited on 08/13/2025).

[12] *Fedora Linux*. URL: https://www.fedoraproject.org/ (visited on 08/27/2025).

[13] *Filebench/Workloads/Varmail.f at Master · Filebench/Filebench*. URL: https://github.com/filebench/filebench/blob/master/workloads/varmail.f (visited on 08/07/2025).

[14] *Fsync(2) - Linux Man Page*. URL: https://linux.die.net/man/2/fsync (visited on 08/13/2025).

[15] Bill Gervasi and San Chang. „NVMe Over CXL™ Defines Memory Class Storage to Improve System Performance.“ In: ().

[16] Mel Gorman. „Understanding The Linux Virtual Memory Manager.“ In: (Feb. 2004). URL: https://www.eecg.utoronto.ca/~yuan/teaching/archive/ece344_2017w/linux-vmm.pdf.

[17] PostgreSQL Global Development Group. *PostgreSQL*. May 2025. URL: https://www.postgresql.org/ (visited on 05/24/2025).

[18] Daniel Habicht et al. „Fundamental OS Design Considerations for CXL-based Hybrid SSDs.“ In: *Proceedings of the 2nd Workshop on Disruptive Memory Systems*. DIMES ’24. New York, NY, USA: Association for Computing Machinery, Nov. 2024, pp. 51–59. ISBN: 979-8-4007-1303-3. DOI: 10.1145/3698783.3699380. URL: https://dl.acm.org/doi/10.1145/3698783.3699380 (visited on 05/15/2025).

[19] Daniel Habicht et al. „Rethinking Storage I/O for Hybrid NVMe and DAX Block Devices.“ MA thesis. Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, July 2024. URL: https://os.itec.kit.edu/downloads/2024_MA_Habicht_Hybrid_Storage_IO.pdf.

[20] *Integrators List - Compute Express Link*. Oct. 2023. URL: https://computeexpresslink.org/integrators-list/ (visited on 05/23/2025).

[21] *Intel® Optane(TM) Persistent Memory (PMem)*. URL: `https://www.intel.com/content/www/us/en/products/details/memory-storage/optane-dc-persistent-memory.html` (visited on 06/21/2025).

[22] *Journal (Jbd2) — The Linux Kernel Documentation*. URL: `https://www.kernel.org/doc/html/latest/filesystems/ext4/journal.html` (visited on 05/29/2025).

[23] Yussuf Khalil et al. „Transparent DAX Mappings: Towards Automatic Kernel Bypass with CXL-Based Hybrid SSDs.“ In: *Proceedings of the 3rd Workshop on Disruptive Memory Systems*. DIMES ’25. New York, NY, USA: Association for Computing Machinery, Oct. 2025, pp. 54–62. ISBN: 979-8-4007-2226-4. DOI: `10.1145/3764862.3768178`. URL: `https://doi.org/10.1145/3764862.3768178` (visited on 10/04/2025).

[24] Jeongho Lee et al. „srNAND: A Novel NAND Flash Organization for Enhanced Small Read Throughput in SSDs.“ In: *IEEE Computer Architecture Letters* (2025), pp. 1–4. ISSN: 1556-6064. DOI: `10.1109/LCA.2025.3571321`. URL: `https://ieeexplore.ieee.org/document/11006506/` (visited on 05/27/2025).

[25] Philip Levis, Kun Lin, and Amy Tai. „A Case Against CXL Memory Pooling.“ In: *Proceedings of the 22nd ACM Workshop on Hot Topics in Networks*. HotNets ’23. New York, NY, USA: Association for Computing Machinery, Nov. 2023, pp. 18–24. ISBN: 979-8-4007-0415-4. DOI: `10.1145/3626111.3628195`. URL: `https://dl.acm.org/doi/10.1145/3626111.3628195` (visited on 05/23/2025).

