Karlsruhe Institute of Technology

Kernel-Based TDM Management

Bachelor's Thesis
submitted by

cand. inform. Florian Leander Singer

to the KIT Department of Informatics

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

April 27th, 2026 – August 27th, 2026

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

Karlsruhe, August 27, 2026

Abstract

High-performance storage I/O operations play an important role in system architecture. Sufficiently sized I/O-heavy database systems have begun to reach throughput requirements that can not be provided by traditional operating systems I/O mechanisms. This is due to overhead from required system calls, the resulting context switches, and the commonly necessary copy operations within the page cache. This work aims to address this inefficiency for CXL-based hybrid SSDs, by implementing a mechanism which will transparently redirect eligible I/O operations to directly accessible persistent memory, bypassing the kernel. This results in higher throughput without having to alter the code of existing applications.

The work is comprised of a kernel subsystem which analyzes file bandwidth statistics and opportunistically maps suitable regions directly into the address space of applications, based on a caching policy which evaluates the collected statistics. A userspace component will transparently redirect I/O operations to the kernel bypass path if a mapping is available, falling back to the conventional system call interface if it is not. Due to the current lack of commercially available hybrid SSDs, Intel Optane persistent memory was chosen as a stand in, as it is still widely available. With Optane, our approach was able to achieve an increased throughput of 135% in the *TPC-C* benchmark.

Contents

Abstract **v**

Contents **vii**

1 Introduction **1**

2 Background **3**
2.1 Persistent Memory Devices 3
2.1.1 Intel Optane 4
2.1.2 Hybrid SSDs 4
2.2 Linux Kernel 6
2.2.1 ext4 7
2.2.2 DAX 7
2.3 x86_64 Architecture Features 8
2.3.1 The syscall instruction 8
2.3.2 Atomic Operations 9

3 Related Work **11**
3.1 SplitFS 11
3.2 Hybrid Storage 12
3.3 Transparent DAX Mappings 12

4 Design and Implementation **15**
4.1 Kernel Subsystem 15
4.1.1 procfs interface 18
4.1.2 file tdm_table 20
4.1.3 daxdev 20
4.1.4 tdm_qw workqueue 21
4.2 Userspace Library 21
4.3 Caching Policy 24

5 Evaluation **27**
5.1 Methodology 27

5.2 Benchmarks 28
5.2.1 diskv 29
5.2.2 TPC-C on SQLite 30
5.2.3 YCSB on Valkey 32
5.2.4 Valkey SET 34
5.3 Caching Policy 35
5.4 Discussion 38

6 Future Work **41**
6.1 Necessary Improvements 41
6.2 Move Toward Common Interface 42
6.3 Prevent fsync-induced Page Faults 42
6.4 Rethinking the Time Quantum 43
6.5 More Efficient Change Notifications 43

7 Conclusion **45**

Bibliography **47**

Chapter 1

Introduction

Storage I/O operations play a large role in server applications, especially commonly databases that provide the central data storage interface for a plethora of services built around it. Consequently, the ability for the database application to commit data to persistent storage quickly, while ensuring that the data can be recovered in the event of a runtime failure, such as an unexpected loss of power, plays a large role. Modern persistent memory [1] technology, like Intel Optane [2], was introduced to address this challenge by providing an intermediate layer between DRAM memory and NAND-flash based storage. Unlike conventionally available storage, persistent memory can directly be addressed by the CPU using load/store instructions but unlike DRAM, the written data is retained after the device has been powered off. However, effectively leveraging persistent memory requires the software to implement explicit support for them. Applications need to be adjusted to support persistent memory and interact with it through the proper I/O interfaces.

The I/O interface for persistent memory in Linux [3] is the *Direct Access* (DAX) [4] API, which allows applications to directly access persistent memory devices directly, or letting applications access the files from a filesystem on the persistent memory device directly, without any page cache indirection needing to happen first. While the file-granular approach provided by the DAX API substantially simplifies the work needed to make persistent memory devices work with existing applications, it still requires these applications to follow modern I/O practices in the form of memory-mapping file pages. Applications that rely exclusively on the traditional POSIX [5] I/O interfaces therefore cannot directly benefit from persistent memory, as every I/O operation has to pass via the kernel.

The recent introduction of the Compute Express Link (CXL) [6] peripheral interconnect standard has enabled approaches to integrate persistent memory access with devices that would previously not have been eligible for such an interface through its *CXL.mem* protocol [7]. Chiefly, recently proposed *hybrid SSDs* [8] that not only provide asynchronous I/O operations like traditional storage, but also expose a region of their capacity as persistent memory. Unfortunately however, the existing DAX API was primarily designed for persistent memory devices whose entire capacity can directly be addressed by the CPU [8], such as aforementioned Optane [2]. The API is therefore not equipped to support such *hybrid* devices in which only a portion of their capacity is exposed as persistent memory.

To support *hybrid SSDs* and to address the shortcomings of the DAX API, Khalil et al. [9] proposes *Transparent DAX Mappings* (TDM) which are supposed to allow arbitrary applications to make optimal use of *hybrid SSDs* without them requiring any software changes. The proposed design achieves this by transparently redirecting calls to POSIX [5] I/O operations to TDM-aware reimplementations, that access the device synchronously using persistent memory if the region is available as such or fall back to traditional asynchronous I/O through the operating system kernel. In this work, we aim to implement the proposed TDM design through a kernel subsystem that runs independently of the userspace process, allowing expensive operations to run independently from the actual software running in userspace.

In Chapter 2 we discuss underlying technologies that are required by our implementation, beginning by examining available options for persistent memory devices, before moving on to discuss relevant software interfaces. After that, we explore work related to our implementation of *Kernel-Based TDM Management* in Chapter 3, such as the proposed *Transparent DAX Mappings* [9] design. In Chapter 4 we explain the design details of our work and show the necessary implementation details. The benchmark results showing the effective performance of our work, both in synthetic and real-world workloads are showcased in Chapter 5. Chapter 6 discusses future improvements to our implementation as well as related areas of research that might be worth looking into further, before the final chapter provides the conclusion we have taken from this thesis.

Chapter 2

Background

In order to understand the proposed *Kernel-Based TDM Management*, the following sections aim to explain and discuss the technologies it is built upon. First, we introduce the properties and implementations of *Persistent Memory* devices [1], including *Compute Express Link* [6], a modern peripheral interconnect standard capable of serving as the base for modern *Persistent Memory* devices. Then we move on to talking about features of the Linux kernel [3] relevant to this work, before finally ending up discussing relevant implementation details of the x86_64 instruction set.

2.1 Persistent Memory Devices

As traditional storage devices exhibit access latencies several order of magnitudes higher than main memory, interacting with them was typically done via asynchronous operations. The operating system requests a storage device do a read or write operation at a particular block address and moves on, while the storage device completes that particular operation. Once it has, it will interrupt the CPU, which will result in the operating system processing the result only then.

Persistent Memory (PMem) devices refer to a class of technology, which allows the CPU to synchronously access persistent data through load/store operations on an address similar to DRAM, just that the backing storage medium is non-volatile and will be retained after it has been powered off [1]. Like DRAM, the CPU accessing the data stored at that particular address will stall the CPU until the device managed to fetch it. As long as the access latency is not *too* high, stalling the CPU can be a worthwhile trade-off, as it avoids overhead introduced by the interrupt handling.

2.1.1 Intel Optane

Intel Optane is a now discontinued persistent memory technology based on Micron's 3D XPoint [1]. It provided CPU-addressable persistent memory over its whole capacity, at speeds around three times slower than DRAM, yet faster than conventional flash storage [1]. While in production, it was the de-facto target for which persistent memory APIs in various operating systems were designed for, such as the Linux DAX API [4] shown in Section 2.2.2.

Intels Optane DC Modules, which we used during evaluation of our implementation (5.1) due to availability, are directly connected to the CPU through the DDR-T variant of standard memory sockets, similar to DDR-connected DRAM [1]. As such, both Optane persistent memory and the volatile DRAM memory compete for the same number of DDR-channels provided by the processor.

2.1.2 Hybrid SSDs

A flash-memory storage device, which not only provides an asynchronous interface, traditionally used for interfacing with storage devices, but also a byte-addressable interface which allows the CPU to directly access it through load/store instructions was initially proposed by Bae at al. [10] and later coined *hybrid SSD* by Habicht et al. [8]. While the synchronous addressing mode allows it to be used as *Persistent Memory*, the asynchronous interface allows the CPU to work on other tasks while the device runs the requested operation.

In particular, this property lets the device become more flexible in how it is implemented. Such is the case with the design proposed by Habicht et al. [8], where comparatively slow backing flash storage is headed by a software-defined DRAM cache, which writes its data back to the flash storage after it loses its external power source. Access to the DRAM cache is fast enough that the trade-off of stalling the CPU while waiting for the data is worth it, while accessing the data only available on flash is done asynchronously.

For this to work effectively, such a hybrid SSD must provide an interface for 'remote' managing of the DRAM cache by the host CPU. This is due to it having to be responsible for ensuring that no further synchronous

access happens after the relevant blocks have been evicted from the DRAM cache and due to the block-based view from the device likely not providing sufficient details about the filesystem. Instead, such an interface lets the OS determine which segments are experiencing high traffic and decide to pin them to the DRAM cache of the SSD in order to expose the CPU memory-addressable fast-path [9].

The peripheral interconnect standard chosen by by Habicht et al. [8] for their hybrid SSD prototype is *Compute Express Link*, as it provides first-class support for both synchronous and asynchronous addressing modes.

Compute Express Link

Compute Express Link (CXL) [6] is a new peripheral interconnect standard built upon *PCI Express* (PCIe) and developed by the *CXL Consortium*, of which Intel is a member, among with other industrial partners [7]. CXL was designed for low latency usecases, while retaining bandwidth characteristics similar to PCIe, allowing it to be used by various classes of CPU-attached peripherals, from neural network accelerators, software-defined networking cards to volatile and non-volatile memory expanders.

The specification defines three protocols, used for different tasks all within the purview of a CPU to peripheral interconnect:

- *CXL.io* is an asynchronous packet protocol used for initialization and configuration of CXL devices but can also serve as a generic message passing mechanism between the CPU and peripheral, and mostly equivalent to the *PCIe Transaction Layer*. As such, there are three different packet types which may be sent: one-shot *posted* packets, which do not require an answer from the receiver, *non-posted*, which do, and *completions* as reply thereof [7].
- *CXL.cache* is the cache-coherency protocol. It allows peripherals to access host memory in a cache-coherent manner, avoiding the pitfall where traditional *Direct Memory Access* (DMA) devices access stale data, due to the CPU not having flushed it back from its internal caches [7].
- *CXL.mem* is the protocol, which allows the CPU to cache-coherently access peripheral memory through simple load/store operations, similar to native memory connected to the host via DDR [7].

In addition to the three CXL protocols, the CXL specification also defines three classes of device types. While each class implements the *CXL.io* protocol, the implementation of the other protocols depends on the device type in question.

