Karlsruhe Institute of Technology

Impact of Read Operations for Transparent DAX Mappings

Bachelor's Thesis
submitted by

Claas Bastian Wittig

to the KIT Department of Informatics

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

05. March 2026 – 06. August 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 6, 2026

Abstract

Compute Express Link (CXL) is a recent interconnect standard providing new possibilities for system expansion such as extending system memory without occupying additional DIMM slot capacity. From these possibilities, hybrid SSDs emerged, which provide persistent memory semantics in addition to block-based interfaces. Hybrid SSDs implement this using a small on-device memory cache that they fill from persistent backing storage if the requested data is not in cache, introducing CPU stalls on cache misses. Previous research showed that the OS needs to actively manage the on-device cache when it is used with DAX, because DAX was not designed with the CPU stalls in mind that hybrid SSDs exhibit. *Transparent DAX mappings* (TDMs) address this issue while also providing unoptimized applications with the improvements that hybrid SSDs persistent memory semantics offer by transparently handling read and write requests through DAX mappings. Evaluation of TDMs focuses on handling write operations, which is why this thesis investigates what impact handling read operations can have.

To answer this question, we design tooling for observing the read and write system call usage and incurred read amplification. During analysis, we focus on detecting scenarios with read amplification, high read bandwidth, and recurring access patterns. Based on these findings, we sketch four policy ideas, which select what files or file regions TDMs handle. We evaluate the two policies that focus on alleviating system call overhead with MariaDB as the application driven by the *Yahoo! Cloud Serving Benchmark*, using our TDM prototype, which emulates hybrid SSD access latencies with Intel Optane DC Persistent Memory.

We observe during analysis that none of the three investigated applications induce substantial read amplification the operating system can detect. This suggests TDMs are not well suited to alleviate read amplification with these applications. We attribute the evaluation results to alleviated system call overhead. When handling read and write operations, TDMs cause up to 10 times higher benchmark throughput with read-and-write workloads and 6 times higher throughput with read-only workloads. Even though hybrid SSD emulation exhibits different characteristics than a real device, the performance improvement indicates that read operations can affect the application's performance when TDMs handle them.

Contents

Chapter 1

Introduction

Compute Express Link (CXL) introduces new possibilities for system expansion in comparison to established protocols like PCIe [25]. It is backwards-compatible with PCIe and adds protocols for memory expansion and cache coherence through `CXL.mem` and `CXL.cache` [25]. This combination of cache coherence and memory expansion enables new hardware devices that expand system memory, without using any additional DIMM slot capacity [25]. The CPU then can access the device memory through CPU load and store instructions, while also using the CPU cache hierarchy [25].

Hybrid SSDs emerged from those new possibilities, combining memory-based and traditional block-based access semantics [39]. Known implementations integrate a cache on the device, which makes byte-granular access with low latencies possible [54, 36]. The hybrid SSD fills the cache on demand from slow block-based backing storage or the operating system explicitly preloads file contents to the cache [54, 36, 72, 106]. Hybrid SSDs combined with the *Global Persistent Flush (GPF)* [25] feature of CXL can provide persistence to the cache storage, making hybrid SSDs persistent memory devices. Notably, hybrid SSDs are expected to have similar cost to NVMe SSDs ($0.20 vs. $0.23 per GB for Samsung CMM-H and NVMe) [106]. Consequently, because they carry no substantially higher cost, hybrid SSDs could serve as a drop-in replacement for NVMe SSDs instead of a new tier in the storage hierarchy [54]. Intel *Optane DCPMM* [49, 26] is an earlier introduced technology than hybrid SSDs, which also has persistent memory access semantics, but has a higher price than hybrid SSDs and uses DIMM slot capacity to make the persistent memory available [26]. Intel Optane persistent memory devices provide uniform access latencies across their entire storage capacity. In contrast, hybrid SSDs have non-uniform access latencies, caused by differences between cache hits and misses, which introduce intolerable CPU stalls [54, 36, 72].

Direct Access for files (DAX) [53] is an API offered by the Linux kernel designed for storage with byte-granular access across the entire device, such as

Intel Optane [39]. It provides no option to prevent CPU-stalls for storage with different access latencies, like hybrid SSDs [39]. If hybrid SSDs are supposed to be used with DAX, the operating system must actively manage the on-device cache to prevent CPU stalls and make the access latencies more uniform [39]. This creates a need for a new way to interact with and manage the on-device cache of such hardware devices [39].

Khalil et al. [54] propose *Transparent DAX Mappings* (*TDM*) as one solution to control these non-uniform latencies without the need to alter user applications. They accomplished this by overriding the existing `read` and `write` `libc` functions while keeping the interface unchanged. The kernel module of *TDM* manages the on-device cache and maps the contents of the cache into the user application's address space [54]. When the application performs a read or write operation to a file that is mapped, the overwritten `libc` function accesses it directly without involving the kernel [54]. Remaining, non-mapped files are accessed through traditional system calls using block-based methods [54]. The kernel module maintains profiling data about each file through reports from `libc` and automatically chooses files for mapping, based on a policy [54].

They evaluate the proposed mechanism with Valkey [100], an in-memory key-value store that persists data changes to an append-only file [54]. As a load generator, they use the *Yahoo! Cloud Serving Benchmark* [20, 54], which shows a possible performance improvement between 65.1 % and 96.2 % for write-heavy workloads compared to their hybrid SSD prototype without *TDM* [54]. However, they focus solely on the write file I/O path and the prototype does not modify read operations. Consequently, they did not identify a performance difference using *TDM* with read-only workloads [54].

Whether extending *TDM*'s mechanism to `CXL.mem` reads would produce a similar effect remains unexplored. Byte-granular access through `CXL.mem` transfers only the data an application actually requires, reducing the amount of data moved compared to traditional block-based access [36]. Applications that only read part of a file system block could see faster access from this reduction, while the resulting decrease in general bus load benefits all users of the bus [36]. Gervasi and Chang [36] point out that these effects are most pronounced in scenarios where an application only requires a small fraction of the data within a block, which otherwise would result in high read amplification. One example they use is persistent key-value stores with small keys and values, which could cause read amplification of up to 97 % if only one key-value pair in a file system block is used [36]. That is why alleviating read amplification through byte-granular accesses could improve performance even more than *TDM* does alone by accessing the mapped files without involving the kernel.

Additionally, read and write operations differ considerably in nature. Write operations carry the additional burden of crash consistency and write-back require-

ments, which developers often control manually through `fsync` calls [3]. Read operations do not share this burden, since the cache replacement mechanism can simply overwrite the data currently held in the cache. This suggests that the scenarios offering the greatest potential for improvement differ between read and write operations. To be able to exploit these differences, new or existing policies could benefit from being designed specifically with read operations in mind.

Taken together, this motivates a closer look at the read operations an application performs and how it interacts with the file system in general. Such an analysis can help form new policies and evaluate existing ones as well as identify which scenarios are worth optimizing through *TDM*. It can also help identify scenarios with high read amplification, which Gervasi and Chang [36] suggest avoiding.

For this reason, we design and implement tooling to trace the file system interactions of applications through operating system interfaces in Section 3.1 and Section 3.2. Then we select a set of applications in Section 3.3 that are expected to interact heavily with the file system during their runtime. This thesis presents and discusses the captured traces in Section 3.3 split by application. Following that, we propose four policy ideas which target better performance through *TDM* and present a way these policies can be evaluated by emulating a hybrid SSD in Chapter 4. Evaluation is presented in Chapter 5 and performed for one policy which aims at improving performance by alleviating system call overhead. Lastly, we discuss the significance of these results with respect to modeling constraints and whether the expected optimization through alleviating read amplification and system call overhead is applicable to the observed applications in Section 6.1.

Chapter 2

Background & Related Work

RomeFS [107] is a file system that is optimized for use with hybrid SSDs. They combine the benefits of memory and block access semantics by splitting the SSD into two regions. The one accessed with byte-granularity is used for metadata and small read and write requests. The other one, accessed with block granularity, is used for large read and write requests. They were able to show a speedup of at least 2x compared to traditional block-based and persistent memory file systems with page rank algorithm as a real world workload [107].

ExPAND [55] is a CXL-optimized prefetcher. It predicts the latencies of the CXL network and calculates prefetch timings. Additionally, a machine learning algorithm predicts the addresses to prefetch. The prediction algorithms are executed on the CXL-device, while the host device only provides the CXL device with required data. Compared to other rule-based and machine learning–based prefetchers, ExPAND brings an average speedup of 4.1x [55]. If the CXL-device supports ExPAND and TDMs, both systems can work together, because a read or write operation with an active TDM looks like a normal read or write operation from the CXL-device, which ExPAND is targeting.

2.1 Transparent DAX Mappings

Compute Express Link (CXL) [24] is an interconnect standard, based on PCIe, which offers memory access semantics and cache coherence, through `CXL.mem` and `CXL.cache`, respectively [25]. As a result it is possible to expand system memory through the PCIe bus and without using up DIMM slot capacity of the CPU, while continuing to use the CPU cache hierarchy [25]. Since CXL version 2.0, a flush-on-fail mechanism which is called *Global Persistent Flush (GPF)* was added. This makes it possible to create persistent memory devices, which are connected through CXL to the system [25]. One type of device emerging from those new possibilities,

called hybrid SSDs, combines memory with persistent background storage [39]. They combine memory- and traditional block-based access semantics [54]. Two announced commercial implementations of a hybrid SSD are Samsung's CXL Memory Module–Hybrid (CMM-H) [54] and Wolley's NVMe-over-CXL [36]. Hybrid SSDs are expected to have similar cost to NVMe SSDs ($0.20 vs. $0.23 per GB for Samsung CMM-H and NVMe) [106]. Both of them will have a much less memory than persistent background storage [36, 54]. Consequently, not all data from the backing storage fits in the cache all at once. As a result, hybrid SSDs have non-uniform access latencies, caused by differences between cache hits and misses, which introduce intolerable CPU stalls [54, 36, 72]. Habicht et al. [39] showed that the existing Linux interface for direct access to files (DAX) [53] can not properly take advantage of the on-device cache of hybrid SSDs. DAX ensures that the kernel does not use the page cache and accesses the storage device directly. It assumes the same access latencies for the entire address range of the device. Consequently, the DAX interface does not have a solution for preventing CPU stalls on cache misses [39]. Additionally, DAX always affects full files. Applications cannot map one part of a file through the page cache and another part directly from the device [39]. To efficiently use the cache, more granular control over which data is requested into the cache is required. Consequently, changes to operating system interfaces are needed to benefit from hybrid SSDs.

Habicht et al. [39] propose a solution, which extends the `mmap` system call, by adding a flag that forces only the mapped file range to use DAX [39]. They integrate this into the kernel by adding a new page type called *DAX pages* to the page cache. The kernel handles them similarly to any other page in the page cache, but they are located in the hybrid SSD's cache [39]. Since *DAX pages* are held in persistent memory, the kernel does not have to perform the `fsync` operation for those pages [39]. This approach improves DAX in combination with hybrid SSDs, because the application has more granular control over which data is held in the device cache, resulting in more efficient use of the cache [39]. To leverage these improvements, applications need to use the new API, which is an extra burden to application developers and not feasible for many old applications [39].