[26] Shaobo Li et al. „ByteFS: System Support for (CXL-based) Memory-Semantic Solid-State Drives.“ In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS ’25. New York, NY, USA: Association for Computing Machinery, Mar. 2025, pp. 116–132. ISBN: 979-8-4007-0698-1. DOI: `10.1145/3669940.3707250`. URL: `https://dl.acm.org/doi/10.1145/3669940.3707250` (visited on 05/19/2025).

[27] Zheng Li et al. „Performance Overhead Comparison between Hypervisor and Container Based Virtualization.“ In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. Mar. 2017, pp. 955–962. DOI: `10.1109/AINA.2017.79`. URL: `https://ieeexplore.ieee.org/document/7921010/` (visited on 10/03/2025).

[28] *Manuals for Intel® 64 and IA-32 Architectures.* URL: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html (visited on 09/04/2025).

[29] *Page-Writeback.c « Mm - Kernel/Git/Stable/Linux.Git - Linux Kernel Stable Tree.* URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/mm/page-writeback.c?h=v6.6.108#n2355 (visited on 09/26/2025).

[30] *Pgbench.* May 2025. URL: https://www.postgresql.org/docs/17/pgbench.html (visited on 08/06/2025).

[31] Paul Alcorn published. *AMD Unveils Zen 4 CPU Roadmap: 96-Core 5nm Genoa in 2022, 128-Core Bergamo in 2023.* Nov. 2021. URL: https://www.tomshardware.com/news/amd-unveils-zen-4-cpu-roadmap-96-core-5nm-genoa-128-core-begamo (visited on 09/04/2025).

[32] Redis. *Redis - The Real-time Data Platform.* URL: https://redis.io/ (visited on 08/13/2025).

[33] *Redis Persistence.* URL: https://redis.io/docs/latest/operate/oss_and_stack/management/persistence/ (visited on 05/24/2025).

[34] *RocksDB | A Persistent Key-Value Store.* URL: http://rocksdb.org/ (visited on 08/01/2025).

[35] *RocksDB: Benchmarking Tools.* URL: https://github.com/facebook/rocksdb/wiki/Benchmarking-tools (visited on 08/12/2025).

[36] Yizhou Shan, Shin-Yeh Tsai, and Yiying Zhang. „Distributed Shared Persistent Memory.“ In: *Proceedings of the 2017 Symposium on Cloud Computing.* SoCC '17. New York, NY, USA: Association for Computing Machinery, Sept. 2017, pp. 323–337. ISBN: 978-1-4503-5028-0. DOI: 10.1145/3127479.3128610. URL: https://dl.acm.org/doi/10.1145/3127479.3128610 (visited on 05/23/2025).

[37] Debendra Das Sharma. „Compute Express Link (CXL): Enabling Heterogeneous Data-Centric Computing With Heterogeneous Memory Hierarchy.“ In: *IEEE Micro* 43.2 (Mar. 2023), pp. 99–109. ISSN: 0272-1732. DOI: 10.1109/MM.2022.3228561. URL: https://doi.org/10.1109/MM.2022.3228561 (visited on 05/16/2025).

[38] *Time(7) - Linux Manual Page.* URL: https://man7.org/linux/man-pages/man7/time.7.html (visited on 08/14/2025).

[39] *TPC-B Homepage*. URL: https://www.tpc.org/tpcb/ (visited on 08/06/2025).

[40] *Ubuntu Manpage: Pg_createcluster - Create a New PostgreSQL Cluster*. URL: https://manpages.ubuntu.com/manpages/noble/man1/pg_createcluster.1.html (visited on 08/27/2025).

[41] Musa Unal et al. „Tolerate It If You Cannot Reduce It: Handling Latency in Tiered Memory.“ In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. Banff AB Canada: ACM, May 2025, pp. 50–57. ISBN: 979-8-4007-1475-7. DOI: 10.1145/3713082.3730376. URL: https://dl.acm.org/doi/10.1145/3713082.3730376 (visited on 10/03/2025).