- *Type 1* devices are supposed to be peripherals that do not need to expose their internal memory back to the host, such as software-defined networking cards. These devices operate using the more traditional DMA interface, with the benefit that *CXL.cache* offers the ability to access host memory cache-coherently, reducing the complexity of software running on the CPU. As such, the protocols implemented by *Type 1* devices are *CXL.io* and *CXL.cache*.
- *Type 2* devices have, unlike *Type 1* devices, internal memory that would benefit from being directly addressable from the host CPU, while still retaining the ability to access host memory coherently. Examples of such devices include GPUs and other specialized accelerators that still have the need to access memory attached to the host CPU cache-coherently through *CXL.cache*, due to the internal *CXL.mem*-powered buffer not being sufficient enough for the task the device is designed for.
- *Type 3* devices do not need to access host memory. As such they do not implement *CXL.cache*, only *CXL.io* and *CXL.mem*. *Type 3* devices may be memory expanders, with the beneficial property that unlike Optane (2.1.1), they do not take away DDR channels that could be used from DRAM memory instead and offer a cheaper method for expanding host memory than adding more DDR channels to the processor.
  The hybrid SSD proposed by Habicht et al. [8] is also a *Type 3* device, with *CXL.io* being used for asynchronous I/O operations, while synchronous operations to the DRAM cache of the device use *CXL.mem*.

2.2 Linux Kernel

The Linux kernel [3] is a UNIX-like free-software operating system kernel, licensed under the GNU General Public License, Version 2 [11]. Due to the free license and it having become widely used, makes it serve as a decent base for academic work.

In this section, we begin by showing the features introduced by the *ext4* filesystem [12] that are necessary for our implementation of *Kernel-Based TDM Management* to function, before moving on to talk about the *DAX* [4] API provided by the kernel to let userspace applications interface with persistent memory devices (2.1).

2.2.1 ext4

The *ext4* filesystem [12] is the de-facto standard journaling filesystem of the Linux kernel. While retaining backwards compatibility with the previous generation journaling *ext3* and journal-less *ext2* file systems, *ext4* has significantly improved from its predecessors, especially when dealing with large files. This is due to the move from the traditional indirect block mapping scheme, to an extents-based approach [13]. Where the filesystem previously had a one-to-one mapping from logical blocks to disk blocks, an extent is capable of representing a large range of contiguous disk blocks, massively reducing the workload needed to delete or truncate large files. Unlike with the old indirect block mappings, the new extents system offers the ability to 'pre-allocate' files in- and outside of the logical file range tracked as part of the inode.

2.2.2 DAX

The 'Direct Access' (DAX) [4] API provided by the Linux kernel, exists in order for user-space processes to have the ability of bypassing the page cache and instead directly map persistent memory devices (2.1), such as Optane (2.1.1). The basic operation with a device implementing DAX is the so-called *devdax* mode. Userspace processes may access its corresponding `/dev/dax*` device node provided by the devfs and `mmap` [14] it, in order to have direct access to the memory of the device.

Building upon it, some file systems, such as *ext4* (2.2.1) and *xfs* support *fsdax*. *fsdax* allows for the creation of file systems on the physical device node, such as `/dev/pmem*`. These device nodes can then be mounted somewhere into the filesystem like traditional block devices and transparently interacted with. Yet, userspace processes have the ability to `mmap` files part of the *fsdax* filesystem with the `MAP_SYNC` flag, which creates a mapping

providing direct memory access to the region of the persistent memory device that stores the file, as long as the filesystem has a block-size that is a multiple of the page size and is aligned properly. *fsdax* by itself is however somewhat limited, as it is not possible for files to grow in size if they are mapped to a particular memory region, needing both `ftruncate` [15] and `mremap` [16] system calls in order to actually achieve it.

2.3 x86_64 Architecture Features

Currently, Intel's x86_64 CPU instruction set family is the most widespread CPU architecture in current workstation and server machines today. As such, many applications only target, or at least prioritize this instruction set over alternatives [17]. Due to new designs targeting the instruction set family continually being developed by Intel, these chips often provide a mechanism to gain access to new features, such as *Compute Express Link* (2.1.2), earlier than others. While our implementation should also work on other architectures that support persistent memory (2.1) through DAX (2.2.2), it was built on and evaluated on the x86_64 system shown in Section 5.1.

This section concerns itself with showing features of the x86_64 instruction set relevant to this work. First we will look at the `syscall` instruction, which is the main mechanism applications that target this instruction set use to communicate with the kernel on Linux, before moving on to *atomic operations* which facilitate memory synchronicity between CPUs in multi-CPU systems.

2.3.1 The syscall instruction

On Linux, the traditional interface of x86_64 applications to interact with the kernel is the `syscall` (`0F 05`) [18] instruction. Traditionally[1] upon encountering this instruction, the CPU would go into its privileged mode and continue execution from an address stored in the interrupt descriptor table [18]. In Linux, this is the `entry_SYSCALL_64` routine [20], which has

[1] Modern Intel processors and Linux release 6.9 and up support *Flexible Return and Event Delivery* (FRED), which adjusts the traditional *system call* handling to incur less latency [19].

to store the current register state of the user-mode process and configure the CPU to be properly in kernel mode, like changing the page table and adjusting the interrupt state. The CPU in kernel mode can then do the task the *system call* instructed it to do, potentially while having to convert userspace addresses passed through the registers reserved for syscall parameters into kernel addresses. After the operation has been completed, the context switching logic now has to happen in reverse, at the end calling `sysret` [18], which resumes execution in user-mode.

With the *system call* being the common interface, with which applications interact with the kernel, I/O heavy applications have to do a `syscall` each time they want to read or write data, at which point the overhead just spent switching between user- and kernel-mode starts to add up. This work aims to reduce the amount of necessary *system call* operations for such I/O heavy applications when operating with files that are stored on a DAX (2.2.2)-capable persistent memory device (2.1), since the actual I/O operation can be completed purely by the CPU through load/store instructions.

2.3.2 Atomic Operations

In order to facilitate multiple CPUs sharing the same system bus, either due to the presence of multiple physical processors or due to the now commonplace 'multi-core' technology, the x86_64 instruction set provides bus locking and/or cache coherency management mechanisms, for performing atomic operations on system memory [19]. These operations are typically used for synchronization of data and datastructures that may be accessed and written to by multiple CPUs in parallel.

Starting with the Pentium processor, load and store operations to an aligned byte, word, doubleword and quadword are guaranteed to be carried out atomically [19], thanks to *cache coherency protocols* between the CPUs being used to invalidate stale caches [19].

The GNU Compiler Collection (GCC) [21] provides a set of architecture-independent atomic operations, through its `__sync` and `__atomic` family of builtin intrinsics [22]. These get compiled down by the compiler into the atomic operations provided by the x86_64 instruction set, when targeting a CPU which supports them.

In our implementation of *Kernel-Based TDM Management*, we aim to leverage atomic operations on double- and quadwords through the use of aforementioned *GCC* intrinsics in order to let a userspace process communicate with a kernel worker running on a separate CPU on the same bus. This mechanism should allow for bi-directional user/kernel communication without incurring the cost the *system call* operation (2.3.1) has.

Chapter 3

Related Work

Since the introduction of *Optane* (2.1.1) by Intel, much research on optimizing the performance of persistent memory devices (2.1) has taken place. After its discontinuation in 2022 [1], focus shifted to suitable replacements, such as *CXL.mem*-capable storage devices [1].

As such, this chapter concerns itself with attempts to address the inadequacies of stock Linux DAX (2.2.2). First we look at *SplitFS* [23], which uses a hybrid kernel/user approach in order to reduce the load file-append operations take. Afterward, we look at a newly proposed interface [24] for the Linux kernel [3], which adjusts the existing DAX [4] API to be compatible with hybrid SSD designs that were shown in Section 2.1.2. Finally, we move on to discussing the proposed design of *Transparent DAX Mappings* [9], which allow userspace applications to harness benefits from persistent memory devices without them needing invasive changes to their source code.

3.1 SplitFS

While letting userspace application access files directly on a persistent memory device (2.1) through Linux DAX [4] works for fixed-size files, the moment the file has to grow past its initially allocated size, it starts to show its flaws. This is partially due to the `mmap` [14] system call treating mappings as a region with a known `offset` and `length`, and due to Linux aggressively zeroing the unused region of partial pages.

Kadekodi et al. [23] proposed *SplitFS*, as an approach to deal with the append issue through pre-allocated 'staging files'. If the application attempts to extend a file, the user-mode component copies the already existing partial block at the end of the current file, should such a partial block exist, into the 'staging file', trailed by the data the application wished to append. During the next `fsync` [25] operation, the *SplitFS* kernel component is instructed to perform a *relink* operation, moving the physical blocks which constituted the 'staging file' into the actual opened file, potentially replacing the previous partially filled block at the end. This mechanism allows *SplitFS* to provide filesystem operation guarantees, which are at least as strict as the guarantees required by POSIX [5].

3.2 Hybrid Storage

While the current Linux DAX interface is sufficient for traditional persistent memory devices such as Optane, new hybrid SSDs (2.1.2) that unlike Optane can not just be addressed synchronously through CPU load/store operations, but also asynchronously like traditional disks, may suffer. This is due to DAX effectively replacing the page cache for each file located in the host memory with a direct view to the mapped storage device. That works for Optane, but on modern hybrid SSDs, direct synchronous access through load/store operations may not be possible everywhere or suffer from a significant latency penalty [8].

In order to make the Linux DAX API [4] more amenable to hybrid SSDs, Habicht [24] proposed a new `hybrid_device` API to the kernel, similar to the existing `dax_device` [26]. This newly proposed API allows the page cache to become hybrid storage-aware, reducing the granularity of mappings into smaller chunks, allowing them to either exist directly on the device if it can be quickly accessed synchronously by the CPU or on the host memory, like traditional files, if they can not.

3.3 Transparent DAX Mappings

For a userspace applications to reap the benefits of persistent memory devices attached to the system, commonly requires adjustment of the applications existing code. Khalil et al. [9] sketch out a design of *Transparent DAX*

Mappings (TDM), as *DAX mappings* which are, from the perspective of the application, automatically created, in order to facilitate a transparent kernel bypass for I/O operations. This is achieved by replacing the `read` [27] and `write` [28] POSIX [5] calls with TDM-aware versions, that bypass the kernel system call interface if the file they are supposed to operate on, is listed in a pre-configured whitelist.

The hybrid SSD design described in Section 2.1.2 only ought to have the data stored in their DRAM cache directly accessed from the CPU using load/store instructions, with the operating system being responsible for telling the SSD which data should be available in the cache. As such, the OS must somehow collect I/O bandwidth data as a basis to make these decisions on. To accomplish this, Khalil et al. [9] propose a lightweight profiling mechanism built around `libc` [29], given that it has full knowledge about both POSIX [5] I/O operations performed using the system call interface and kernel bypass path.

For the benchmarks done by Khalil et al. [9], which achieve an up to 96.2% throughput increase in selected workloads, a 'simplified prototype' exists in the form of *libtdm* [30] and *tdm-writecache*, which only rely on a userspace library to manage the kernel bypass logic, the former also requiring the hybrid storage changes proposed by Habicht [24].