Khalil et al. [54] propose *Transparent DAX Mappings* (*TDM*) as another solution to control cache contents and mitigate non-uniform latencies. In contrast to adding a new API to the operating system, *TDM* aims to improve usage of hybrid SSDs without adding one [54]. In addition, they required *TDM* to never perform worse than NVMe while also improving performance and/or energy use [54]. They accomplished this by overriding the existing `read` and `write` `libc` functions while keeping the interface unchanged [54]. The kernel module of *TDM* manages the on-device cache and maps the contents of the cache into the user application's address space [54]. When the application performs a read or write operation to a file that is mapped, the overwritten `libc` function accesses it directly without

involving the kernel [54]. `libc` accesses the remaining, non-mapped files through traditional system calls using block-based methods [54]. The kernel module continuously receives profiling data from `libc` indicating which file descriptors use how much bandwidth [54]. `libc` calculates used bandwidth with the help of a sliding window in $O(1)$ time overhead [54]. Based on this data, the *TDM* kernel module applies a *TDM* policy to decide which file regions should be loaded into the cache and mapped into the user's address space [54]. Khalil et al. [54] propose a policy that selects the files with the highest used bandwidth while keeping their total size below the on-device cache size. By doing so, the policy aims to affect the largest possible share of transferred data while maintaining uniform access-latency of the cache [54]. Their design of *TDM* also supports file append operations by bypassing the kernel through DAX mappings [54]. To achieve this, the *TDM* kernel module maps a temporary file to the on-device cache, whose size is controlled through profiling data that solely observes append bandwidth [54]. When an append operation occurs, `libc` writes the appended data to the mapped temporary file without immediately involving the kernel [54]. The *TDM* kernel module later moves this data from the temporary file to the correct file asynchronously [54].

They evaluate the proposed mechanism with Valkey [100], an in-memory key-value store that persists data changes to an append-only file [54]. As a load generator, they use the *Yahoo! Cloud Serving Benchmark* [20, 54] with the included workloads. Write-heavy workloads show a possible performance improvement between 65.1 % and 96.2 % compared to the hybrid SSD prototype without *TDM* active [54]. Even without *TDM* active, their hybrid SSD prototype already achieves up to 89.3 % higher throughput than a normal NVMe SSD for write-heavy workloads [54]. However, they focus solely on the write file I/O path and the prototype does not modify read operations. Consequently, they did not identify a performance difference using *TDM* with read-only workloads [54]. They implement their prototype entirely in user space through overriding `read`, `write` and `open libc` functions [54].

2.2 eBPF & bpftrace

To evaluate the performance implications of hybrid SSDs and Transparent DAX Mappings as well as being able to identify useful *TDM* policies, detailed insights into software file system interactions are required. *bpftrace* [9] provides the necessary dynamic tracing capabilities to quantify metrics such as readahead and read amplification. It is a community open source project providing a programming language and related tooling to trace the Linux kernel and user space software [9]. *bpftrace* builds on the eBPF technology of the Linux operating system, which emerged from the previously defined Berkeley Packet Filter (BPF) [61]. Berkeley

Packet Filters were originally defined, as the name suggests, to filter network packets in an efficient and versatile way, by defining their own instruction set, the kernel evaluates for each packet [61]. Extended BPF (eBPF) extends the idea to load a program from user space to execute it in the kernel [86], and adapts it to other parts like the scheduler [32] or system calls [77]. With eBPF it is possible to extend the kernel's functionality without the need to rebuild the kernel or even restart the machine [86]. To support this, the `bpf` system call [28] was added to load, run, and interact with eBPF programs from user space. eBPF programs are run by the kernel in a privileged context, so they can manipulate, extend, and observe the inner workings of the kernel [86, ch. 1]. They are written with the eBPF instruction set, which is supported by LLVM and GCC [86, ch. 3]. To ensure a provided eBPF program does not cause the kernel to crash or stall, the verifier checks whether the program is safe to execute by, for example, ruling out that the program can run indefinitely or crash [29]. This limits the possibilities an eBPF program has and, for example, forbids unbounded loops [29]. If the program is approved by the verifier, the execution happens in most cases with the help of a Just-in-Time compiler (JIT) and in some cases with a kernel interpreter [86, ch. 1]. The JIT compiles the eBPF instruction set into the CPU's native instructions, which minimizes overhead and increases the performance of the eBPF program [86, ch. 1].

bpftrace provides its own language and toolkit to run eBPF programs that focus on tracing the kernel [9]. The project provides a command line tool, which makes it possible to compile, verify, and run eBPF programs with a single command [9]. The *bpftrace* language is inspired by D, the language of the tracing tool *DTrace* [96, 27]. It associates a function with one or more attachment points, including predefined kernel trace points, kernel functions that are not inlined, and hardware events [7]. *bpftrace* automatically determines the argument-types of the attached functions and trace points [96, ch. 2]. For unknown types it is possible to include existing header files to conveniently handle data and, for example, access attributes of structs [96]. To share data between multiple function invocations or between different functions, it must be shared through eBPF maps [86, ch. 2]. The kernel provides eBPF maps as data storage that is accessible from eBPF programs, the kernel, and user space [86, ch. 2]. These maps can behave like various data structures, such as a hash map, an array, or a ring buffer [86, ch. 2]. User applications can access these maps through the `bpf` system call, while eBPF programs have direct access to them [86, ch. 2]. These maps can be used in *bpftrace* scripts similar to normal variables. To make it easy to write tracing software, *bpftrace* provides various helper functions to interact with eBPF maps, read memory (in kernel and user space), print data, and read current system and process states, among others [8]. Some are wrappers around helper functions of eBPF the kernel provides, like `str`, which reads a string from user or kernel space into *bpftrace*'s memory [8]. Others are additions by *bpftrace*, like `uptr`, which marks the given pointer as a user space pointer [8].

Chapter 3

I/O Analysis

As described in Section 2.1, cache memory on hybrid SSDs is very limited. Typically, it makes up only a small fraction of the total capacity [54, 92]. To use *TDM* with hybrid SSDs, mapped file regions must have been previously copied to the cache and stay there for the time the mapping exists. As a result only a small selection of files can be transparently mapped into memory by *TDM* at the same time. This situation raises the question of which files to move into the cache and create *TDM* mappings for. A previously proposed solution to this question was a policy that maps the files with the highest used I/O bandwidth [54]. This approach aims affect as many file operations as possible. Other aspects of file interactions, such as read amplification, the differing characteristics of read and write operations, or the varying bandwidth across the regions of a file, could also influence how well a policy performs.

TDM policies must be chosen with care, because they are crucial for its success. Wrong decisions could waste limited cache space and prevent better suited files from being loaded. Even though reverting a mapping is generally possible, it should be avoided, since it takes time to load the file regions into cache and generates more SSD load. As a result it is necessary to use a well-suited policy with *TDM*.

Identifying suitable policies relies on an idea about how applications perform file operations. Observing file operations of user applications gives insight into which files are frequently used and if they are modified or only read. Additionally, it reveals whether the whole file or only a fraction is interacted with. Observing background storage usage of the operating system is complementary to application file operations. Combined, they can be used to identify unnecessary loads caused by readahead or read amplification or repeated block I/O of the same regions. Read amplification caused by the fixed block-size or, in case the page cache is used, page size is preventable with byte-granular access using *TDM*. Using these observations and statistics makes it possible to choose files based on other attributes than used I/O bandwidth. That is why this chapter is spent analyzing file access patterns

before later deriving policies in Section 4.1 from these findings.

The goal of this chapter is to gather a broad understanding of file access patterns used by real-world applications. Special focus is placed on the detection of readahead and read amplification, since these can be optimized through byte-granular access to the background storage. Additionally, files with a high operation frequency are identified, especially those with small data access per operation, as these are expected to benefit most from alleviating system call overhead. The following sections first present which data is recorded and the associated implementation details. Subsequently, the selected applications for analysis are introduced and lastly their file access patterns and induced background storage usage are explained.

3.1 Tracing Approach

One way to find out how user applications interact with the file system is by observing relevant system calls. A file system is usually managed by the kernel, thus it is possible to look at any interactions through associated system call invocations.

Observing which system calls are used is possible in multiple ways. One way would be to directly manipulate the kernel source code and add some way to extract the collected data for further processing. This approach is very error-prone and requires the whole kernel to be rebuilt and restarted. Additionally, any issue causes the kernel to crash in the worst case. Even though behavioral changes of the kernel are achievable, they are not required and do not justify the issues caused by manipulating such an essential piece of the system. As a result this approach is not suitable for this thesis.

Another alternative would be to use existing user space tooling to observe the interaction with the kernel like *strace* [94] or the Linux kernel audit system [4]. These are able to precisely observe which system calls are executed by which processes and threads [4, 94], while being convenient to use. However, they are limited to recording the system call itself and lack the ability to observe internal kernel events. Observing the internal workings of the kernel is necessary to precisely identify when read amplification occurs or readahead is triggered. *bpftrace*, introduced in Section 2.2, is a tracing language which makes it possible to observe the inner workings of the Linux kernel and file system [96]. The possibility to attach an eBPF program to any not-inlined kernel function enables detection of when a readahead is triggered or a page is loaded from the block device. Thus, *bpftrace* was chosen as the tracing tool for this thesis.

bpftrace does not need to observe every system call to understand file system interactions, but only those which interact with file data. We observe the `openat`, `open`, and `close` system calls to associate file information like the name and

size to the file descriptor during analysis. Additionally, flags passed to the open system call contains information about how the kernel is supposed to handle the read and write operations, for example whether to use the page cache. The `read`, `pread64`, `write`, and `pwrite64` system calls are the most valuable to us. We observe them to understand how the application accesses or manipulates the file contents. Because `read` and `write` do not include the offset within the file as an argument, the applications must use `lseek` to control the target file offset of the operation. Observation of `lseek` provides timing information about when applications change the file offset, relative to the other system calls. We use it additionally to follow the current file offset of each file during analysis, because we do not publish the offset with each operation. We exclude the vectorized variants of these system calls (`preadv`, `pwritev`, `readv`, `writev`) from further analysis because no application investigated in Section 3.3 uses them to interact with files.

Our tracing excludes `mmap` from observation even though it is a widely used system call to read and manipulate files. It is used to map files directly into the application's memory to read and write it like any other memory address. Normal memory reads do not involve the operating system or system calls, thus making them not traceable with *bpftrace*. While observing the `mmap` system call shows which part of a file is mapped into the application's address space, the accessed memory addresses are not visible. Although there is existing tooling to accomplish byte-granular observation of memory-mapped files [60], this thesis does not use it. One reason against this is that *TDM* does not manipulate the `mmap` system call, so these insights are less useful than `read`/`write` system call observations. Another reason is that the observed applications did not use `mmap` that much as we show in Chapter 3 to justify the higher effort of additional required tooling.

We need to trace kernel-internal tracing to get insights into which data is read from background storage. The internal workings of a file system and how it interacts with internal operating system APIs can vary a lot between different file systems. Observing the correct functions and noticing the correct events is hard to do with a variety of different file systems. That is why this thesis will limit any trace files to *ext4* [31] as file system. *ext4* is chosen because it supports DAX out of the box, is a widely used file system, and we are most familiar with it.

To analyze read-amplification later in Section 3.3, we need to observe block loading from background storage. The bio layer is an abstraction layer for block devices, which *ext4* uses to read and write data from and to the background storage [11]. Each bio operation requires one `struct bio` instance, which specifies the details of the operation, like target background storage location and the source or destination in OS memory [11]. Those `bio` structs need to be submitted to the bio layer, which performs the described operation. The kernel provides a predefined trace point when that happens, which the script observes [5].

Looking at the *ext4* source code [30], there are three main reasons which cause

the file system to load data from background storage: direct I/O, *asynchronous readahead*, and synchronous read into the page cache. Direct I/O is caused by read and write operations of an application which opened the file with the `O_DIRECT` flag [70]. This flag tells the kernel not to cache the file contents into the page cache [70]. As a result every file operation from the application causes block I/O [70]. Synchronous reads into the page cache and *asynchronous readahead* are caused when an application opened a file with the page cache enabled, which is the default. When an application requests a specific section of a file and this section is not already present in the page cache, the operating system needs to load these sections synchronously. The operating system does that in multiples of a page. This behavior is called *synchronous readahead* by the kernel [12]. *Asynchronous readahead* on the other hand is there to prevent synchronous reads. The kernel tries to speculate which data will be requested by the application in the future and loads it early. These loads are also initiated in read and write system calls, but are not waited for and are placed into the page cache outside of a running system call. That is why it is possible to read multiple pages at once, without big impact on performance.