[42] USENIX Association. *Proceedings of the FREENIX Track, 2004 USENIX Annual Technical Conference: June 30 - July 02, 2004, Boston, MA, USA*. Berkeley, Calif: USENIX Assiation, 2004. ISBN: 978-1-931971-22-5.

[43] *Valkey*. URL: https://valkey.io/ (visited on 05/24/2025).

[44] *Valkey Documentation · Benchmarking Tool*. URL: https://valkey.io/topics/benchmark/ (visited on 10/03/2025).

[45] *Valkey Documentation · Persistence*. URL: https://valkey.io/topics/persistence/ (visited on 05/24/2025).

[46] Alexander van Renen et al. „Persistent Memory I/O Primitives.“ In: *Proceedings of the 15th International Workshop on Data Management on New Hardware*. DaMoN'19. New York, NY, USA: Association for Computing Machinery, July 2019, pp. 1–7. ISBN: 978-1-4503-6801-8. DOI: 10.1145/3329785.3329930. URL: https://dl.acm.org/doi/10.1145/3329785.3329930 (visited on 05/23/2025).

[47] Haris Volos, Andres Jaan Tack, and Michael M. Swift. „Mnemosyne: Lightweight Persistent Memory.“ In: *SIGARCH Comput. Archit. News* 39.1 (Mar. 2011), pp. 91–104. ISSN: 0163-5964. DOI: 10.1145/1961295.1950379. URL: https://dl.acm.org/doi/10.1145/1961295.1950379 (visited on 05/23/2025).

[48] *Which Generation of Intel Xeon Scalable Processors Support Compute...* URL: https://www.intel.com/content/www/us/en/support/articles/000059219.html (visited on 09/04/2025).

[49] *Workqueue — The Linux Kernel Documentation*. URL: https://docs.kernel.org/core-api/workqueue.html (visited on 07/16/2025).

[50] *Write Ahead Log (WAL)*. URL: https://github.com/facebook/rocksdb/wiki/Write-Ahead-Log-(WAL) (visited on 10/03/2025).

[51] *XArray — The Linux Kernel Documentation.* URL: `https://docs.kernel.org/core-api/xarray.html` (visited on 08/15/2025).

[52] Zhe Yang et al. „CoinPurse: A Device-Assisted File System with Dual Interfaces.“ In: *2020 57th ACM/IEEE Design Automation Conference (DAC).* July 2020, pp. 1–6. DOI: `10.1109/DAC18072.2020.9218679`. URL: `https://ieeexplore.ieee.org/abstract/document/9218679` (visited on 06/09/2025).

[53] *YCSB/Rocksdb/README.Md at Master · Brianfrankcooper/YCSB.* URL: `https://github.com/brianfrankcooper/YCSB/blob/master/rocksdb/README.md` (visited on 05/25/2025).

[54] Seung Won Yoo et al. „DJFS : Directory-Granularity Filesystem Journaling for CMM-H SSDs.“ In: *23rd USENIX Conference on File and Storage Technologies (FAST 25).* 2025, pp. 35–51. ISBN: 978-1-939133-45-8. URL: `https://www.usenix.org/conference/fast25/presentation/yoo` (visited on 05/19/2025).

[55] Jianping Zeng et al. *Performance Characterizations and Usage Guidelines of Samsung CXL Memory Module Hybrid Prototype.* Mar. 2025. DOI: `10.48550/arXiv.2503.22017`. arXiv: `2503.22017 [cs]`. URL: `http://arxiv.org/abs/2503.22017` (visited on 05/17/2025).

[56] Yekang Zhan et al. „RomeFS: A CXL-SSD Aware File System Exploiting Synergy of Memory-Block Dual Paths.“ In: *Proceedings of the 2024 ACM Symposium on Cloud Computing.* SoCC ’24. New York, NY, USA: Association for Computing Machinery, Nov. 2024, pp. 720–736. ISBN: 979-8-4007-1286-9. DOI: `10.1145/3698038.3698539`. URL: `https://dl.acm.org/doi/10.1145/3698038.3698539` (visited on 05/19/2025).