Chapter 4

Design and Implementation

This chapter presents our design and implementation of *Kernel-Based TDM Management*, a mechanism which allows I/O operations on DAX storage devices to bypass the *system call* (2.3.1) into the kernel and instead directly read and write to the device using `memcpy` [31], which should result in a performance benefit due to the reduction of overhead introduced by the system call. However unlike the traditional DAX interface described in Section 2.2.2, our approach aims to support *hybrid SSDs* (2.1.2) by only selectively determining which I/O operation should bypass the kernel. Additionally, we also aim to support high throughput append operations that the current DAX interface suffers with. The design is based around the proposed concept and laid out requirements of *Transparent DAX Mappings* (TDM) [9], which as per 'transparency' requirement allows existing applications to benefit from the performance increase promised by this TDM implementation without needing any source code changes. As this work introduces new API into the kernel for this in the form of the *procfs interface* (4.1.1), the transparency requirement is achieved through *libtdmiobypass* (4.2), described later in this chapter. This library transparently redirects calls to POSIX [5] I/O operations, such as `read` [27] and `write` [27] to its own TDM-aware implementations.

4.1 Kernel Subsystem

Figure 4.1 shows the general overview of the approach we have taken for *Kernel-Based TDM Management*. By splitting *fsdax* [4]-capable files into fixed-size chunks, marked *C0* through *C4* in the overview, the TDM kernel component may map individual chunks into the address space of the

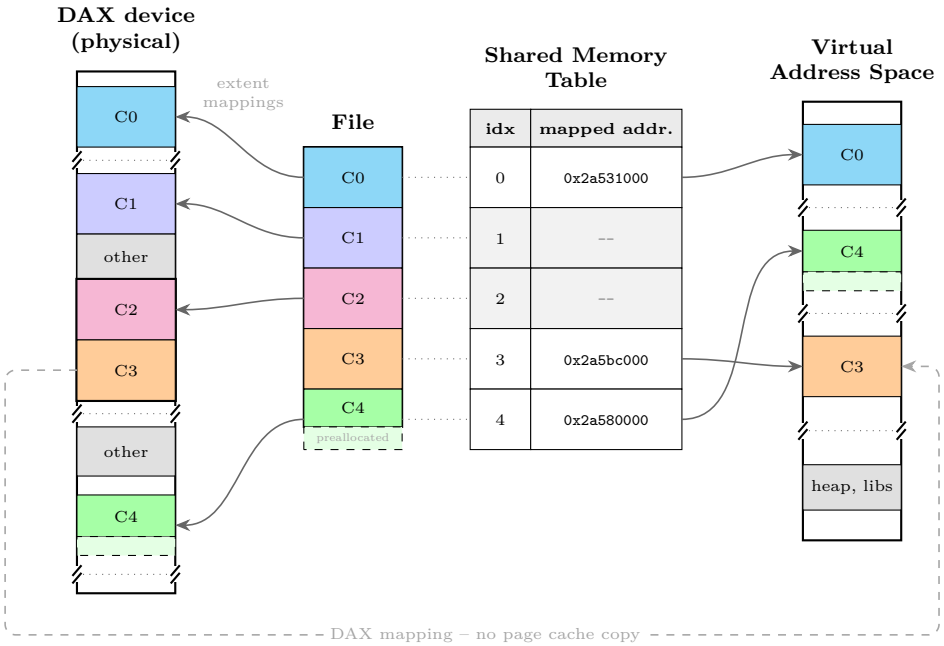


Figure 4.1: Overview of our proposed design for *Kernel-Based TDM Management*. A file stored on a DAX device is split into fixed-size chunks, that may or may not be mapped into userspace. The userspace is told the address of the mapped chunks through a shared memory table written by the kernel. The partial chunk at the end of the file is still fully pre-allocated on the DAX device and also mapped as the whole chunk.

userspace process. As the file is on a filesystem stored on a DAX device, this means that load/store operations by the CPU in the mapped region happen directly on the storage device, instead of the host memory-backed page cache (2.2.2). If a particular chunk is not currently mapped, the data can still be accessed using the traditional system call interface.

The userspace process is informed about the existence of those mappings through a shared memory table that allows for bidirectional communication between it and the TDM kernel component through the use of atomic operations (2.3.2) without incurring the latency penalty that would incur from the traditional system call interface (2.3.1). Any expensive operations, such as the mapping and pre-faulting of the TDM chunks inside the address space of the process can be done by a kernel worker running on a separate CPU. This does of course require that the machine that makes use of this implementation needs to have multiple CPUs available. Without that, the kernel would have to switch the running process into the kernel worker for it to complete the queued work and then switch back again, which would likely incur vastly more latency than the system call interface.

Not pictured in the overview is that the shared memory table begins with a fixed-size header that allows the userspace process to adjust file metadata such as the length by atomically adding or setting the corresponding fields inside the header, with the kernel worker asynchronously applying the change to the inode [32] later on. With length in particular, this allows for fast append operations, as the partial chunk at the end of the file (*C4* in Figure 4.1) is pre-allocated to encompass the whole chunk on the DAX device and mapped as such. The userspace process may increase the length of the file and whatever data is currently contained within the pre-allocated region will become part of the file.

To determine which chunks will be mapped into the userspace process, the kernel worker evaluates a caching policy based on the I/O bandwidth that each chunk experienced. As I/O operations through the DAX-mapping bypass the kernel, these bandwidth statistics cannot be collected by the kernel itself. As such, the shared memory table contains a *bandwidth class* column (also not pictured in the overview) that is used by the process to tell the kernel worker about the bandwidth experienced for each chunk.

We will now go on and show the individual kernel components our implementation is comprised of, beginning with the *procfs interface* that provides the shared memory table for each file to the userspace process, before moving on to the internal structures and the kernel worker.

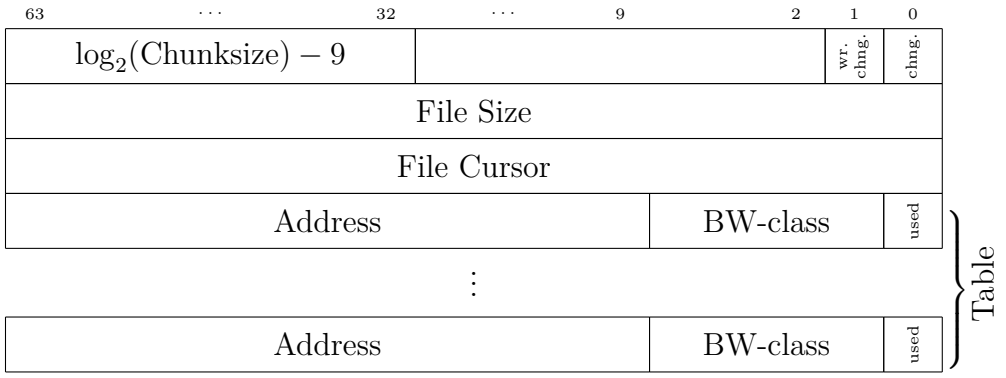


Figure 4.2: Layout of the shared-memory table, which has both a fixed-size metadata header and a dynamically sized list of entries. This allows the shared-memory table to accommodate a variable number of entries while keeping the metadata at a well-known location.

4.1.1 procfs interface

In our implementation, any communication with the TDM kernel subsystem takes places through its `procfs` [33] interface. While every process has access to its own `/proc/self/tdm/` (symlinked to `/proc/<pid>/tdm/`) directory, an attempt to access the `tdm` directory from a foreign process will fail. The directory contains an entry for each open file descriptor of a process that is backed by a DAX-capable file. As such, running `ls /proc/self/tdm/ 8< /dax/capable/file` in a POSIX [5]-compatible shell on a system with our implementation of *Kernel-Based TDM Management*, we would expect to see an entry for fileno. 8 listed by the `ls` [34] executable, as it currently has a DAX-compatible file opened as that file descriptor by the shell.

To use TDM with a file stored on a DAX device, the userspace process is expected to `open` [35] and `mmap` [14] the corresponding entry from the `/proc/self/tdm/` directory. While the `offset` parameter passed to `mmap` must be zero, since the fixed-size header is at the start of the mapping, deciding the `length` parameter is left at the behest of the process.

The returned memory segment is the shared memory table shown in Figure 4.1 and follows the format laid out in Figure 4.2. The number of entries is determined by the `length` argument that was passed to the

`mmap` [14] system call. Should the file eventually grow past the number of entries available in the table, the userspace process may `mremap` [16] it with a larger `length` parameter.

The fixed-size header of the shared memory segment provides the userspace with the *Chunksize*, which determines the granularity of each entry within the table. Like mentioned earlier, it also lets the process read and adjust the *size* of the file and, since the mappings correspond to an individual file descriptor, also adjust the *cursor* (also known as *offset*) of the file descriptor, by atomically reading or writing to the right field within the header.

As for the entries within the table itself, each entry refers to its respective chunk of the file, with each chunk referring to a segment of the backing file with the size of *Chunksize*. The entry itself consists of the *virtual memory address*, written by the kernel, where this particular chunk got DAX-mapped into - or `0` if it has not been mapped.

The lower eight bits two through nine are not part of the address, instead they are written by userspace to indicate the bandwidth usage of that particular chunk, which like mentioned earlier will be used by the kernel worker as the basis to evaluate its caching policy on. Chunks with a higher bandwidth class have a higher chance of their DAX mapping being created or retained, while chunks with a lower bandwidth class have a higher chance of it not getting created or it being evicted.

The least significant bit of the entry is used to indicate whether the chunk is currently in-use. If it is, the kernel subsystem will not unmap the DAX-mapped memory segment referred to by *Address* until the `USED` bit is unset. Yet it may reset *Address* back to `0`, as the mechanism to tell the process it should not be used for future kernel-bypass I/O operations anymore.

Like the problem with the kernel not knowing about the bandwidth statistics of bypassed chunks, the kernel also might not know if any write operation has taken place and as such, it cannot inform other processes using the *inotify* API [36]. To avoid this from happening, the userspace process may set the `WRCHNG` bit inside the fixed-size header, in which case the kernel worker will do so the next time it checks the shared memory table.

To avoid having the kernel worker walk every open table each time, the fixed-size header contains a `CHANGED` bit. If the bit was not set by the userspace process, the kernel worker will immediately move on, avoiding the need to scan any of the entries contained as part of the shared memory table, saving time and CPU cycles.

The kernel has been adjusted, in order to provide `linux/tdm.h` as a C-include for userspace code, providing a `struct tdm_filetable`, which matches the layout of the memory segment from Figure 4.2.

4.1.2 file tdm_table

On Linux [3], each file descriptor is represented through a `struct file` [37] on the kernel side. In order to keep track of administrative data for our implementation of *Kernel-Based TDM Management*, each `struct file` [37] is now outfitted with a `struct tdm_filetable`. That struct contains data, such as the `t_daxdev` member, which stores a reference to the DAX device (4.1.3), on which the file is available or the `t_folios xarray` [38], where the references to the pages of the TDM shared memory table (4.2) are stored in.