Both *asynchronous* and *synchronous readahead* cause read amplification. Read amplification caused by *asynchronous readahead* is grounded in false predictions on which data is needed and can be prevented by disabling readahead altogether. *Synchronous readahead* and the associated read amplification happen because a whole page must be read from background storage to be placed into the page cache. Additionally, the file system must read a multiple of the background storage's block size, which also affects direct I/O. These two factors cause the operating system to load more data than requested. This is only preventable by reading precisely the data required, without being limited by the block size and without the need to load entire pages into the page cache. This is preventable with `CXL.mem` byte-granular access, since the pages do not need to be loaded into the page cache, nor is the operating system bound to block sizes. That is why it is important to differentiate between these three events. Conveniently, the kernel already provides trace points for each of these events that the *bpftrace* script observes [84].

3.2 Tracing Implementation

We capture the relevant system calls with one singular *bpftrace* [9] script. It attaches to each trace point defined in Chapter 4. For each invocation of the trace points, the script records the following metadata:

- the arguments of the system call or kernel function
- the current file size

- the current value of `CLOCK_BOOT` [15]
- the causing `tid` and `pid`

The current file size is exported to capture the maximum size reached during tracing. The remaining metadata reconstructs the chain of events each file operation triggers inside the kernel. The script constructs one JSON Object [10] for each traced event and aggregates all of them into one singular list. We decided against using a JSON library, since these objects rarely change format and it is simpler to just write a `printf` statement with a fitting format. *bpftrace* writes the `printf` output to a file when it is invoked with the `-o` flag [97], which we use to save the JSON output to a file.

bpftrace recognizes events from various processes in the system [86], even though only the ones caused by the observed application are relevant to this thesis. Sifting through those events could be accomplished by filtering by `tid` and `pid`. This approach turned out to be very error-prone and two main issues when tracing applications. First, determining the first `pid` of each application without missing any occurring events poses a challenge. The kernel assigns a dynamically picked `pid` to a process during its creation. Only when the *bpftrace* script knows this `pid` before the first event occurs can it capture all events the process causes. Second, even though every application starts with one process and thread, new children can be created with system calls like `fork`, `clone`, `clone2`, and `clone3`. Since the `clone*` system calls have options to select whether to share open files or to explicitly set `pid` and `tid` [16], these options need to be correctly tracked by the *bpftrace* script. That is why we decided to filter the events by file system name instead. To accomplish this, for each system call the corresponding `vfs_*` kernel function that is traced. The `vfs_*` is called and receives the affected `file` structure from the system call handler. The *bpfscript* event handler uses the `file` structure [35] to filter by the `i_sb` field of the associated `i_node` structure [35] of the file. The script captures the associated system call only if the target file system name matches the one in the `i_node`. The *bpftrace* script, adds the arguments of the system call to the JSON list while handling the associated `vfs_*` kernel function event. The `vfs_*` event handler does not receive the original arguments of to the causing system call. It instead gathers these arguments from eBPF maps previously filled from the system call event handler. eBPF maps are data storages which can be accessed from different event handlers and user space and can behave like hash maps. Section 2.2 explained these in detail. Each system call is split into two separate trace points, one when entering and one when exiting the system call. The return value of the system call is only present in the exit system call, which adds its own entry to the JSON list, resulting in two entries for each called system call. The script captures the `bio` related trace points between the `vfs_*` function with the correct file system as target and the system call exit trace point are invoked.

This filters out any activity which is caused by other processes and threads as well as reads to load new instructions of the running application.

3.3 I/O Pattern Observations

With the tracing approach established in the previous sections, we now analyze the observations of the tracing. The analysis focuses on detecting patterns of system call usage, distribution of accessed file regions, and read amplification. We focus on these aspects because we consider them relevant for *TDM* policy design. *TDM*, as established in Section 2.1, affects only read and write file operations. Consequently, we narrowed the applications down to ones which we expect to exhibit high I/O utilization through these operations. We focus on server applications, since CXL is new and not yet adopted in personal hardware, which is why we expect server software to adopt hybrid SSDs first. The following sections analyze three applications:

- Graph500 [37] (Section 3.3.1)
- Valkey [100] (Section 3.3.2)
- MariaDB [58] (Section 3.3.3)

Each section first introduces the application and explains why we chose it. The tracing setup follows, covering the benchmark we use to generate load and the application-specific settings we configure for each run. Lastly, each section presents the recorded file operations and analyzes them with respect to the aspects listed above.

Some configurations and setup steps are common across all applications, so we explain them once here. First, we record each application once per run configuration. Multiple runs would mitigate noise from other applications on the system and make the timings and delays more representative. However, we do not analyze detailed timings or delays, but only which data is accessed, which is why a single run per configuration is sufficient. Second, prior to each run, we call `sync` [95] and then clear the operating system page cache through `/proc/sys/vm/drop_caches` [76]. This ensures no file contents from previous runs remain in the page cache and that the kernel must load all file contents from background storage [76]. Third, we execute every run configuration of each application once with kernel readahead enabled a second time with it disabled. When enabled, we ensure the readahead configuration is set to its default value of 512 sectors with the `blockdev` command line utility [6]. This configures the kernel to load $512 \times 512\,\mathrm{B} = 256\,\mathrm{KiB}$ around the current read position [81]. Consequently, to disable readahead we set the value to 0. Lastly, we execute all benchmarks on a machine with an *Intel(R) Xeon(R) Silver 4215 CPU* with 2.50 GHz and 128 GiB DDR4 RAM from Micron operating

at 2133 MT/s. It runs Fedora 42 (Server Edition) [33] with the Linux kernel version 6.18.13 as the operating system. We store all files that the applications interact with on a *Samsung SSD 990 EVO Plus* with 1 TB of capacity.

3.3.1 Graph500

Graph500 is a benchmark designed to evaluate the performance of supercomputers on graph processing workloads [37]. The benchmark was established to provide insight into how well current supercomputers handle data-intensive computing tasks, which are becoming increasingly prevalent in fields such as social network analysis, machine learning, and cybersecurity. Graph500 provides a reference implementation which showcases the required behavior and algorithm which is also suited to be run on a much smaller machine [38].

Several factors motivate the choice of Graph500 for this analysis. First, graph processing workloads need to handle a very large graph structure that may not fit into system memory all at the same time. We assume this causes the application to access the file multiple times. Second, the reference implementation of Graph500 is written in C and uses MPI for parallel processing, which aligns well with the tracing approach described in the previous section [38].

Tracing Setup

Graph500 performs a breadth-first search and a single-source shortest path on a randomly generated graph using multiple processes [38]. The generated graph is shared across all processes for processing [38]. We configure Graph500 [37] with the `TMPFILE` [38] environment variable, to write the generated graph to a temporary file. Graph500 then uses this file to broadcast the graph data to all processes. Graph500 uses the Message Passing Interface (MPI) [63] C interface providing helper functions to run multi-process applications [38, 63]. For tracing, we run the application with a different number of processes. It is possible to specify a graph scale and edge factor. The graph scale modifies the amount of vertices in the graph. The total amount of vertices is calculated as: $\texttt{nvertices} = 2^{\texttt{scale}}$. The total amount of edges is calculated by multiplying the edge factor with the number of vertices: $\texttt{nedges} = \texttt{nvertices} \times \texttt{edge\_factor}$. Although the default edge factor is 16, we define a scale of 24 for tracing, because it proved to be a good compromise between reasonable runtime and big file size. We consider a big file size important to be able to observe a high amount of file operations. Table 3.1 shows each different configuration we run the tracing with.

No.	Processes	readahead	Pre-generated Graph	Scale	Edge Factor
1	2	on	no	24	17
2	2	on	yes	24	17
3	8	on	yes	24	17
4	2	off	no	24	17
5	2	off	yes	24	17
6	8	off	yes	24	17

Table 3.1: Tracing configurations executed with Graph500

Observed File Access Patterns

Graph500 was chosen to represent parallel graph algorithms. It uses the file system to share the generated graph with all processes [38]. First the graph is generated by a singular process which writes the generated graph to a file [38]. Next, when processing starts, each process reads its own part of the graph into memory. Graph500 issues all file operations through MPI library functions rather than calling glibc wrappers or system calls directly [38]. It then performs multiple breadth-first search algorithm passes from different start nodes entirely in memory [38].

When looking at the traced file access patterns, this behavior can be observed through all variants and is displayed in Figures 3.1 to 3.3. The figure displays all file operations of all processes combined, so when a file region is accessed one time, it means only one process touched this region. Since each file region is read two times at maximum even with eight concurrent processes, this means the graph is split between all processes and graph data is only read when required. Graph500 reads 96 MiB with each read operation. Looking at what data the kernel

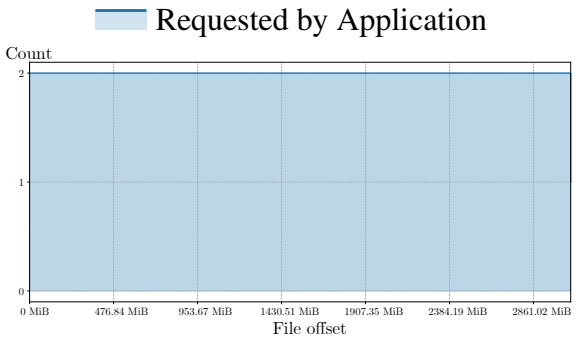


Figure 3.1: In configuration 1, the application accesses every file region twice, while the kernel never loads it from background storage because it is already present in the page cache.

loads from the background storage produces two insights. First, the runs without a pre-generated graph do not cause any reads or writes to the background storage as seen in Figure 3.1. This is because the graph data was written to the page cache right before reading it. The kernel consequently reads the data from the page cache rather than from background storage. The file is opened with the `MPI_MODE_DELETE_ON_CLOSE` [63, p. 492] option, which causes the file to be deleted on close. This potentially prevents any other operations on the background storage. This thesis does not observe swapping of memory pages. If the kernel writes pages to background storage to free up system memory, the figure does not show it. Looking at the runs with pre-generated graphs in Figures 3.2 and 3.3,

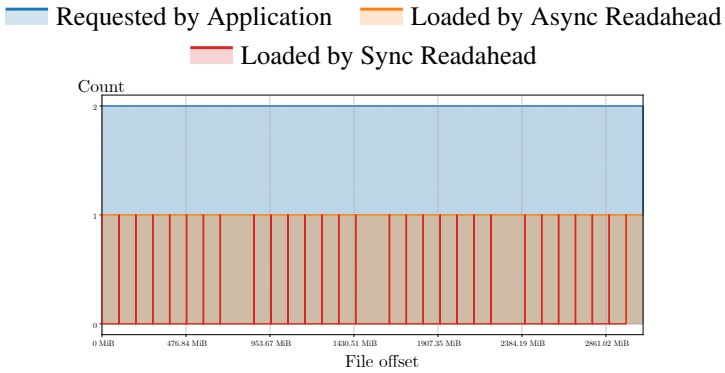


Figure 3.2: In configuration 3, the application accesses every file region twice, while the kernel loads it once: a small part synchronously and the majority asynchronously.

read operations to the background storage are visible. All these operations cause the resulting data to be cached in the page cache, which is why each file region is only read once from background storage. The kernel reads some small regions of the file with synchronous readahead to satisfy the read system call. As seen in Figure 3.2, the kernel loads most data into the page cache with asynchronous readahead. Section 2.2 explained the difference of asynchronous and synchronous readahead. Every duplicate read operation from the application is served by the page cache. Readahead does not cause any unnecessary loads from background storage, because Graph500 reads the entire file. Readahead reduces the amount of data the kernel needs to load synchronously, which changes the delay of the individual read system calls.

Analysis shows that Graph500's use of file operations is very minimal and fairly optimal. Usage of the page cache prevented any duplicate reads from the background storage. By aligning each read to page cache and file system block sizes, Graph500 avoids read amplification, and its large read size minimizes system call overhead.