4.1.3 daxdev

The central component of the kernel-based TDM implementation is the `daxdev` [26]. When TDM is enabled, each `daxdev` will keep track of open TDM mappings attached to their particular device. Once the userspace accesses the `procfs` interface (4.1.1), the open mapping will be attached to it and the daxdev will start spinning the `tdm_qw` workqueue (4.1.4), which does the heavy work.

As the `daxdev` [26] is so to say only an abstract subclass, to be implemented by actual device drivers implementing DAX functionality (such as pmem [39]), the `struct dax_operations` [26] vtable has been extended with a `evaluate_tdm_policy` vfunc, letting the device driver with the most knowledge about the device be in charge of evaluating which mappings should be created, retained, and discarded.

4.1.4 tdm_qw workqueue

The main work done by the kernel subsystem happens within a `tdm_qw` workqueue. Each `daxdev` (4.1.3) that may hold a DAX mapping established through TDM, contains such a workqueue. Each workqueue only holds a single task, which calls `tdm_qw_cb` after a fixed delay. The delay defaults to half a second, but may be changed using the `tdm.quantum` kernel argument.

Within `tdm_qw_cb`, the TDM kernel subsystem does the most important work. First it allocates the `tdm_cpolicy` caching policy structure, then walks over all currently open TDM tables attached to the `daxdev` backing the `tdm_qw` workqueue. If the `CHANGED` bit has been set in the table header, it attempts to sync the file metadata stored in the TDM table with the kernel internal file and inode state. Further, it looks at all pages comprising the shared memory TDM table and the entries they are comprised of and fills the caching policy structure with any segments that have a bandwidth class greater than zero.

The `tdm_cpolicy` caching policy structure is then passed off to the newly added `evaluate_tdm_policy` vfunc of the `daxdev` implementation to be evaluated. Based on the evaluation results, `tdm_qw_cb` will then attempt to fulfill the requests made by the caching policy. This involves establishing mappings for newly requested chunks as well as attempting to discard mappings for chunks that have stopped being requested by the policy. However when attempting to discard a mapping that is currently marked as used by the process (i.e., the `USED` bit is set) it will instead set the *Address* to zero, like was mentioned in Section 4.1.1 and instead attempt to destroy the mapping the next time `tdm_qw_cb` is called.

4.2 Userspace Library

In order to make use of the interface provided by the kernel subsystem, the `libtdmiobypass` userspace library redirects calls from POSIX [5] I/O operations to TDM-aware replacement implementations, which fall back to the original implementation, should a file not be TDM-capable or the accessed segment not currently be mapped.

The redirection to TDM-aware implementations is achieved through the preload mechanism of the dynamic loader (commonly GNU's `ld-linux` [40]). As `libtdmiobypass` is built as a shared object, adding it to the preload libraries, either by its path being part of the `LD_PRELOAD` environment variable [40] or its path being passed via the `--preload` flag [40] to the dynamic loader results in its dynamically exported symbols being searched first, whenever the dynamic loader has to look up a new symbol.

While the dynamic loader loads `libtdmiobypass`, it calls `tdm_initialize` as it is listed in the `.init_array` section [41]. `tdm_initialize` determines file descriptor limits of the current process and allocates appropriately sized datastructures. Additionally, it uses the `dlsym` API provided by the dynamic loader in order to determine the direct addresses of the POSIX [5] I/O operation implementations to allow the POSIX [5] I/O operation reimplementations to fall-back to the implementations provided by libc, while avoiding getting redirected by the dynamic loader symbol resolution to the implementation of `libtdmiobypass`, resulting in an infinite loop until the stack space has been exhausted.

As the TDM table exposed through the `procfs` interface only deals with a single consumer per segment, multi-threaded processes demand additional synchronization mechanisms in order to facilitate concurrent `read`/`write` access to the same file; thus the need for the allocated datastructures during initialization. The basic datastructure follows a doubly indirect entry pointer, allowing lock-free access to the per-file descriptor datastructure (`TdmIoEntry`) used to keep track of relevant TDM state, as expanded further below. While the first layer has a defined fixed size (currently `TDM_N_SLOTS` = 256), the second layer is determined by the hard file descriptor limit of the current process, retrieved during initialization by getting the `RLIMIT_NOFILE` resource through the `getrlimit` [42] system call: $\frac{\texttt{rlim_max}}{\texttt{TDM_N_SLOTS}} + 1$

Each `TdmIoEntry` tracks the TDM state for an individual file descriptor. That means, it stores the address and size of the TDM shared memory table retrieved through `/proc/self/tdm/<fd no.>`. In order to facilitate concurrent access to the same file, access to the shared memory table is guarded by a *pthread rwlock* [5], preventing one thread from re-mapping the table in-case it needs to access an entry outside the currently mapped area, while other threads are interacting with the table. In addition, in order to synchronize the `USED` bit of concurrently accessed chunks, the structure also contains a fixed-size hash table, tracking per-entry concurrent access.

While a hashtable with a fixed number of buckets is algorithmically flawed due to it not being able to guarantee amortized $\mathcal{O}(1)$ access times anymore, the additional complexity induced from a dynamically resizing hash-table while retaining concurrent access was deemed unnecessary.

To interface with this datastructure, the main internal API to be used is:

```
1 TdmIoEntry *tdm_map_get_entry (int fd, size_t max_offset);
2
3 ssize_t tdm_entry_read (TdmIoEntry *self, void *buf,
4                         size_t count, off_t offset);
5
6 ssize_t tdm_entry_write (TdmIoEntry *self, const void *buf,
7                          size_t count, off_t offset);
```

which allows the POSIX [5] I/O API reimplementations to find the `TdmIoEntry` corresponding to a specific file descriptor, possibly allocating it or the indirect array if it did not exist before or determining that the file backed by the file descriptor is not DAX-capable, as well as reading and writing to them using a `pread` / `pwrite`-like API.

From this, `libtdmiobypass` implements the POSIX [5] functions `open`, `openat` [43], `creat` [44], `read`, `pread` [27], `write`, `pwrite` [28], `ftruncate` [45], `fstat` [46], and `close` [47]. Depending on the compiler configuration, it might also re-expose `pread`, `pwrite`, `ftruncate` and `fstat` as `pread64`, `pwrite64`, `ftruncate64`, and `fstat64` correspondingly, in order to avoid the issue of symbol names not necessarily matching the call name of the function, which is allowed by the POSIX standard [5].

Besides the already discussed `TDM_N_SLOTS`, the behaviour of *libtdmiobypass* can be adjusted using a set of preprocessor defines passed to the compiler. These behaviour adjustments were added during implementation and testing, in order to either address current shortcomings or to better understand the behaviour of our implementation.

- `-DCOLLECT_STATS`: If *libtdmiobypass* gets built with `-DCOLLECT_STATS`, it will collect data about the number of read and write accesses happening through the system call interface and through the TDM fastpath per file descriptor and print them after it was closed or at the end of the program. This metric can be used to gauge the effectiveness of our implementation during different workloads.

- `-DNO_LOCKING`: This option removes the use of any *pthread* lock [5] inside *libtdmiobypass* for single threaded workloads. Due to a lack of time, it does not remove the per-entry hash table that deals with concurrent access, even though that cannot happen during single threaded workloads. As the single-threaded use of *pthread* does not result in any sort of slowdown due to lack of lock contention, the `-DNO_LOCKING` flag will not be used during the evaluation (5).
- `-DNO_SYNC`: Due to an issue on how the `fsync` [48] system call currently treats files backed by *fsdax* [4], calling `fsync` on such a file causes all mappings to the file within the page table to become invalidated, similarly to any regular file in the host memory page cache. This results in a page fault the next time *libtdmiobypass* attempts to access pages from the mapping, as will be later shown during the evaluation (5.2.2). Having the userspace process encounter a page fault, sweeps away the work done by the kernel worker (4.1.4), having meticulously pre-faulted all pages within the created mappings. Instead, the 'fast'-path read and write operation for each page now happens through a fault path in the kernel, which has as bad, if not worse, overhead than the system call operation that was supposed to be avoided due to the overhead caused by it.
  In order to remedy this issue, the `-DNO_SYNC` preprocessor flag, causes *libtdmiobypass* to redirect calls to the `fsync` POSIX [25] function to a a no-op replacement, using the same mechanism as for the other POSIX I/O operations as laid out earlier.

4.3 Caching Policy

In Section 5.2, we will attempt to evaluate this implementation through a set of benchmarks, testing both synthetic and real-world workloads. For reasons of availability, expanded upon further within the evaluation chapter (5.1), our test system was only equipped with Intel Optane Persistent Memory (2.1.1).

Both due to a lack of time and the aforementioned configuration of our evaluation system, we only implemented a basic caching policy for the *nvdimm/pmem* driver, which is both used for Optane and *emulated* persistent memory through the `memmap` kernel parameter [49]. This caching policy iterates through the chunks of the caching policy input and indicates that

it wishes to map every chunk that got requested with `BWCLASS` = 255. As such, the current implementation of *libtdmiobypass* will mark any chunk it encounters during an I/O operation with aforementioned bandwidth class of 255, in order to have it established the next time the kernel worker reads the shared memory tables and evaluates the caching policy.

This policy has been designed to work well for Optane (2.1.1), given that the whole capacity of the persistent memory device may and should be directly synchronously accessed by the CPU for optimal performance, as there is not any sort of relevant cost associated with each mapping. However this is not the case for the in Section 2.1.2 discussed *hybrid SSD* prototype, with which only chunks contained in the DRAM cache of the device ought to be mapped, as *CXL.mem* access to uncached data results in the CPU stalling the unacceptably long time it takes the device to read the data from NAND flash.

Chapter 5

Evaluation

To obtain a better understanding of the performance of our implementation described in the previous chapter, this chapter concerns itself with running a set of I/O benchmarks on files stored on a persistent memory device with and without *libtdmiobypass* (4.2) and measuring as well as comparing the resulting performance metrics. We begin by outlining our evaluation setup, before moving on to running a set of benchmark suites and measuring their performance characteristics. With the statistics collected by *libtdmiobypass*, we then aim to better explain the results collected during the benchmarks, given our current caching policy. Lastly, we discuss our findings and examine the implications of the observed performance characteristics for the design of *Kernel-Based TDM Management* and effectiveness of our implementation.

5.1 Methodology

Given the lack of commercially available *hybrid SSDs* (2.1.2) and the expense associated with prototypes such as Samsungs CMM-H [50] or the FPGA-based prototype from Habicht et al. [8], we only made use of Intel Optane Persistent Memory. As such, we did not have to adjust our basic caching policy outlined in Section 4.3, as it works pretty well with Optane, given that unlike *hybrid SSDs*, the whole capacity of the drive is addressable as persistent memory.

The *Supermicro SYS-1029U-TRT* machine used for evaluation purposes is described in table 5.1 and is running Fedora 44 Server Edition [51] with Linux 7.0 [3], modified with the changes described in Section 4.1. For all

Component	Specification
Mainboard	Supermicro X11DPU
CPU	2× Intel Xeon Gold 5220 18 Cores @ 2.20GHz
Main Memory	12× SK Hynix 32GiB DDR4 SDRAM 2666MHz PC4-21300 CL19 ECC Registered 2Rx4
Persistent Memory	8× Intel 128GiB Optane DC Module PC4-21300 DDR4-2666 15W
Storage	Toshiba XG5 1TB M.2 NVMe Gen3x4 SSD (OS) 3× Micron 7300 PRO 960GB NVMe SSD

Table 5.1: Description of the *Supermicro SYS-1029U-TRT* machine used for evaluation.

tests, SELinux [52] had to be set to 'Permissive', as the default ruleset included as part of Fedora Linux does not associate a security context with the TDM handles, preventing them from being `mmap`ed [14]. Under the assumption, that all attached Optane persistent memory modules perform equally well, we just picked the first module exposed by the operating system as the drive hosting a customized *ext4* (2.2.1) filesystem without journaling in order to store all files used for measuring the performance during the evaluation.

5.2 Benchmarks

To evaluate the performance characteristics of our implementation, we will now run a set of I/O benchmark suits, on files from a filesystem that is stored on the aforementioned Intel Optane persistent memory module. The set of benchmark suites contains both synthetic and real-world workloads, allowing us to evaluate our implementation under a range of different access patterns that may be encountered. We run each benchmark suite individually with and without *libtdmiobypass* and depending on the benchmark some additional tweaks in order to have the benchmark perform optimally. If any such tweaks were used, we describe in detail why they are necessary.

We will first look at the performance characteristics of *diskv*, our internal disk validation tool used during implementation, which thanks to its basic access pattern makes it an ideal candidate for our current caching policy. Afterward we evaluate the performance of our implementation during traditional *on-line transaction processing* (OLTP), with the *Transaction Processing Performance Council Benchmark C* (TPC-C) [53] benchmark, before moving on to test our implementation in regards to append-only workloads using the popular cloud database benchmark *Yahoo! Cloud Serving Benchmark* (YCSB) with the key-value data store *Valkey* [54]. We end up continuing to use the into *Valkey* [54] builtin SET benchmark, in order to evaluate the induced latency of our implementation.

5.2.1 diskv

During implementation, we developed *diskv* as an internal disk validation tool, which effectively just writes and reads a well-known byte-sequence to/from a file. Due to its simplicity and its repeating access pattern, it serves as a pretty much optimal case for our basic caching policy discussed in Section 4.3, given that after evaluation of the policy any I/O operation to an already hit chunks should from this point onward always use the kernel bypass fast-path.

For *diskv* to run on a file, it takes in (o, l, n), with $o \in \{\text{write}, \text{validate}, \text{both}\}$, l defining the length of the byte-sequence, and n the number of iterations the operation should be repeated.

When using the traditional system call interface, *diskv* will do $l * n$ `read` [55] / `write` [56] system call operations if $o \in \{\text{write}, \text{validate}\}$ or $2 * l * n$ if $o = \text{both}$. However given our current caching policy and a sufficiently large l that a single iteration takes longer than the kernel worker time quantum q, with *libtdmiobypass diskv* will do at most l `write` system call operations, with the remaining $(n-1)*l$ operations being done through the kernel bypass path. Under the assumption that an I/O operation using the kernel bypass path is faster than an operation using a system call, the performance speedup when running *diskv* with *libtdmiobypass* compared to running it without will asymptotically diverge toward infinity, given a growing n.

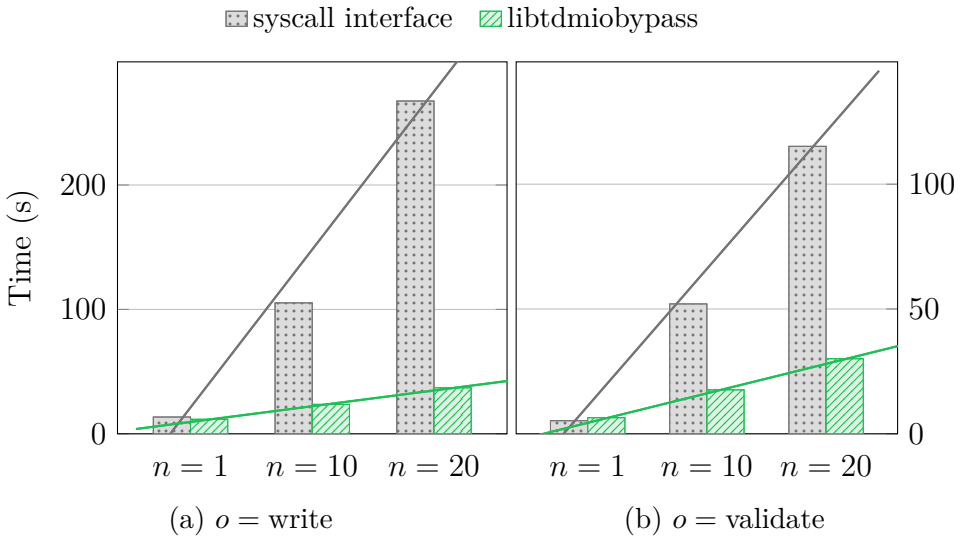
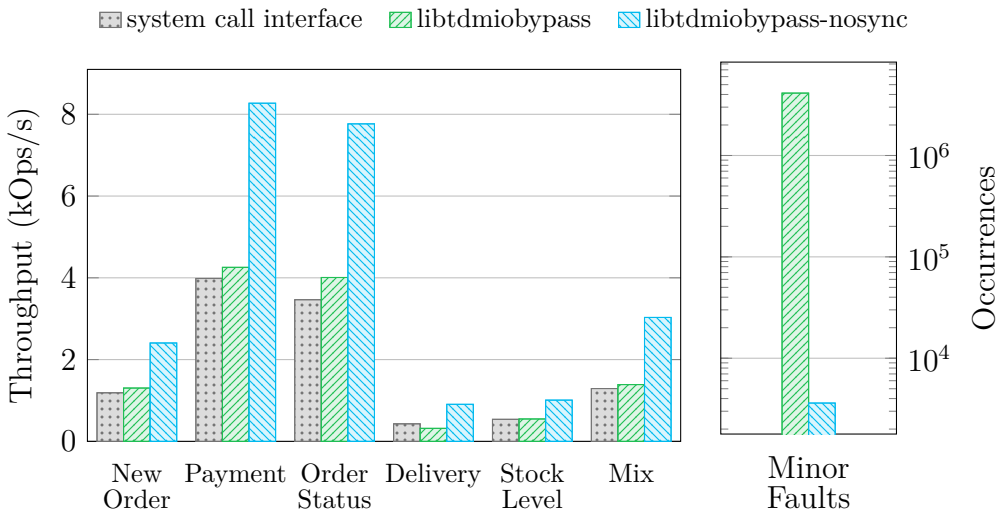


Figure 5.1: Time taken by *diskv* to complete its workload on a $l = 16MiB$ segment. While the time taken for $n = 1$ is roughly the same, running *diskv* with *libtdmiobypass* quickly outpaces *diskv* without by an ever-increasing margin.

As can be seen in Figure 5.1, the amount it takes for *diskv* to complete its $n = 1$ workload is roughly the same, regardless on whether the traditional system call interface or *libtdmiobypass* is used. However, as n starts to increase beyond that, using *diskv* with *libtdmiobypass* quickly starts to outpace the traditional system call interface.

5.2.2 TPC-C on SQLite

In order to evaluate traditional *on-line transaction processing* (OLTP) performance, we make use of the *Transaction Processing Performance Council Benchmark C* (TPC-C) [53] benchmark, with it being backed by a *SQLite* [57] database. For that, we use the *SQLite* port of *TPC-C* from *SplitFS* [23], with minor modifications to have it compile on the toolchain provided by Fedora Linux 44. The *SQLite* port was chosen, as it does not require running a separate process for the database, thus simplifying the whole setup. For *libsqlite*, the amalgamate constructed from *SQLite 3.53.3* [57] was built as part of the compilation of the TPC-C benchmark.



(a) Throughput of individual workloads

(b) Page faults during the evaluation

Figure 5.2: Evaluation of the *TPC-C* [53] benchmark on *SQLite* [57]. While there is generally a minor improvement when using *libtdmiobypass*, it is rather insignificant. However the `-nosync` variant of *libtdmiobypass* experiences a throughput increase of 135% on average, due to experiencing 99.91% less minor faults.

The benchmark simulates the order processing of a multi-warehouse wholesale system [58]. There are five different transaction types: New Order, Payment, Order Status, Delivery, and Stock Level, with each transaction involving a series of SQL statements, with different ratios of reads and writes.

The TPC-C benchmark was run using the default parameters: four warehouses, one district, 200,000 transactions and with SQLite in *Write-Ahead-Logging* (WAL) mode.

The throughput for each of the transaction types and the total average ('Mix') is shown in Figure 5.2a, like was done by Zhong et al. [58]. In the benchmark, the system call interface achieves a throughput of 1291.9 operations per second, with *libtdmiobypass* only performing marginally better on average with 1389.0 operations per second. Delivery transactions perform slightly worse with *libtdmiobypass* than using the system call interface.

The cause behind the lack of throughput increase with *libtdmiobypass* is the in Section 4.2 mentioned issue with the `fsync` [48] system call on DAX-mapped files. It it triggered by *SQLite* repeatedly calling `fsync` [25] on the WAL file, resulting in a page fault for each page the next time *libtdmiobypass* attempts to access a page from the mapping, as shown in Figure 5.2b. With the benchmark having to do the expensive access through the kernel fault path, *libtdmiobypass* only achieves a 7.5% throughput increase for the 'Mix' workload.

libtdmiobypass remedies this issue, through its `-DNO_SYNC` (4.2) preprocessor define. With this define set, any call to the `fsync` POSIX [25] function will be redirected to a no-op replacement (4.2), resulting in the kernel not running the flawed `fsync` handling. As such, the adjusted version of *libtdmiobypass* achieves significant improvements over the system call interface. With an average throughput of 3030.7 operations per second, it results in a performance increase of 135%, compared to the traditional system call interface.

5.2.3 YCSB on Valkey

We use the *Yahoo! Cloud Serving Benchmark* (YCSB) [59], a popular cloud benchmark for database implementations, to test the performance of the key-value data store *Valkey* [54], configured to use the *append-only file* (AOF) persistence mode. This mode appends all insert/update-transactions into a log file, allowing the current state to be restored by replaying the entries from the file back into the database [60]. The AOF persistence mode was chosen, in order to evaluate the performance of our kernel TDM subsystem (4.1) and *libtdmiobypass* (4.2) in regards to append-only workloads.

The benchmark contains six core workloads: A (50% read + 50% update), B (95% read + 5% update), C (100% read), D (95% read + 5% insert), E (5% insert + 95% scan), and F (50% read + 50% read-modify-write). Before testing, all workloads were modified, with both `recordcount` and `operationcount` increased hundredfold from the default $1,000$ to $100,000$, in order to better simulate longer living workflows needed for the kernel-based TDM management to work, as shown in Section 5.2.1. Before each workload, the *Valkey* state directory gets cleaned and the server restarted.