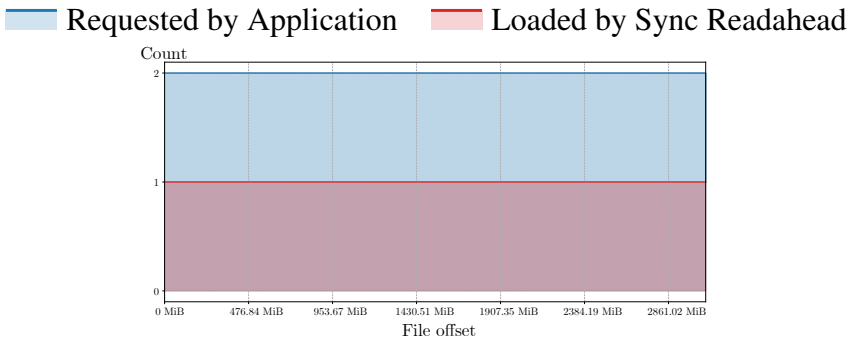


Figure 3.3: In configuration 5, the application accesses every file region twice, while the kernel loads it once and entirely synchronously because readahead is disabled.

3.3.2 Valkey

Valkey [100] is an open source key-value database, which aims to have high performance [100]. Valkey is a fork of Redis [85] and was created from Redis version 7.2.4 [99]. Valkey keeps all data in memory do achieve fast response times, while offering two strategies to persist data [101]. Valkey has multiple fields of application, for example as a database, a cache, or a message broker [101].

Khalil et al. [54] used Valkey to evaluate the first *TDM* approach described in Section 2.1. We chose Valkey for I/O analysis because they demonstrated performance improvements by handling write operations with *TDM*. This result led us to suspect that handling read operations could also benefit from using *TDM*.

Tracing Setup

We use the *Yahoo! Cloud Serving Benchmark* (YCSB) [20] to generate load that Valkey [100] can process. We employ all workloads YCSB provides, which differ in the types of operations they perform [22]. YCSB splits the workloads into two stages, which we trace separately. The first stage is the loading stage, which inserts all entries into the data store [22]. The second stage is the running stage, which performs the insert, read, update, and delete operations on the prefilled data store [22].

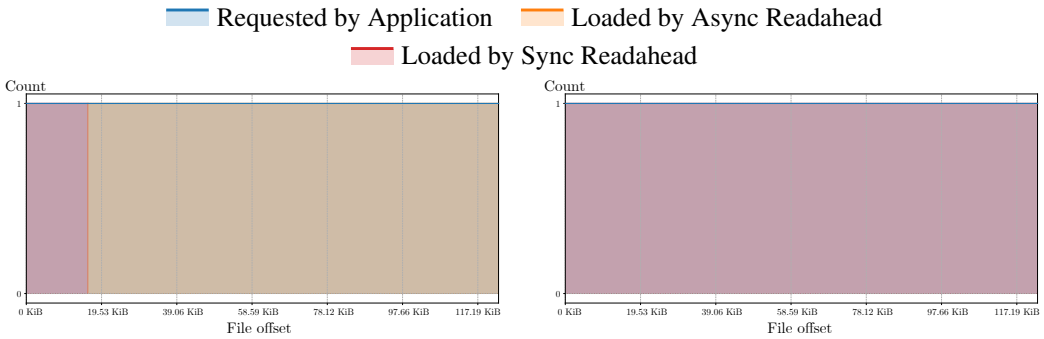
Valkey provides three persistence options: append-only file (AOF), Valkey database (RDB) persistence, and no persistence at all [102]. With AOF persistence, Valkey appends every change to the append-only file and persists them with `fsync` in a configurable interval [102]. The default interval is one `fsync` every second. By default, Valkey rewrites the append-only-file when it has doubled in size and reached at least 64 MiB in size [102]. With RDB persistence, Valkey creates a snapshot that contains the whole key-value store in a configurable interval [102].

The default interval is one snapshot every hour if at least one change occurred, every five minutes if at least 100 changes occurred, and every minute if at least 1000 changes occurred. With persistence disabled, Valkey holds all data in memory, without writing it to the file system [102]. To get a complete picture of file usage with all configurations, we run the YCSB workloads with each possible persistence option alone. Valkey also supports combining AOF and RDB persistence [102]. We did not trace these combinations, because these two options do affect each other and isolated observation of each is therefore sufficient. We leave the snapshot interval of RDB persistence and the `fsync` interval of AOF persistence at their default configuration. Additionally, we trace each persistence configuration once with enabled readahead and a second time without readahead. This results in a total of six run configurations for each included workload of the YCSB benchmark [22]. We increased the record and operation count to 10,485,760 and 5,242,880 respectively for each workload [21]. These settings create enough operations and produce a long enough runtime to trigger the snapshot mechanism and to capture `fsync` operations.

Observed File Access Patterns

We split the files Valkey interacts with during tracing into two groups: the first group consists of config files, the second of files that Valkey uses to persist stored entries. We discuss these groups separately, starting with the configuration files.

Valkey accesses all configuration files the same way in all traced workloads. For this reason, we discuss all configuration files together and distinguish only between enabled and disabled readahead. Configuration files are completely read sequentially once at startup. Valkey does not disable the page cache with `O_DIRECT` [70], which is why the kernel copies these files into the page cache. All read requests use a size of 4 KiB, which matches both the system page size and file system block size. As we already observed in the analysis of Graph500 in Section 3.3.1, files which the application reads completely using the page cache trigger one read from the background storage for each file region. This is again visible in Figure 3.4 for the configuration files Valkey interacts with. The sole difference between enabled readahead in Figure 3.4a and disabled readahead in Figure 3.4b is whether the pages are loaded synchronously or asynchronously. These differences affect system call latency, but not incurred read amplification. Because the files sizes are not multiples of 4 KiB, the file system only partially fills the last file system block. The file system always reads entire blocks. This is why it is forced to read more data from the background storage than necessary. The total accumulated file size of all config files is around 200 KiB, changing slightly depending on which configuration is selected. Because of the unaligned file sizes, the kernel loads 8 % more data than would be necessary.



(a) Readahead enabled

(b) Readahead disabled

Figure 3.4: The configuration files are accessed identically across all workloads. The majority of data is loaded asynchronously when readahead is enabled, while the entire file is loaded synchronously when readahead is disabled.

Next we discuss Valkey's file access patterns with the persistence files. Those files are handled differently whether RDB snapshots or AOF persistence is configured. If RDB snapshots are enabled, the already existing snapshot is read once on startup sequentially. We observe the same read behavior as with the configuration files shown in Figure 3.4. All reads have a size of 4 KiB and Valkey reads the entire file once. When Valkey creates new RDB files, it writes sequentially to a temporary file in the same directory. The write operations all have a size of 4 KiB, except the last one, which varies in size. Once Valkey finished writing the entire file, it is renamed to the configured RDB file name and replaces the old snapshot. Valkey does not read these new files until the next restart. We assume this behavior ensures that the official snapshot on disk is not corrupt, due to a crash while Valkey writes it.

The behavior we observed with AOF persistence enabled, differs from the remaining file interactions of Valkey. The AOF persistence mode uses two files to persist data: a base RDB snapshot and one append-only file which tracks each change since the last snapshot was created [102]. On rewrite Valkey creates a new base and a new append-only file. The base file is created the same way as the RDB files we described previously and combines both old base and append-only files [102]. Except from one complete sequential read at startup, again with 4 KiB operations, Valkey only writes to these files. The write operations to the append-only file produce a consistent write bandwidth during the entire workload. These vary between 73 B and 9 MiB. The described file interactions are similar between all traced YCSB workloads [22]. They differ in three aspects: the write bandwidth to the append-only file, the frequency at which Valkey rewrites the append-only file, and the frequency at which Valkey creates new RDB snapshots. With read-only workloads Valkey does not write to the append-only file.

Analysis of Valkey reveals that it causes read amplification only because of files that are misaligned with file system blocks. Because Valkey issues read operations solely during startup and reads each file just once, we suspect they have little impact on performance. Every file access pattern occurs sequentially without overlap.

3.3.3 MariaDB

MariaDB is a relational database management system (DBMS), created by forking the MySQL source code [58]. MariaDB supports multiple storage engines which can be chosen per table [93]. Different storage engines support different features and have different file system usage patterns [93]. There are storage engines which store the data only in memory, some which reduce I/O operations, and some which are general purpose [93]. InnoDB is a general-purpose storage engine and optimizes workloads which mix read and write operations. Aria is one of two storage engines which are used for system tables. It optimizes read-heavy workloads [93].

We selected MariaDB as an application to investigate, because as a DBMS we expect it to exhibit high I/O usage with a storage engine that is not in-memory [23]. Additionally, highly optimized storage engines usually avoid `mmap` for handling storage data [23]. This matters because our tooling does not trace `mmap`, as discussed in Section 3.1. Lastly, a DBMS is a common server application, which fits the focus of this thesis. We chose MariaDB as a representative DBMS because of its ease of use and availability as an open-source project [58].

Tracing Setup

We use the same benchmark as with Valkey [100]: the *Yahoo! Cloud Serving Benchmark* (YCSB) [20] to generate load that MariaDB [58] processes during tracing. YCSB provides a set of core workloads, which perform different types of operations [22]. It splits the workloads into two stages. The first stage is the loading stage, which inserts all entries into the data store [22]. The second stage is the running stage, which performs the insert, read, update, and delete operations on the prefilled data store [22]. We do not analyze the load stage with MariaDB, because the resulting traces became too big for processing. The same reason holds for workload E of YCSB, which is why we investigate only workloads A, B, C, D, and F.

With MariaDB the user can choose for each table which database engine controls the storing of the table entries. We selected InnoDB as the database engine for tracing because it is a feature-rich general-purpose database engine that MariaDB suggests for most use cases [93, 14]. Additionally, MariaDB uses it as the default for newly created tables, which leads us to expect broad usage. For tracing, we have disabled `innodb_doublewrite` [47] and `innodb_use_native_aio` [47].

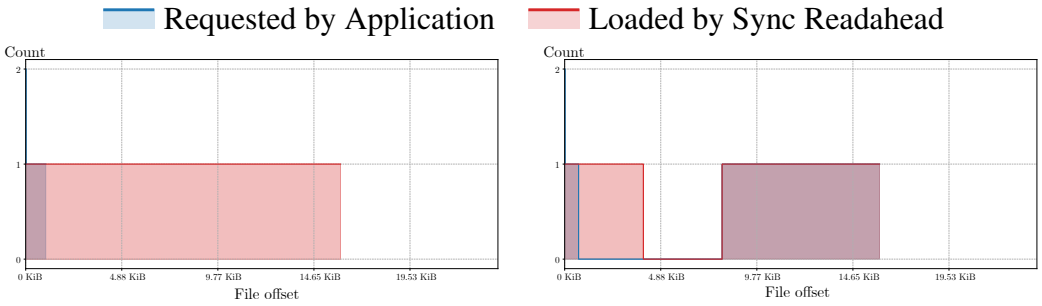
If `innodb_doublewrite` [47] is enabled, InnoDB would write all data to a buffer file first, before writing it to the target file [42]. This provides a copy of the data to recover from, if the write operation to the target file was prematurely terminated [42]. Disabling `innodb_use_native_aio` [47] prevents InnoDB from using the `io_uring` [50] interface of the Linux kernel, which our tooling from Section 3.1 does not observe. We disabled this option to capture more of the I/O operations InnoDB performs. Otherwise, there are no additional arguments passed to the provided startup script `mariadbd-safe`. Like with Valkey, each workload of the YCSB workloads [22] is run once with OS readahead enabled and a second time with readahead disabled. The benchmark is configured to insert 20,971,520 entries into the table before the benchmark and tracing starts. This creates a database size of 33 GiB. We chose this size to keep the benchmark runtime manageable while remaining a larger scale needed to observe file interactions comprehensively. The operation count of the run stage performs 5,242,880 operations, which is exactly a quarter of the inserted entries. This operation count affects fewer entries than the database contains. We chose this count to increase the chance of observing read amplification and to expose read patterns more clearly.