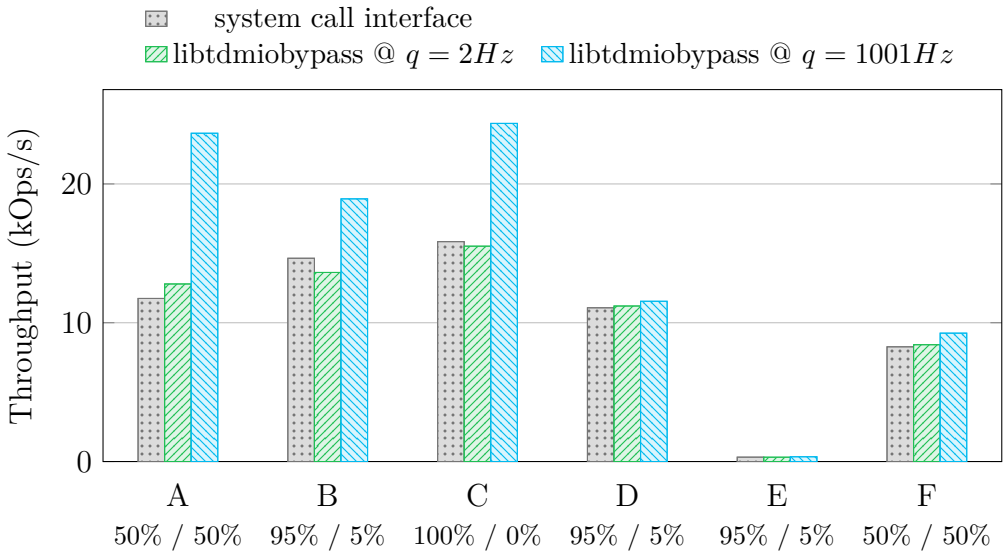
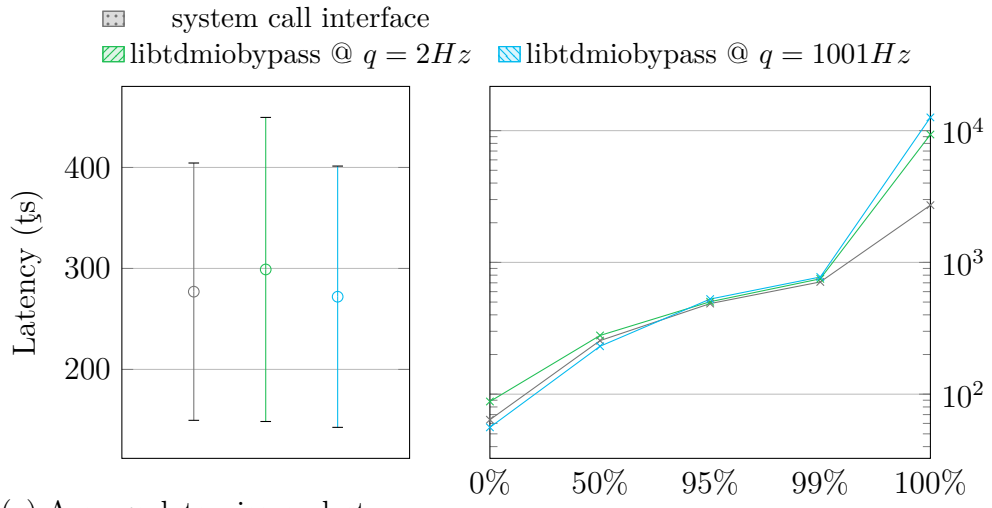


Figure 5.3: Throughput of *YCSB* [59] core workloads on Valkey [54], with percentages indicating the ratio of read and write operations. With `tdm.quantum` being set to the default $2Hz$, the throughput increases by at most 8.89%, while increasing the quantum frequency to $1001Hz$ results in up to 101.3% increased throughput.

Figure 5.3 shows, that with the stock TDM configuration of $q = 2Hz$, the throughput for each workload does not significantly change regardless on whether *Valkey* is only using the traditional system call interface or *libtdmiobypass*. As we will later expand upon in Section 5.3, this is due to the currently implemented caching policy not playing particularly nicely with the append-only workload from *Valkey*. The database simply writes the append-only log very quickly, making it difficult for the kernel worker to keep up, given the default scheduling quantum of $2Hz$. A better caching policy could be able to detect the append-only nature of the workload and react accordingly, by establishing mappings for multiple chunks ahead of the current end of the file. However, reducing the time quantum of the *tdm_qw* workqueue also achieves passable results, at the cost of increasing the power consumption of the machine, assuming the CPU running the workqueue could have otherwise gone into a sleep state.



(a) Average latencies and standard deviation

(b) Latencies broken up per percentile

Figure 5.4: *Valkey* [54] SET benchmark latencies when using the system call interface or *libtdmiobypass* with both the default $q = 2Hz$ and an adjusted $q = 1001Hz$. Latencies stay mostly similar, with a small increase of 7.94% in average latency of *libtdmiobypass* with the default $q = 2Hz$. However the highest recorded latencies are up to 363.4% higher with *libtdmiobypass*.

By running the kernel with `tdm.quantum=1001`, the time quantum of the workqueue gets set to $1001Hz$, which due to implementation internals using milliseconds, results in `tdm_qw_cb` (4.1.4) being continuously called, almost completely pinning one of the available CPUs. By doing that, Figure 5.3 shows a throughput increase of up to 101.3% in write-heavy workloads, similar to the 96.2% from the userspace-only TDM prototype shown in Khalil et al. [9].

5.2.4 Valkey SET

We employ the *Valkey* [54] SET benchmark [61] in order to test the latencies induced by both the system call interface and *libtdmiobypass*. The setup of the *Valkey* key-value data store remains the same as in Section 5.2.3, with the SET benchmark configured to use keys up to $2^{20} - 1$ and to run 2^{20} SET operations.

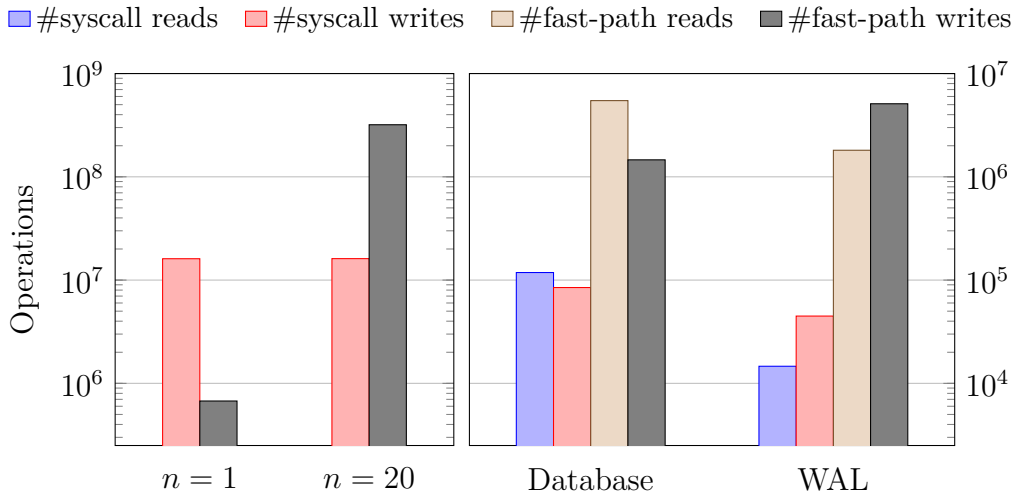
The results from the *Valkey* [54] SET benchmark [61] are displayed in Figure 5.4. While there are no major latency improvements either way, a small latency increase is noticeable when using *libtdmiobypass* with the time quantum being set at $2Hz$. As will be shown in Section 5.3, at that frequency *Valkey* simply outpaces the kernel worker, and most I/O operations will take the slow system call path. As such, the latency increase can be seen as pure overhead introduced by *libtdmiobypass* having to check whether the TDM fast-path is currently available or not.

Also of note is the slowest latency each, recorded in Figure 5.4b. It shows that with *libtdmiobypass*, a few operations took significantly longer than when only using the traditional system call interface. This is likely due to lock contention within *libtdmiobypass* itself, when multiple writers either were attempting to each increase the size of the shared memory table due to the current size not being enough to contain relevant entries and the writers having to wait until all current operations have finished, or multiple writers simply contesting the same slot within the fixed-size hash-table needed to keep track of parallel writes to the same chunk.

5.3 Caching Policy

Even though there is currently only the one caching policy outlined in Section 4.3 implemented, in this section we will be looking at the I/O statistics collected by *libtdmiobypass* using the `-DCOLLECT_STATS` preprocessor flag (4.2) during the benchmarks from the previous section. With the preprocessor define set, *libtdmiobypass* will collect data about the number of read and write accesses happening through the system call interface and through the TDM fast-path per file descriptor and print them after it was closed or at the end of the program. Based on the collected statistics we will attempt to evaluate the effectiveness of the current caching policy and attempt to offer a reasonable explanation on why the benchmarks behaved as they did during evaluation in the previous section.

Figure 5.5a shows that with our current caching policy, the number of I/O operations utilizing the system call path stays relatively constant regardless of the n parameter chosen, as was explained in Section 5.2.1. The remaining I/O operations are fulfilled through the kernel bypass fast-path. As *diskv* is expected to make $20\times$ as many I/O accesses with $n = 20$ compared to $n = 1$, this logically results in the number of fast-path writes



(a) `write` *diskv* workload, with $l = 16MiB$

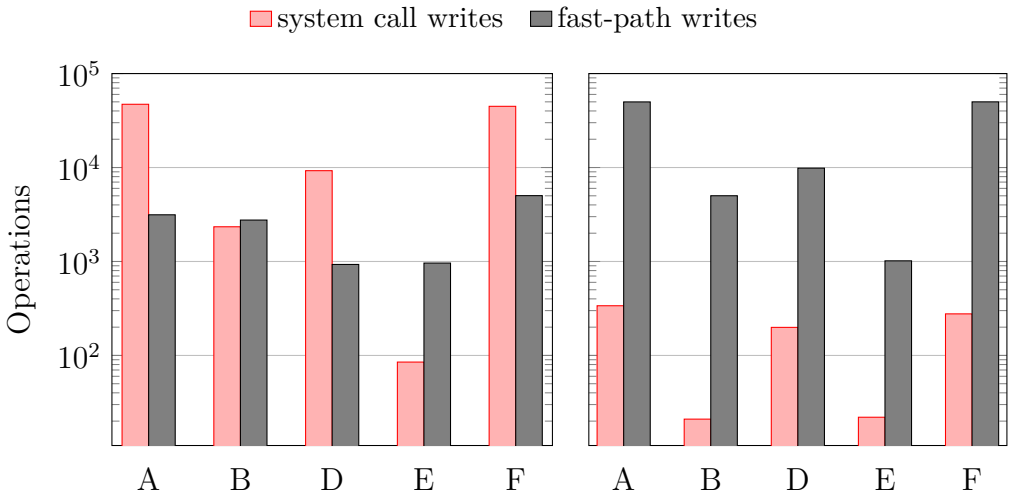
(b) *TPC-C* [53] on *SQLite* [57] workload

Figure 5.5: Comparison on the number of system call vs. kernel-bypass I/O operations. Given their corresponding benchmark results (5.2), workloads with a significantly higher number of fast-path I/O operations than system call path ones perform measurably better with than without *libtdmiobypass*.

increasing significantly with $n = 20$, given that the number of system call writes remain roughly constant. The heavy use of the kernel bypass fast-path results in *diskv* having massively improved performance when running through *libtdmiobypass*, as was shown in Figure 5.1a.

As shown in Figure 5.5b, both kinds of I/O operations on the *SQLite* [57] database and the write-ahead log (WAL) during the evaluation of the *TPC-C* benchmark took the kernel bypass path significantly more often than the system call fallback. When combining read and write operations, the kernel bypass path is used 3313% more often than the system call path for I/O operations on the database and 11518% more often for I/O operations to the write-ahead log.

Figure 5.6 shows the number of system call and kernel bypass fast-path writes that have taken place during the evaluation of the individual core workloads from the *YCSB* [59] benchmark on *Valkey* [54] with *libtdmiobypass*, both with the the stock configuration of the kernel and with `tdm.quantum` being set to $1001Hz$. As workload C only does read opera-



(a) $q = 2Hz$ (b) $q = 1001Hz$

Figure 5.6: system call vs. kernel-bypass I/O operations for *YCSB* [59] on *Valkey* [54], with regard to the configured *TDM* kernel worker scheduling quantum. At $q = 2Hz$ there are generally more system call operations than fast-path ones, but with $q = 1001Hz$ fast-path I/O operations outpace system call ones significantly.

tions, no data was written in the append-only file and recorded by *libtdmiobypass*. As such the workload was excluded from the figure. Figure 5.6a shows the initial run of the *YCSB* workloads with the stock configuration of $q = 2Hz$, where the slow system call path was, with the exception of workload E, used 596.4% more often than the kernel bypass fast-path for write operations, when averaging the results of all workloads. This contrasts the results from the previous benchmarks (Figure 5.5), where we have seen the fast-path being used significantly more often. The difference can be attributed to the kernel worker only running twice per second in the stock configuration, which results in the *TDM* shared memory tables (4.1.1) being read and the caching policy evaluated only twice per second, making it very difficult to keep up with the pace of *Valkey*.

As explained in Section 5.2.3, while a better caching policy would be more efficient, setting the time quantum to $1001Hz$ has the result of having the *TDM* kernel worker immediately re-schedule itself after it completed, effectively pinning one CPU to the task of continually scanning the shared memory tables and evaluating the caching policy based on the entries. As seen in Figure 5.3 this offers a throughput increase compared to

the system call interface of 101.3% and can be explained by the collected statistics shown in Figure 5.6b. Here, every core *YCSB* workload results in significantly higher use of the kernel-bypass I/O path by *Valkey* [54], with workload B resulting in the fast-path being used 23752% more often than the system call path.

5.4 Discussion

In this work, we have succeeded at taking the *Transparent DAX Mappings* concept by Khalil et al. [9] and implementing it through the use of a Linux kernel [3] subsystem (4.1), as we had initially set out to do. In Section 5.2, we have measured how our kernel-based implementation shapes up against the traditional system call interface for I/O operations. To prove the viability of our kernel-based approach and the adjusted design allowing for finer-grained decisions than the file-level, our aim was to match or exceed the performance increase of the userspace-only prototypes [30] used by Khalil et al. [9], in order to initially evaluate the concept of TDM.

While our kernel-based implementation managed to deliver throughput improvements comparable to, and in some cases exceeding those of the existing prototypes, it does not offer the same latency improvement that was shown by Khalil et al. [9]. We assume this is likely due to an inefficient implementation of the *libtdmiobypass* userspace component. This should be able to be partially addressed by improving *libtdmiobypass*, to make it less reliant on *pthread* locks [5]. However, our design of the shared memory table (4.1.1) inherently results in requiring some form of synchronization when dealing with multiple operations on the same chunk in parallel, due to the per chunk lock-state being serialized into a single bit. This limitation was not apparent to us when we were considering the design of the shared-memory interface, as we primarily looked at the task from the perspective of a single-threaded consumer.

Even though we were able to reach throughput increases of over 100% in multiple real-world workloads, we did not reach a state where our implementation could be considered ready to be used in real production workloads. This is due to our implementation always requiring additional intervention besides use of the base-*libtdmiobypass* to achieve these increased throughput results for the real-world workloads in our benchmarks. While the required interventions allowed us to demonstrate the potential performance

benefits of our approach, they are inherently unfit to be used as-is in a production environment. During our benchmark, these interventions either took the form of the `-nosync` variant of *libtdmiobypass* (4.2) to address the issue of the `fsync` system call causing expensive page faults, or a very high `tdm.quantum` frequency to address the currently lackluster caching policy (4.3).

Nevertheless given the results, we believe that an improved implementation of *kernel-based TDM management* will be necessary to unlock the full potential of *hybrid SSDs* (2.1.2). This is due to the inherent cooperation between the *hybrid SSD* driver, the filesystem, and the TDM subsystem that will be necessary in order to properly control which regions of the *hybrid SSD* capacity ought to be available for direct access by the CPU via persistent memory.

Chapter 6

Future Work

While this work manages to implement the kernel-based TDM management as was set out from the start, our implementation is not finished and is lacking several features and improvements that should be added before it can be considered ready. Furthermore, during the course of this work, several new topics where discovered, that may be worth looking into further.

This chapter concerns itself with expanding upon all the improvements and topics that might prove useful further work.

6.1 Necessary Improvements

Like was already mentioned in Section 4.3, there is currently only a very basic caching policy implemented. Improving the current one to better deal with append-only workloads, as well as implementing a caching policy for hybrid SSDs (2.1.2) is the immediate first issue that needs to be addressed.

Additionally, reworking *libtdmiobypass* to reduce the amount of lock contention encountered with multiple writes would likely reduce the latency of the highest percentile in the *Valkey* [54] SET benchmark [61] (5.4b). Given our current interface however, actually being able to reduce the contention resulting from the attempt to resize the shared memory table might not be possible without serious adjustments.

6.2 Move Toward Common Interface

The current approach using the shared memory table makes using the kernel bypass path very easy for userspace, as the address pointing to where the mapping of any chunk may be located can easily be computed by just knowing the offset and chunk size. Communicating data the other direction however, results in the kernel subsystem having to walk the entirety of the shared memory table. While that is fine for sufficiently small files, it suffers when dealing with very large files, as bandwidth changes in one small region of the file, which is not quite unlikely given the size, require a whole scan of the table, slowing the kernel worker down with unnecessary work.

An adjustment of the current user-kernel interface, with the bandwidth statistics being transported over an asynchronous message passing channel like a shared memory ring buffer, would avoid the long table walking issue. It would also open up the possibility of far higher-grained bandwidth classes, as the current interface only supports a total of 256 classes (4.2).

The Linux kernel [3] already provides such an asynchronous task submission interface in the form of *io_uring* [62]. Instead of reinventing the wheel, implementing the bandwidth statistic passing on top of *io_uring* would likely be a good choice. Yet, moving further and implementing anything other than the bandwidth passing through *io_uring* will present further difficulty, as the atomic operation semantics the kernel-based TDM management heavily relies upon are lost due to the asynchronous nature of the interface.

6.3 Prevent fsync-induced Page Faults

As was shown in Figure 5.2b, the `fsync` [48] system call results in DAX mappings being tagged as dirty, which results in a page fault the next time the userspace process attempts to access each page that belongs to the mapping. As these page faults cause heavy performance degradation, adjusting the relevant mechanisms to stop marking the pages as dirty is a central requirement for making kernel-based TDM management operate as expected, without the need for questionable 'hacks' like the `-nosync` variant of *libtdmiobypass*, which redirects any call to `fsync` [25] to a no-op implementation, in order to work around this issue, that Linux DAX [4] has with `fsync` (4.2).

6.4 Rethinking the Time Quantum

Instead of the current approach of having a kernel worker read the TDM tables and evaluate the caching policy periodically at a set frequency, evaluating the caching policy based on signals from the user-space process could offer significant efficiency gains. In particular, every I/O operation or every nth I/O operation that uses the system call fallback path could serve as signal for the kernel to check if the TDM table has changed and if so re-scan the table and evaluate the caching policy on an as-needed basis. Unlike with the timing-based approach, reacting on system calls also means that the kernel does not need to check every individual opened TDM shared memory table if the `CHANGED` bit is set as only the `struct file` [37] passed via the file descriptor into the system call will need to be checked.

We believe that there are two main approaches that could be used. The first would be having the kernel immediately check the shared memory table and evaluate the caching policy during the system call itself, while the CPU is already in kernel mode. This would have the advantage that the userspace process sees instantaneous results inside the shared memory table, once control is handed back to the process. The alternative approach would be to have the system call immediately queue the kernel worker on another CPU, which would avoid stalling the process while the TDM subsystem does the necessary work, at the cost of only offering the results inside the shared memory table fairly soon into the future, during which more I/O operations that have to take the system call path might be run by the process. As there are benefits and downsides to each of the approaches, picking the better one requires further research. However implementing the latter approach within our current implementation should be fairly straight forward.

6.5 More Efficient Change Notifications

Currently, synchronization between the userspace process and kernel worker only happens through shared memory, with the kernel worker repeatedly checking if the `CHANGED` bit has been set before going on to walk the shared memory table if it is. Instead of relying on the one CPU running the kernel worker having to poll the memory value, CPU architectures often support ‘Inter-processor Interrupts’ (IPI) [19], which allow one CPU to

interrupt another CPU or a group of CPUs that share the same bus. It may be possible to use these interrupts to asynchronously signal the kernel worker on another CPU to begin work, without incurring high latency on the processor running the process. This mechanism does not have to be limited to the kernel-based TDM management, instead it might be possible to define some sort of generic async system call interface using it.

Chapter 7

Conclusion

Continued research on hybrid storage devices paves the way for cost-effective, high-performance storage that has the ability to be accessed either through traditional asynchronous I/O interfaces or directly by the processor using load/store instructions. In order to make optimal use of these new devices, new I/O paradigms are necessary.

In this work, we have presented our take on how *Kernel-Based TDM Management* might be used to facilitate high-performance I/O access to *hybrid SSDs* (2.1.2). By redirecting calls from applications to the POSIX [5] I/O operations, TDM allows existing applications to benefit from *hybrid SSDs* without requiring modifications to their source code [9]. In contrast to previous approaches that operate at file-level granularity, our design provides sub-file granular access through the use of fixed-size chunks, offering greater flexibility in how the DRAM cache resources of the device are used.

During evaluation (5), we have tested our implementation on TPC-C [53], a benchmark simulating real-world OLTP performance and reached an average throughput increase of 135%. We further evaluated our approach using the write-heavy workload A from the core workloads of the *YCSB* [59] database benchmark with *Valkey* [54], a fork of the popular *Redis* [63] key-value data store. With *Valkey* configured to use the AOF persistence mode, to determine the effectiveness of append-only workloads, we were able to achieve a throughput increase of 101.3% with our *libtdmiobypass*.