Observed File Access Patterns

bpftrace captures 65 different files MariaDB interacts with during the YCSB workloads. MariaDB keeps track of table definitions with *frm* files [82, ch. 1], of which 21 files are present in the captured traces. The database reads all of these files completely in a sequential manner without bypassing the page cache. Sizes of these files vary from 972 B to 10,383 B. None of these files is aligned with the file system blocks or system pages, which are 4 KiB in size. Every file with a file size that is not a multiple of this block size forces the kernel to load more data than required from background storage. This mismatch causes the kernel to load 2.5 times more data than the application requested. In 5 of these files MariaDB reads some sections twice, but because of the page cache this does not cause any duplicate reads from the background storage. In total MariaDB issues 48 read operations over all of these 21 files, and each file is read at most 4 times during each workload. Even though files with *frm* extension make up 32 % of the total file count, their attributed I/O operations are negligible compared to table data files described in the following paragraphs.

Beyond schema metadata files, MariaDB uses database engines to handle table data [93], as already mentioned in Section 3.3.3. Aria is one of them and MariaDB uses it primarily for system tables, but users can also choose to use it with any table they create [93]. System tables store user information, privileges, time settings, and other global database configuration and statistics [98]. Aria stores table data in files with the *MAI* and *MAD* extensions [2]. These 28 files make up 43 % of all

observed files, while their combined size of 312 KiB accounts for only 0.001 % of the total file size. We first discuss *MAI* files, which store the metadata and index of the table. The observed sizes of those files are always a multiple of 4 KiB and range from 8 KiB to 24 KiB. Most of these files exhibit the same read pattern, for which Figure 3.5a shows *db.MAI* as a representative example. The sole exception to this read pattern is *tables_priv.MAI* shown in Figure 3.5b, which receives one additional read operation. The first read always has a size of 24 B while the second varies in size and lies between 448 B and 1046 B. In all cases, Aria does not bypass the page cache for these files. Consequently, the overlapping 24 B do not cause duplicate reads from background storage. As already mentioned, Aria deviates from this behavior for *tables_priv.MAI* by issuing a third 8 KiB read operation that starts at offset 8192. When readahead is enabled, because of the first and second read operation the kernel had already loaded the data requested by the third read. For all of those files, *bpftrace* records read amplification, as illustrated in Figure 3.5. On average, the kernel loads 15 times more than required with readahead enabled and 6 times more without readahead.

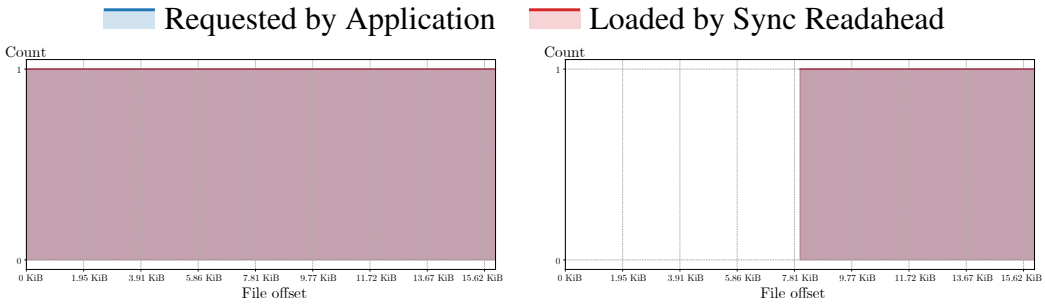


(a) *db.MAI*: Readahead enabled (b) *tables_priv.MAI*: Readahead disabled

Figure 3.5: Aria reads the *MAI* files identically across all workloads. If readahead is disabled, the kernel incurs less read amplification. When readahead is enabled, the last read operation of *tables_priv.MAI* is already present in the page cache.

Aria uses files with the *MAD* extension to store the contents of the table [2]. In contrast to files with the *MAI* extension, Figure 3.6a shows that Aria reads each *MAD* file exactly once and in its entirety during startup. Again, Aria deviates from this behavior for *tables_priv.MAD* by skipping the first 8 KiB of the 16 KiB file and reading only its second half, as Figure 3.6b shows. Each read operation has a size of 8 KiB and does not bypass the page cache. Because each file size is a multiple of 8 KiB and the read operations align with the system page and file system block size, *bpftrace* records no read amplification. Notably, *bpftrace* does not capture any write operations to *MAD* and *MAI* files during the YCSB workloads, and Aria has the same read behavior in each workload. YCSB performs

no table schema modifications or user privilege changes during execution, which explains the minimal file interactions observed.



(a) *db.MAD* (b) *tables_priv.MAD*

Figure 3.6: Aria reads the *MAD* files identically across all workloads. It reads *tables_priv.MAD* partially, whereas it reads each of the remaining files once and in its entirety.

Apart from Aria, *bpftrace* also captured file interactions of InnoDB [44], which is the storage engine we chose for the table YCSB uses. InnoDB creates one file per table with the *ibd* file extension if the `innodb_file_per_table` [47] option is enabled [43]. Unlike Aria, InnoDB combines the table schema, index, and data in a single file [43]. Aside from the table files, InnoDB uses on-disk undo logs to implement transactions [48]. In addition to the undo log, the storage engine maintains a redo log as a file on disk to ensure data is written reliably even in the event of a crash [45]. Lastly, InnoDB uses an *ibdata1* file to save data that cannot be attributed to one specific table, but to the system as a whole [46]. We exclude the redo log file from I/O analysis, since InnoDB memory-maps the file during tracing. The *bpftrace* script does not correctly observe the `mmap` system call, as described in Section 3.1, thus making the analysis incomplete with respect to file interactions.

We begin analysis with the undo logs. InnoDB interacts with three of them during the benchmarks, and always passes the `O_DIRECT` [70] flag to the open [70] system call to bypass the page cache. As a result, every file operation causes a direct I/O operation to the background storage. Each of the read operations that target the undo log files has a size of 16 KiB, which is a multiple of the file system block size of 4 KiB. Because these read operations are also aligned with the system pages and file system blocks, *bpftrace* cannot observe any read amplification. When reads align with block boundaries and the page cache is inactive, the file system loads exactly the requested amount of data without additional overhead from readahead or page-size mismatches. If read amplification occurs in these files, it must be caused by InnoDB requesting more data than it needs, which the used tooling

cannot observe. Each of the three undo log files has a size of 10 MiB. During most workloads, InnoDB reads five sections more frequently than any other part of the file and never reads the second half, as Figure 3.7a displays. During workload D, InnoDB exhibits this behavior only for the *undo001* file, and reads the beginning of the remaining undo log files more frequently, as Figure 3.7b displays. Workload C does not cause any more frequently read sections, and reduces the read operations to 46, the lowest across all workloads. Read operations peak during workload A, where InnoDB issues slightly more than 3000 of them to the *undo001* file. Generally, the *bpftrace* script captures fewer read operations targeting the undo log files when the workload performs fewer update or insert operations. Workload C has neither update nor insert operations, and workload A performs the highest share of update operations, at 50 %. This indicates that update and insert operations cause InnoDB to read the undo log more frequently. We also observed that the more frequently read sections occur only if the workload performs sufficient update and insert operations. InnoDB reads the *undo001* the most across all workloads, while the read operation count targeting the *undo002* and *undo003* files are comparable. Write operations mirror the pattern of the read operations but occur more frequently overall, as Figure 3.7c shows.

(a) Workload A: *undo001* (read)

(b) Workload D: *undo002* (read)

(c) Workload A: *undo001* (write)

Figure 3.7: InnoDB's undo log files exhibit five frequently read regions across most workloads. Workloads that lack these regions perform a low share of table entry manipulations. InnoDB bypasses the page cache for all of these files.

InnoDB reads the *ibdata1* file in the same way as the undo log files. It also always reads data in sizes that are a multiple of the file system block size and system page size. As with the undo log files, it bypasses the page cache. This alignment again causes our tooling to report no read amplification. In every workload, InnoDB reads the file offset range from 1 MiB to 3 MiB with two 1 MiB read operations. All remaining read operations have a size of 16 KiB and target the beginning of the file. This produces a peak similar to that of the undo log files, as Figure 3.8 shows. The number of 16 KiB reads differs between workloads. Workloads B and D cause the most reads with 949 and 817 read operations respectively. During workload C, *bpftrace* captures 17 read operations, the least across all workloads.

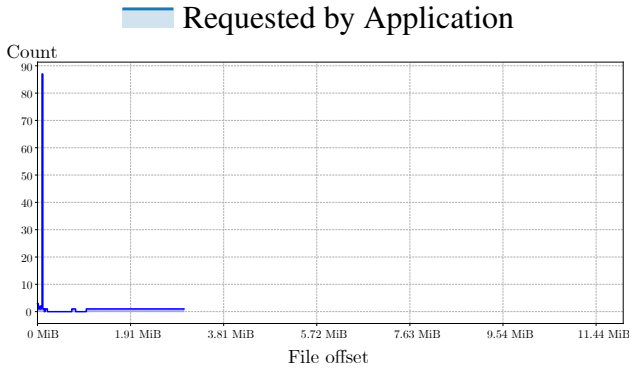


Figure 3.8: Recorded during workload A. InnoDB reads only the beginning of the *ibdata1* file and bypasses the page cache. Across all workloads, workloads B and D read this first region most frequently.

The files with the *ibd* extension, each of which holds one table, are the last remaining file category of InnoDB not yet analyzed. *bpftrace* captured InnoDB interacting with four files. We split these files into two groups for analysis. The first group consists of three files: *innodb_table_stats.ibd*, *innodb_index_stats.ibd*, and *gtid_slave_pos.ibd*. The first two store statistics about the tables and indexes that InnoDB manages [66, 67]. MariaDB uses the last one during replication to keep track of the replica's working position [65]. The second group consists only of the *usertable.ibd* file, which holds the data the YCSB benchmark creates and accesses. The files of the first group share the same access patterns across all workloads, independently of the readahead configuration. InnoDB first reads the whole file with a single 65 KiB read operation, matching the size of these files. This loads the entire file into the page cache. Later in the trace, but still during startup, InnoDB opens these files again, this time bypassing the page cache. Then two reads follow, covering the first 16 KiB and last 16 KiB of the file, as can be seen in Figure 3.9. This causes the kernel to load the start and end of each file twice from background storage.

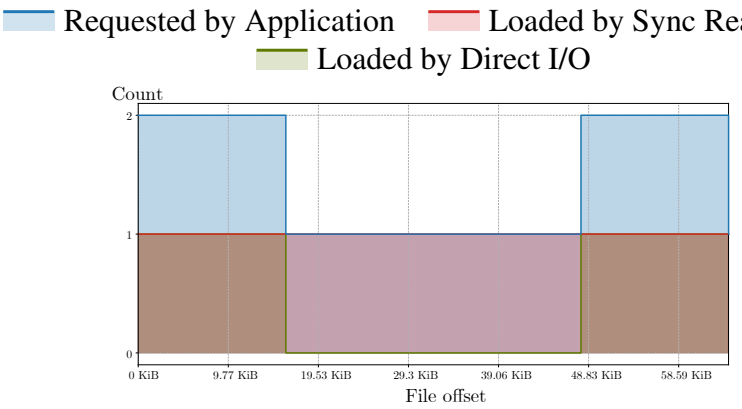
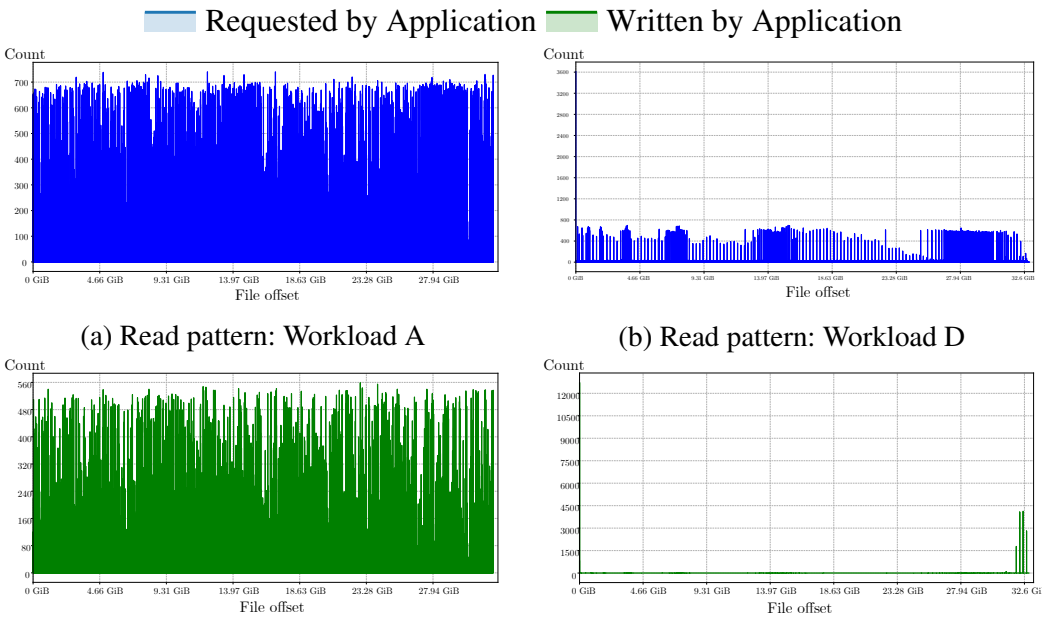


Figure 3.9: The access pattern of *innodb_table_stats.ibd* stays the same across all workloads. InnoDB opens this file twice, once while bypassing the page cache and once without. The first and last 16 KiB are loaded twice as a result.



(a) Read pattern: Workload A

(b) Read pattern: Workload D

(c) Write pattern: Workload A

(d) Write pattern: Workload D

Figure 3.10: The *usertable.ibd* file is the most frequently read file, and InnoDB bypasses the page cache. The read patterns of workload A resemble those of workloads B, C, and F, while only the write patterns of workloads B and C match that of workload A. During workload D, InnoDB reads one region at the beginning of the file more frequently than the rest. Additionally, during workload D InnoDB writes only to the beginning and end of the file.

InnoDB produces the most read and write operations for the *usertable.ibd* file. It reads this file in the same way as the already discussed undo log files and the *ibdata1* file, using 16 KiB read operations, bypassing the page cache, and aligning with file system blocks and system pages. This again causes the *bpftrace* script to be unable to capture any read amplification. InnoDB produces similar file read patterns for the *usertable.ibd* file across workloads A, B, C, and F, independently of the readahead configuration. During these workloads, InnoDB reads every section of the file multiple times, without any visible pattern. As a representative example, Figure 3.10a shows the read pattern during workload A. Workload D is the only workload deviating from this pattern. InnoDB reads the beginning of the file more frequently than the remaining file: it reads the beginning 3611 times while reading the remaining regions 7 times on average, which is visible in Figure 3.10b. Additionally, InnoDB reads substantially fewer bytes during workload D than during workloads B and C as visible in Figure 3.11, even though all three perform a comparable share of read operations: 95 % for workloads D and B, and 100 % for workload C. We attribute this differences to workload D being the only workload we investigated that inserts entries, rather than only updating and reading existing data. Workload E would also have included new entries, yet we excluded it from analysis, as discussed in Section 3.3.3. InnoDB's write operations mirror the pattern of the read operations for workloads A, B, and F, which differ only in the average times InnoDB interacts with each region. As a representative example, Figure 3.10c shows the write pattern during workload A. During workload D, InnoDB only writes at the beginning and at the end of the file, as shown in Figure 3.10d. Workload C causes no write operations to the *usertable.ibd* file, which is also visible in Figure 3.11.

Figure 3.11: Total bytes read from and written to the *usertable.ibd* file during each workload. Workload D reads the least data of all workloads, which we attribute to it being the only workload that inserts data.

MariaDB incurs read amplification with about half of the files it interacts with. All of these files are small, so the data loaded through read amplification amounts to only about 0.0002 % of the total requested data. For any file with the

ibd extension, the undo log files, and the *ibdata1* file, our tooling cannot observe read amplification because of the way InnoDB issues the read operations. Among all analyzed applications, only MariaDB produces a read pattern that concentrates on specific file regions instead of reading the file uniformly, the behavior InnoDB exhibits with its undo log files.

Chapter 4

Design and Implementation

This chapter focuses on the design of *TDM* policies. It presents four such policies, explains how we expect each of them to work and introduces tools for evaluating them. Designing such policies requires care, as wrong decisions could waste limited cache space and prevent better-suited files from being loaded.

In Section 3.3 we investigated the different file access patterns of multiple analyzed applications. The analysis reveals two central insights that shape the design of *TDM* policies focused on read operations. First, no investigated workload performed on any application led to significant read amplification. Second, most files are read roughly the same number of times over their entire size, with one exception of the undo log files MariaDB uses, entirely discussed in Section 3.3.3. In turn no file sections that can be best handled through *TDM* emerged from analysis. As a result of these findings, we assume policies focusing on read operations benefit most by alleviating system call overhead.

Building on these insights, Section 4.1 discusses *TDM* policies that could improve performance of the analyzed applications through read operations. The goal of designing these policies is to find out what effect handling read operations through *TDM* can have in practice.

After establishing which policies are possible, Section 4.2 explains how we emulate a hybrid SSD by overriding `libc` functions, since no real hybrid SSD is available for evaluation. This emulation has two major drawbacks: it does not handle write operations correctly, and append operations are not supported at all. Left unaddressed, these issues would cause undefined behavior in the applications under test. To work around this limitation, Section 4.3 proposes a replay mechanism that reproduces a previously captured workload. This replay mechanism does not depend on correct data being written to or read from the hybrid SSD, making it possible to use the emulated hybrid SSD with more workloads and policies. We evaluate one of the introduced policies using the emulated hybrid SSD in Chapter 5.

4.1 TDM Policies

The analysis of the traced applications in Section 3.3 showed that total read amplification is limited and that the I/O bandwidth used by the analyzed workloads concentrates on a small collection of files. Neither benchmarking Graph500 [37] nor Valkey [100] led to significant read amplification. MariaDB [58] caused read amplification in about half of the files it interacts with. All of these files are fairly small, so the total amount of data loaded due to read amplification amounts to only about 0.0002 % of the total requested data and only about 0.0007 % of total file size observed. The analysis further showed that most files are read roughly the same number of times over their entire size. As a result, making general assumptions about which file regions to map is not possible, because the analysis did not reveal a sufficiently general pattern across enough files. One exception is InnoDB, one of MariaDB's storage engines analyzed in Section 3.3.3. With most workloads, InnoDB read five sections of each undo log file much more often than the remainder of the file. As neither reducing read amplification nor addressing imbalanced read bandwidth over the whole file size offers possibilities to design *TDM* policies, we target our design on alleviating system call overhead through *TDM*.

4.1.1 All Files Policy

The **All Files Policy** serves as a baseline for identifying the maximum improvement achievable through kernel bypass. It maps every file, disregarding the hybrid SSD's cache-size limit. We expect this policy to improve MariaDB's performance, since it issues many read and write operations during runtime as shown in Section 3.3.3. In Section 3.3.1 we showed that Graph500 issues few I/O operations during runtime and should therefore be affected only marginally. Valkey's write operations are dominated by append operations during runtime if the append-only file persistence mode is configured as we already discussed in Section 3.3.2. The impact of performing those append operations through *TDM* has already been evaluated with significant demonstrated performance improvements [54], as mentioned in Section 2.1. That is why we expect Valkey would benefit from this policy in environments that support mapping appended files through *TDM*. Remaining file operations of Valkey are minimal and are not expected to affect performance much. Applying this policy to a real hybrid SSD is unrealistic, because it may map more files concurrently than the on-device cache can hold. This introduces CPU stalls whenever it accesses files that are not currently present in the cache, because they must first be loaded from the backing storage, as discussed in Section 2.1. It is therefore only usable on the emulated hybrid SSD introduced in Section 4.2, which has no cache size limitation, or in settings where the required cache size is known in advance. On the emulated hybrid SSD, this policy additionally cannot

map files that are appended to during runtime, resulting in only minimal expected performance impact for Valkey.

4.1.2 No Big Files Policy

Unlike the *All Files Policy (4.1.1)*, the **No Big Files Policy** targets a more realistic cache size, making it applicable to real hybrid SSDs. To this end, it excludes very large files from *TDM* mapping. This filtering strategy is motivated by the analysis of MariaDB in Section 3.3.3. The analysis showed that most files are relatively small and that the largest file accounts for 99.6 % of the total size of the files MariaDB interacts with. Since we want to reduce the required cache capacity applicable to real hybrid SSDs, the only option is to remove the biggest files if partial file mappings are not supported. Its strategy is otherwise identical to the *All Files Policy*, adding a check of the file size against a threshold value when a file is opened. This threshold could be coupled to the hybrid SSDs cache-size ensuring a certain number of files can be mapped in total. Choosing the threshold value must be done with similar care as the whole policy, as it has a similar effect on *TDM* as already discussed in Chapter 3. We will leave this question unanswered since it requires a more general analysis of system-wide file usage. To analyze the impact of this policy on Graph500 [37], Valkey [100], and MariaDB [58], we assume the threshold only excludes the biggest file MariaDB interacts with. As with the *All Files Policy*, Graph500 is not expected to be affected much and Valkey is only affected if *TDM* handles append operations. For MariaDB, the largest file alone accounts for more than 99 % of all bytes read and read operations recorded during tracing. Excluding it, we expect to reduce the performance increase by a huge amount compared to the *All Files Policy*. For MariaDB, this strategy succeeds in shrinking the cache to a reasonable size, but it does not fully resolve the limited-cache-size problem in general, since the cache can still grow unbounded through many small files.

4.1.3 Small Files Policy

As described in Section 3.3.2, files smaller than the file system block size, or whose size does not align with it, exhibit particularly high read amplification. The **Small Files Policy** targets exactly these files by mapping them with *TDM* whenever their size falls below the file system block size or does not align with it. For files larger than one file system block, mapping only the last block would reduce cache usage while preventing all incurred read amplification due to unused space within a block. As with the first two policies this one does not define what to do if cache is full. However, this scenario is expected to occur less often, since for each file only one block is present in the cache, making it possible to map many files. Because the total

amount of read data affected by this policy is small with the analyzed applications, we expect it to have only a minimal impact on the applications analyzed in this thesis. This policy could nonetheless have a greater impact on a system running many applications simultaneously if the data volume saved across all of them accumulates to a significant amount. Verifying this claim requires a system-wide analysis of opened files, to determine whether the data saved is sufficient to produce a measurable effect, which is not part of this thesis.

4.1.4 Hotspot Policy

As shown in Figure 3.7, certain sections of MariaDB's undo log files are read significantly more often than the remainder of the file. The **Hotspot Policy** targets exactly these sections, mapping only the sections of a file that are read more frequently than a defined threshold value. The number of sections per file can be defined statically, which results in a variable section length depending on file size. Each section could receive its own counter, which is increased for each access to this section. Since the amount of sections is static, the counters require $O(1)$ additional memory for each file. Increasing the counter is linear in the number of affected sections per file operation, bounded by the maximum number of sections. Selecting which file sections to handle through *TDM* could be realized in the same way as files are chosen in the first prototype described in Section 2.1, requiring a runtime of $O(n \log n)$ where n is the number of sections currently open. This is a multiple of the currently open files and can grow quite large, making careful selection of the amount of sections per file necessary. Both the *No Big Files Policy (4.1.2)* and this policy handle the undo log files, because they are among the more frequently read files. For *No Big Files Policy* the undo log files account for 90 % of the bytes read of the files mapped. This is why we expect the *Hotspot Policy* to have a similar effect on MariaDB as the *No Big Files Policy*. Valkey and Graph500 do not read particular sections of a file more often than others. Thus, all or none of these files would be mapped, affecting the performance of the applications in the same way as the *No Big Files Policy*.