Moreover, our results from the *YCSB* benchmark show, that our sub-file granular approach in the form of fixed-size chunks is competitive with the file-level granularity employed in the prototype presented by Khalil et al. [9]. At the same time, this finer-grained approach offers greater flexibil-

ity for managing the available DRAM cache capacity, as mentioned above. Overall, our work demonstrates the potential offered by *kernel-based TDM management* as a possible approach for integrating hybrid storage devices into existing software while retaining the performance benefits offered by persistent memory. In this regard, we have outlined possible avenues for future work. In particular, ironing the kinks of our implementation by actually integrating a caching policy made for *hybrid SSDs* (2.1.2) and making Linux [3] handle `fsync` [48] operations on DAX-mapped storage properly, are necessary relevant improvements before making our implementation of *kernel-based TDM management* feasible in real workloads.

Bibliography

[1] P. Desnoyers, I. Adams, T. Estro, A. Gandhi, G. Kuenning, M. Mesnier, C. Waldspurger, A. Wildani, and E. Zadok, "Persistent memory research in the post-optane era," in *Proceedings of the 1st Workshop on Disruptive Memory Systems*, ser. DIMES '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 2330. [Online]. Available: https://doi.org/10.1145/3609308.3625268

[2] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane dc persistent memory module," 2019. [Online]. Available: https://arxiv.org/abs/1903.05714

[3] L. Torvalds and Linux Kernel Contributors, "Linux kernel," https://www.kernel.org/, 1991–2026, free operating system kernel. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/?h=v7.0-rc7

[4] The kernel development community, "Direct Access for files," *The Linux Kernel Documentation*. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/dax.html

[5] IEEE and The Open Group, *IEEE Standard for Information Technology–Portable Operating System Interface (POSIX®) Base Specifications, Issue 8*, The Open Group Std. IEEE Std 1003.1-2024, 2024, issue 8, 2024 Edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/

[6] Compute Express Link Consortium, Inc., "Compute express link (cxl) specification 3.0," https://www.computeexpresslink.org, 2022.

[7] D. Das Sharma, R. Blankenship, and D. Berger, "An introduction to the compute express link (cxl) interconnect," *ACM Comput. Surv.*, vol. 56, no. 11, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3669900

[8] D. Habicht, Y. Khalil, L. Werling, T. Gröninger, and F. Bellosa, "Fundamental os design considerations for cxl-based hybrid ssds," in *Proceedings of the 2nd Workshop on Disruptive Memory Systems*, ser. DIMES '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 5159. [Online]. Available: https://doi.org/10.1145/3698783.3699380

[9] Y. Khalil, D. Habicht, P. Ellinger, F. Bellosa, J. González, A. Manzanares, and V. Shah, "Transparent dax mappings: Towards automatic kernel bypass with cxl-based hybrid ssds," in *Proceedings of the 3rd Workshop on Disruptive Memory Systems*, ser. DIMES '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 5462. [Online]. Available: https://doi.org/10.1145/3764862.3768178

[10] D.-H. Bae, I. Jo, Y. A. Choi, J.-Y. Hwang, S. Cho, D.-G. Lee, and J. Jeong, "2b-ssd: The case for dual, byte- and block-addressable solid-state drives," in *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, 2018, pp. 425–438.

[11] Free Software Foundation, "Gnu general public license, version 2," Jun. 1991, version 2, June 1991. [Online]. Available: https://www.gnu.org/licenses/old-licenses/gpl-2.0.txt

[12] T. Ts'o, *ext4(5) — File Formats Manual*.

[13] A. Mathur, M. Cao, S. Bhattacharya, A. Dilger, A. Tomas, and L. Vivier, "The new ext4 filesystem: current status and future plans," *Proceedings of the Linux Symposium*, pp. 21–34, 01 2007.

[14] Linux man-pages project, *mmap(2) — System Calls Manual*.

[15] ——, *truncate(2) — System Calls Manual*.

[16] ——, *mremap(2) — System Calls Manual*.

[17] M. Kumar, G. Kaur, and P. Singh Rana, "Aarch64 vs. x86_64 in HPC Cloud: Performance and Cost Evaluation," in *2024 Eighth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, 2024, pp. 85–88.

[18] Intel Corporation, *Intelő 64 and IA-32 Architectures Software Developer's Manual, Volume 2B: Instruction Set Reference, M–U*, 92nd ed., Intel Corporation, 2026. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[19] ——, *Intelő 64 and IA-32 Architectures Software Developer's Manual, Volume 3A: Instruction Set Reference, Part 1*, 92nd ed., Intel Corporation, 2026. [Online]. Available: https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html

[20] Linux Kernel Contributors, "Linux Kernel Source: `entry_SYSCALL_64`," 2026, contains the definition of `entry_SYSCALL_64`. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/arch/x86/entry/entry_64.S

[21] GNU Project, "gcc: The gnu compiler collection."

[22] ——, *Using the GNU Compiler Collection (GCC)*, Free Software Foundation. [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-15.3.0/gcc.pdf

[23] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, A. Kolli, and V. Chidambaram, "Splitfs: reducing software overhead in file systems for persistent memory," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 494508. [Online]. Available: https://doi.org/10.1145/3341301.3359631

[24] D. Habicht, "Rethinking storage i/o for hybrid nvme and dax block devices," Master Thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, Jul.19 2024.

[25] IEEE and The Open Group, "fsync — posix.1-2024," 2024, issue 8, 2024 Edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/functions/fsync.html

[26] Linux Kernel Contributors, "Linux Kernel Source: `struct dax_device`," 2026, contains the definition of `struct dax_device`. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/drivers/dax/super.c

[27] IEEE and The Open Group, "pread, read — posix.1-2024," 2024, issue 8, 2024 Edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/functions/read.html

[28] ——, "pwrite, write — posix.1-2024," 2024, issue 8, 2024 Edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/functions/write.html

[29] GNU Project, "glibc: The gnu c library," 2025. [Online]. Available: https://www.gnu.org/software/libc/libc.html

[30] P. Ellinger. [Online]. Available: https://gitlab.kit.edu/kit/itec/os/research/habicht/hybrid-ssd/tdm/libtdm/

[31] IEEE and The Open Group, "memcpy — posix.1-2024," 2024, issue 8, 2024 Edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/functions/memcpy.html

[32] Linux man-pages project, *inode(7) — Miscellaneous Information Manual*.

[33] ——, *proc(5) — File Formats Manual*.

[34] IEEE and The Open Group, "ls — posix.1-2024," 2024, issue 8, 2024 Edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/utilities/ls.html

[35] Linux man-pages project, *open(2) — System Calls Manual*.

[36] ——, *inotify(7) — Miscellaneous Information Manual*.

[37] The kernel development community, "`struct file`," *The Linux Kernel Documentation*. [Online]. Available: https://docs.kernel.org/filesystems/api-summary.html#c.file

[38] M. Wilcox and Linux Kernel Contributors, "Xarray," *The Linux Kernel Documentation*. [Online]. Available: https://docs.kernel.org/core-api/xarray.html

[39] The kernel development community, "Persistent memory," *The Linux Kernel Documentation*. [Online]. Available: https://nvdimm.docs.kernel.org/index.html

[40] Linux man-pages project, *ld.so(8) — System Manager's Manual*.

[41] GNU Project, "glibc Source: `call_init`," 2026. [Online]. Available: https://sourceware.org/git?p=glibc.git;a=blob;f=csu/libc-start.c;hb=2.42

[42] Linux man-pages project, *getrlimit(2) — System Calls Manual*.

[43] IEEE and The Open Group, "open, openat — posix.1-2024," 2024, issue 8, 2024 Edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/functions/open.html

[44] ——, "creat — posix.1-2024," 2024, issue 8, 2024 Edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/functions/creat.html

[45] ——, "ftruncate — posix.1-2024," 2024, issue 8, 2024 Edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/functions/ftruncate.html

[46] ——, "fstat — posix.1-2024," 2024, issue 8, 2024 Edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/functions/fstat.html

[47] ——, "close — posix.1-2024," 2024, issue 8, 2024 Edition. [Online]. Available: https://pubs.opengroup.org/onlinepubs/9799919799/functions/close.html

[48] Linux man-pages project, *fsync(2) — System Calls Manual*.

[49] Linux Kernel Contributors, "The kernel's command-line parameters," *The Linux Kernel Documentation*. [Online]. Available: https://www.kernel.org/doc/html/v7.0/admin-guide/kernel-parameters.html

[50] M. Soltaniyeh, G. Sun, X. Yao, A. Beygi, R. Kachare, D. Zhao, H. Huen, A. Chang, S. Murugesapandian, and C. Kahn, "Revisiting memory hierarchies with cmm-h: Use device-side caching to integrate dram and ssd for a hybrid cxl memory," in *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems*, ser. HotStorage '25. New York, NY, USA: Association for Computing Machinery, 2025, p. 4551. [Online]. Available: https://doi.org/10.1145/3736548.3737828

[51] Fedora Project, "Fedora linux 44 server edition," Apr. 2026. [Online]. Available: https://download.fedoraproject.org/pub/fedora/linux/releases/44/Server/

[52] S. Smalley, "Configuring the selinux policy," National Security Agency, Fort Meade, MD, USA, Tech. Rep., Feb. 2002. [Online]. Available: https://media.defense.gov/2021/Jul/29/2002815725/-1/-1/0/CONFIGURING-SELINUX-POLICY-REPORT.PDF

[53] Transaction Processing Performance Council, "Transaction Processing Performance Council Benchmark C," 1992–2021. [Online]. Available: https://www.tpc.org/tpcc/

[54] Linux Foundation and Valkey Contributors, "Valkey (9.0.3)," https://valkey.io/, 2024–2026, redis Fork.

[55] Linux man-pages project, *read(2) — System Calls Manual*.

[56] ——, *write(2) — System Calls Manual*.

[57] D. R. Hipp, "SQLite," Jun. 2026. [Online]. Available: https://www.sqlite.org

[58] S. Zhong, C. Ye, G. Hu, S. Qu, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Swift, "MadFS: Per-File virtualization for userspace persistent memory filesystems," in *21st USENIX Conference on File and Storage Technologies (FAST 23)*. Santa Clara, CA: USENIX Association, Feb. 2023, pp. 265–280. [Online]. Available: https://www.usenix.org/conference/fast23/presentation/zhong

[59] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, ser. SoCC '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 143154. [Online]. Available: https://doi.org/10.1145/1807128.1807152

[60] Linux Foundation and Valkey Contributors, "Valkey documentation – persistence," 2024–2026. [Online]. Available: https://valkey.io/topics/persistence/

[61] ——, "Valkey benchmark," 2024–2026. [Online]. Available: https://valkey.io/topics/benchmark/

[62] Linux man-pages project, *io_uring(7) — Linux Programmer's Manual*.

[63] S. Sanfilippo, "Redis in-memory data structure server," https://redis.io, May 2009.