These four policies vary considerably in how practical they are to apply on a real hybrid SSD. The *All Files Policy (4.1.1)* is only feasible in an idealized environment with an unconstrained cache and mainly serves as an upper bound for the achievable improvement, while the remaining policies use a more realistic cache size for the applications observed in Chapter 3. We evaluate the *All Files Policy* and *No Big Files Policy (4.1.2)* in Chapter 5. We do not evaluate the other policies since they need partial file mapping support or system-wide mapping to work, which is not supported by the *TDM* prototype used for evaluation. The remainder of this chapter presents the tools used to evaluate the presented policies.

4.2 TDM Prototype

Evaluating these policies requires hardware whose access semantics resemble those of hybrid SSDs, to be in any way applicable or expressive. Additionally, the hardware must be able to map file contents into user address spaces and access the same data through block-based methods. This thesis will implement the *TDM* prototype by emulating a hybrid SSD by combining an NVMe SSD and Intel Optane persistent memory [49]. Persistent memory provides memory access semantics and low access latencies, while the NVMe SSD provides data through block-based methods. Block-based access to hybrid SSDs will be similar to accessing an NVMe SSD [69]. Similarly, access latencies and throughput of Intel Optane persistent memory are comparable to prototypes of CXL-based memory expanders [57, 73]. As a result, we expect this model to be close enough to real hybrid SSD hardware to get an indication on real performance with the policies chosen for evaluation in Chapter 5. In contrast to the original design described in Section 2.1, this prototype will have no kernel module, but entirely work within user space. Because Intel Optane does not require system-wide cache management, and because Chapter 5 tests the policy with only one application at a time, this approach is viable.

The *TDM* prototype replaces a subset of `libc` functions by loading itself with the `LD_PRELOAD` [56] environment variable. When the `LD_PRELOAD` environment variable is specified, the dynamic linker loads the specified library before any other library or application [56]. This makes it possible to override functions of `libc`, because the functions of the specified library precede entries of `libc` [56].

This prototype generally requires the same file system to be present on both the Intel Optane and the NVMe SSD. Both file systems must be mounted to an arbitrary location on the system. Two environment variables communicate these mount points to the *TDM* prototype. When the application calls an open variant, the prototype first checks if the file accessed is a child of the NVMe mount point by comparing absolute paths. If it is, the prototype checks whether it should create a DAX mapping for the file, based on the currently selected policy. The creation of a *TDM* for a file works as follows: The prototype opens the associated file under the Optane mount point by replacing the prefix of the absolute path. Next, it creates a DAX mapping of the entire file. If successful it saves the file descriptor, current file offset, address of the mapping, length of the file, and a mutex [79] to an `m_mappings` array, that is indexed by the file descriptor. In the read and write functions of the prototype, it first checks if the passed file descriptor is in the `m_mappings` array and handles it appropriately. In the case of an active mapping, the prototype performs the operation by calling `memcpy` [62]. Notably, if a write operation appends data to a file, it does not perform the operation and returns an error code. If required, the prototype updates the offset in the `m_mappings` entry.

Otherwise, if the file has no active mapping, the arguments are passed unmodified to the appropriate `libc` function. When the application calls a close function variant, the prototype removes active mappings and closes the file. The mutex [79] that is part of the `m_mappings` entry synchronizes each operation that affects the associated mapping, causing the later thread to wait, if another application is currently performing an operation on that file.

This implementation always maps entire files and does not support append operations. Additionally, this *TDM* prototype can not change whether a file is mapped, since this handling only happens during open and close functions. These limitations are acceptable, because this prototype only aims to provide an indication on the impact of the evaluated policies. Policies which do not use these features should still give a close representation of what impact the policy would have, since the access latencies and behavior of accessing a mapped or unmapped file are similar, as discussed in the beginning of this section. In addition to the missing features of the *TDM* mechanism, the emulation fails to handle write operations correctly and does not enforce a cache size limit. The write operations only fail in one specific scenario: When a file is mapped and the application writes some data, the prototype writes this data to the Optane persistent memory. If the application later closes and reopens this file, and the policy then decides this file should not be mapped, any data the applications then reads from the NVMe SSD does not have the previously written data. This behavior could cause an application to behave incorrectly or crash altogether. To overcome this issue, Section 4.3 describes a workload generator, which replays previously captured `libc` function calls. This replay mechanism does not rely on the correct data being written or read, but aims to generate a similar load, which can be optimized through *TDM*.

4.3 Workload Replay

Section 4.2 introduces the *TDM* prototype used for evaluation and points out that it does not handle write operations correctly, so that as a result subsequent reads an old version of the file. This prevents some applications from working correctly with the presented prototype. To work around this issue, we designed the following workload replay system. Its main goal is to generate a load of file operations similar to that of a previously captured application, while not relying on write operations being processed correctly.

The workload replay system consists of two parts: the **recorder** captures `libc` function usage and the **replayer** re-executes those functions. As the *TDM* prototype does, the **recorder** tracks the `libc` function usage of the observed application by overriding those functions through the `LD_PRELOAD` environment variable. As a result it produces a trace consisting of one entry for each overwritten function with

nanosecond precise timestamps. In addition to each open, close, read and write variant [70, 83, 104, 75, 17], which are also overwritten by the *TDM* prototype, the **recorder** overrides the `pthread_create` [78] function, so it can create the correct amount of threads at the correct time. For each occurring function call, the **replayer** saves following things to the trace in JSON format:

- `pthread_self` [80]
- function name
- function enter nanoseconds (`CLOCK_BOOTTIME` [15])
- function exit nanoseconds (`CLOCK_BOOTTIME` [15])
- all arguments of the function

Each thread writes to its own file, to minimize synchronization effort of writes to the trace file. Based on these traces, the **replayer** software can execute the same `libc` functions.

The **replayer** then reads these files and re-executes the functions in the same order and with the same arguments, except the buffer addresses and file descriptors. For the file descriptors it maintains a mapping from the file descriptor defined in the traces to the file descriptors returned from open during runtime. For each function, the **recorder** captured the enter and exit nanosecond timestamps. From these timestamps, the **replayer** calculates the time between two functions using the following formula:

$$t_{\texttt{between}} = t_{\texttt{next\_enter}} - t_{\texttt{prev\_exit}} \tag{4.1}$$

The **replayer** uses this duration to read the next function arguments from the JSON entry and waits the remaining duration with the `nanosleep` [68] function. It only waits for the time between the start and end of two successive functions to ensure differences of file operations have an impact on the total runtime of the **replayer**.

The **replayer** does not use the written or read data, but discards them. This eliminates the issue, the *TDM* prototype from Section 2.1 has, that written data may not be read back correctly. One notable drawback of this approach is that it does not simulate dependencies between threads correctly. As an example if thread *A* performs a file operation and holds a mutex at the same time and releases the mutex when the operation finishes. In case a second thread *B* waits for this mutex at the same time, thread *B* continues when *A* releases the mutex. The **replayer** is unaware of this dependency. If the file operation of thread *A* finishes faster during replay than during recording, the **replayer** causes thread *B* to wait longer than it would in a real scenario. This is because the time thread *B* waits for the mutex is part of the duration between functions calculated in Equation (4.1). Even with this drawback, the evaluation results indicate performance impact of the *TDM* prototype. Faster

file operations speed up the replay, and other measured data, such file operation length, still provide meaningful insights.

Chapter 5

Evaluation

In Chapter 3 we analyzed three applications with differing I/O usage. This analysis showed that these applications neither cause much read amplification nor show generally applicable read access patterns indicating a focus on specific sections of a file. We used these results in Chapter 4 to propose a set of policy ideas: two that aim to alleviate system call overhead, one policy that aims to reduce read amplification system-wide, and one that tries to better use the hybrid SSD's cache by only mapping sections of a file. These policies differ in practicality and do not define a strategy when the hybrid SSD's cache is full.

This chapter evaluates the two policies that focus on alleviating system call overhead with the *TDM* prototype presented in Section 4.2. Our main goal is to determine how large the impact of *TDM*'s kernel bypass can be. As a workload for the *TDM* prototype, we chose a setup similar to the one in the I/O analysis, using MariaDB [58] as an application and *Yahoo! Cloud Serving Benchmark* (YCSB) [20] as a benchmark. Section 5.1 describes the evaluation approach in detail. We discuss the results of the evaluation in Section 5.2 and state possible explanations for them. Chapter 6 discusses the significance of the evaluation results, the proposed policies, and the identified I/O patterns, together with a possible reason for the low captured read amplification, and an outline of future work.

5.1 Approach

This evaluation tests a subset of the proposed policies from Section 4.1 and determines what impact these policies can have. To accomplish this, we evaluate the *All Files Policy (4.1.1)* and *No Big Files Policy (4.1.2)* with MariaDB [58] as the application, and we run the same workloads once more without *TDM* as a baseline. We expect the *Small Files Policy* (4.1.3) to affect overall performance only when applied to the entire system. The *Hotspot Policy* (4.1.4) requires *TDM* mechanism

to support mapping files partially. The *TDM* prototype from Section 4.2 supports neither being applied system-wide nor mapping files partially. This is why we could not evaluate the latter two policies. We chose MariaDB as the application because it showed a high intensity of file operations, as discussed in Section 3.3.3, so that each policy has enough operations to affect the application's performance. The analysis of Valkey [100] in Section 3.3.2 and of Graph500 [37] in Section 3.3.1 showed that both applications perform a limited number of file operations. We therefore excluded both applications from the evaluation because they are unlikely to affect benchmark performance when *TDM* is added, as already discussed in Section 4.1. We configure MariaDB in the following way:

- We use InnoDB [44] as the active database storage engine. It is the same engine we used during I/O analysis in Chapter 3. This lets us verify our assumptions about how the policies affect the performance of MariaDB.
- We turn off `innodb_doublewrite` [42], which would cause every page InnoDB writes to be written a second time to an additional file.
- We use the `mariadbd-safe` [59] wrapper to start MariaDB without modifying any configuration beyond the above.

These configurations differ from those used in Chapter 3 in one option. During analysis, we disabled the `innodb_use_native_aio` [47] option. Disabling this option prevents InnoDB from using the `io_uring` [50] interface of the Linux kernel. During evaluation, we did not disable this option. This causes InnoDB [44] to issue some of the file operations through the `io_uring` interface. The *TDM* prototype, which we introduced in Section 4.2, returns the original file descriptor used for the internal DAX mapping. Consequently, these `io_uring` file operations target the prototype's file and use the Linux DAX mechanism whenever the policy selects it for mapping.

We chose *Yahoo! Cloud Serving Benchmark* (YCSB) [20] as the benchmark to evaluate the *TDM* policies because, as already demonstrated in Section 3.3.3, it generates sufficient load for MariaDB to perform enough file operations so that *TDM* can affect overall benchmark performance. We excluded workloads D and E [18] of YCSB because they perform insert operations in the database. Insert operations into the database cause file append operations that the *TDM* prototype does not support. For the same reason, we do not run the load phase of each YCSB workload with the *TDM* prototype. YCSB splits each workload into two phases: the load phase and the run phase [19]. In the load phase, YCSB inserts a configurable number of entries into the database [19]. In the run phase, YCSB performs a configurable number of read, insert, read-modify-write, or update operations with the distributions per workload displayed in Table 5.1 [18]. We configure YCSB to insert 20,971,520 entries during the load phase. This causes MariaDB to produce a

table file size of 33 GiB, the same size we generated during analysis in Section 3.3.3. For the run phase, we configure YCSB to perform 10,485,760 operations with 20 concurrent threads. We chose this operation count, because it is exactly half of the inserted entries and causes YCSB to leave some entries untouched. To mitigate any distortions, for example, from other applications or unfavorable scheduling, we run each workload 50 times with the same configuration. The results presented in Section 5.2 are averages of these 50 runs.

Neither evaluated policy dynamically changes the set of files handled by *TDM* at runtime. Consequently, all file operations for the same file either access the DAX mapping or use the block-based path. As a result, the *TDM* prototype described in Section 4.2 always reads from and writes to the same device. This prevents the situation where the prototype writes to one device but reads from the other and would therefore return incorrect data. As a result, the prototype works correctly with the policies used for evaluation. Additionally, benchmarking with the real application yields more realistic results. That is why we decided against using the replay system.

All benchmarks are run on a machine with an *Intel(R) Xeon(R) Silver 4215 CPU* with 2.50 GHz and 128 GiB DDR4 RAM from Micron with a speed of 2933 MHz. It uses Fedora 42 (Server Edition) [33] with the Linux kernel version 6.18.13 as the operating system. All files are located on a *Samsung SSD 990 EVO Plus* with 1 TB of storage space. The *TDM* prototype from Section 4.2 uses one *Intel® Optane™ DC Persistent Memory* (NMA1XXD128GPS) [49] with 128 GiB of total storage.

5.2 Benchmark Results

This section presents measured results of the *Yahoo! Cloud Serving Benchmark* (YCSB) [20] run against MariaDB [58]. We run workloads A, B, C, and F [18] of the benchmark once without the *TDM* prototype from Section 4.2, and once each with the No Big Files Policy (4.1.2) and the All Files Policy (4.1.1). We omit workloads D and E because the *TDM* prototype causes issues with these workloads, as described in Section 5.1. The distribution of read, update, and insert operations for each workload is displayed in Table 5.1. Notably, workload F does not update the entries with a sole update operation, but instead performs a bundled read-modify-write operation, which reads and then updates the same entry [18]. The read operations that are part of the read-modify-write operation are included in the 75 % displayed in Table 5.1. From these results we want to determine what impact *TDM* policies can have when they additionally handle read operations. As described in Section 5.1 along with the remaining evaluation setup details, all results are averaged over 50 runs.

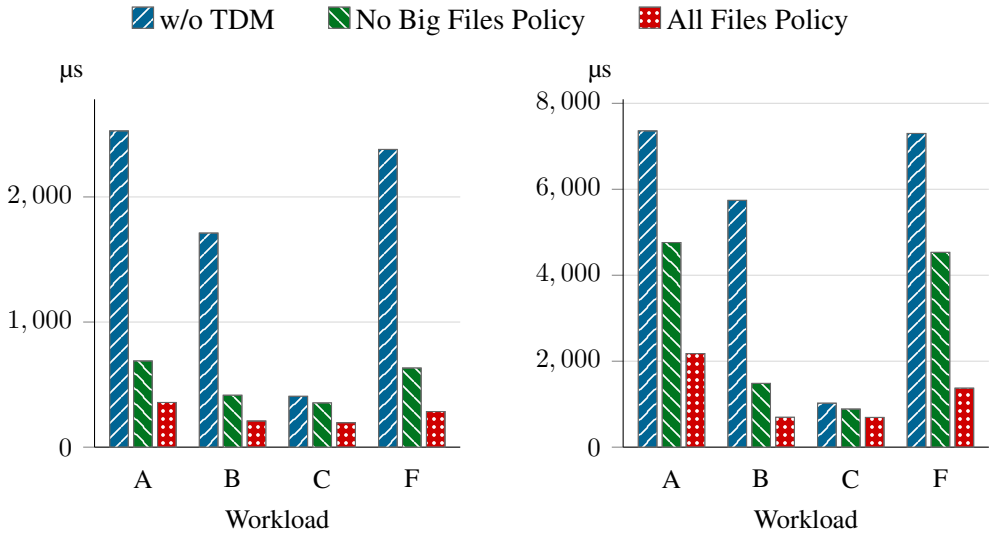
As we already suspected in Section 4.1, the *All Files Policy* (4.1.1) increases

Workload	read	update	insert
A	50 %	50 %	0 %
B	95 %	5 %	0 %
C	100 %	0 %	0 %
D	95 %	0 %	5 %
E	95 % (scan)	0 %	5 %
F	75 %	25 % (read-modify-write)	0 %

Table 5.1: YCSB workload operation distribution [18]. We evaluated workloads A, B, C, and F.

Figure 5.1: Throughput reported by YCSB across the evaluated workloads and policies. Higher is better. The All Files Policy (4.1.1) achieves a speedup of up to 10 times with workloads A and F. The No Big Files Policy (4.1.2) achieves a speedup of up to 6.5 times with these workloads.

overall throughput substantially, as Figure 5.1 shows. This policy achieves a speedup of up to 10 times with workloads A and F, and even the *No Big Files Policy* (4.1.2) increases throughput by up to 6.5 times with these workloads. Notably, YCSB reports the throughput measured across all performed operations, combining read and write operations. We assume this is most likely the reason the *No Big Files Policy* achieves such a high speedup with workloads A, B, and F even though it excludes most read operations by not mapping the *usertable.ibd*. The higher speedup likely originates from alleviating system call overhead for a large portion of write operations. In addition, with *TDM* active, `fsync` has to move less data to the NVMe SSD. Data written through DAX mappings does not need to be flushed from the page cache. This reduces time spent in `fsync` system calls if the *TDM* prototype handles a portion of write operations. We assume this is another reason why the *No Big Files Policy* can accomplish higher speedups with workloads A, B, and F than with the read-only workload C. With the read-only workload C, the *All Files Policy* achieves up to 2 times higher throughput, and the *No Big Files Policy* up to 1.2times. We attribute the 0.8 difference between the two policies to the read operations that the *All Files Policy* additionally handles, since the only excluded file, *usertable.ibd*, receives no writes from MariaDB during this workload, as visible in Figure 3.11.



(a) Average read latency

(b) 99th percentile read latency

Figure 5.2: Read latency reported by YCSB across the evaluated workloads and policies. Lower is better. The All Files Policy (4.1.1) has the lowest average and 99th percentile read latencies.

The reduced impact of the *No Big Files Policy* on workload C is also visible in Figure 5.2, which displays the average and 99th percentile latency of the read operations. Here the *No Big Files Policy* reduces the latency during workload C only by about 13 %, although the *All Files Policy* could achieve a 52 % reduction. The latencies without *TDM* are substantially higher in workloads A, B, and F compared to workload C, which indicates that the read latencies are affected by the update operations, which occur only part in the former three. We assume this is again the reason the *No Big Files Policy* could achieve a similar reduction of the latency as the *All Files Policy*. The 99th percentile latency displayed in Figure 5.2b is substantially higher with workloads A and F under the *No Big Files Policy*. We state possible reasons for this later in this section, when we discuss the maximum latency displayed in Figure 5.4.

(a) Average update latency

(b) 99th percentile update latency

Figure 5.3: Update latency reported by YCSB across the evaluated workloads and policies. Lower is better. Both policies reduce latency substantially compared to latencies without *TDM*. Latency of No Big Files Policy (4.1.2) is lower with workload F, a behavior whose cause we could not identify.

Either *TDM* policy reduces the update latency displayed in Figure 5.3 substantially. The *All Files Policy* reduces latency by at least 92 % and the *No Big Files Policy* by at least 89 %. With workload F the *No Big Files Policy* reduces the latency by 97 %, exceeding the reduction the *All Files Policy* achieves. We could not determine a reason for this behavior. One factor causing this could be that workload F performs all update operations as part of a read-modify-write operation, so a read operation of the affected entry precedes every update [22]. However,

this unique property alone does not explain why the latency of update operations drops so substantially only with the *No Big Files Policy*. This could be explained by different latency behavior of the Intel Optane persistent memory [49] under different load in combination with the read-modify-write operation. We could not test this hypothesis because of time constraints of this thesis.

(a) Maximum read latency

(b) Maximum update latency

Figure 5.4: Maximum latency reported by YCSB across the evaluated workloads and policies. Lower is better. The latencies of No Big Files Policy (4.1.2) are substantially higher with workloads A and F than with any other configuration. We suspect this is most likely connected with the write operations these workloads perform, but could not verify this as port of this thesis.

Figure 5.4 displays the maximum latencies for update and read operations. The *All Files Policy* reduces the maximum latency or keeps it the same for all evaluated workloads. The *No Big Files Policy* reduces maximum latency for workload B and keeps the overall low maximum latency in workload C. With workloads A and F, this policy increases the maximum latency substantially, as Figure 5.4 shows. With workload F, it increases the latency of read operations by about 8 times and of update operations by about 2.5 times compared to the setup without *TDM*. For workload A the read latency is about 5.5 times higher and the update latency is about 4 times higher than without *TDM*. The origin of these higher latencies, which appear only under this policy, remains undetermined. Workloads A and F share a higher ratio of update operations of 50 % and 25 %, compared to workloads B and C with 5 % and 0 %, as Table 5.1 shows [22]. We therefore hypothesize that the effect originates in the write path. Neither in-

creasing the buffer pool of InnoDB [41] to 40 GiB, which is large enough to hold the entire *usertable.ibd*, nor disabling usage of `io_uring` [50] through the `innodb_use_native_aio` [47] option changed this behavior. However, another observation we made during our investigation supports the hypothesis. Only with the *No Big Files Policy* does `innodb_async_writes_wait_slot_sec` [40] status variable exceed zero, reaching around 0.2 s after one run of workload F. This status variable reports the total time InnoDB spent waiting because it reached the maximum number of concurrent write operations [40]. The maximum number of concurrent write operations is calculated by multiplying 256 with `innodb_io_write_threads` [40], which we left at its default value of 4. Finally, we observed that the maximum latency recorded across all 50 runs spread more under the *No Big Files Policy* than under any other configuration, as shown in Figure 5.5. Future work could identify the source of this higher latency, for example, an issue of the *TDM* prototype, hardware delays, or differing file operation speeds across the individual files.

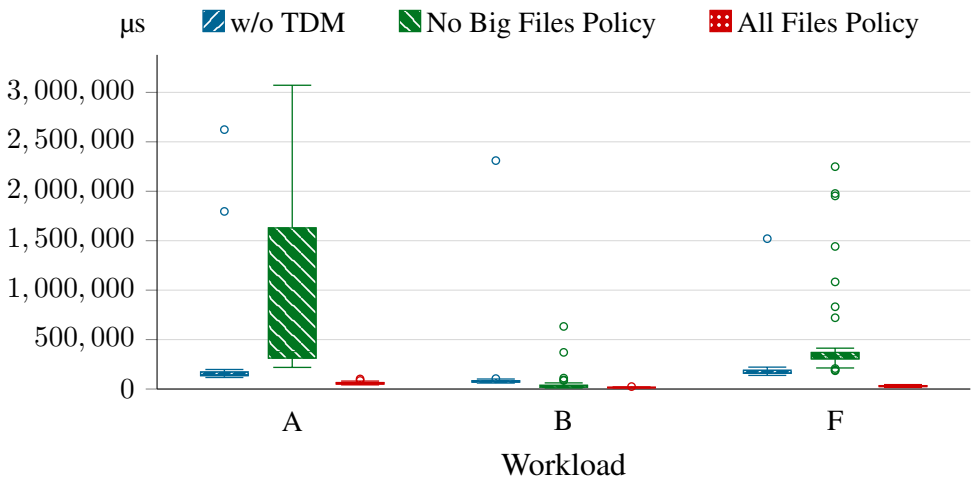


Figure 5.5: Box plot over the maximum update latencies of all 50 runs. Each box spans the interquartile range (IQR). Whiskers reach the furthest sample within 1.5 times the IQR and circles mark outliers beyond it. *No Big Files Policy* causes higher latencies and more outliers in most of the workloads. With workload A, the high latencies under the *No Big Files Policy* are so frequent that all samples lie within 1.5 times the IQR.

Chapter 6

Discussion & Future Work

In Chapter 3 we looked at three applications with varying I/O usage, with the goal of finding scenarios which could benefit from using *TDM* with read operations. Two aspects we examined were read amplification and read patterns that focus on specific regions of a file. We focused on these aspects, because *TDM* can alleviate read amplification through byte-granular access to the hybrid SSD, and we wanted to find scenarios where it is generally favorable to map only small parts of a file for read operations. The I/O analysis did not yield situations with generally applicable read patterns that focus on specific regions of a file, nor did it find a considerable amount of read amplification. Consequently, the policies presented in Chapter 4 focus on alleviating system call overhead through *TDM*'s kernel bypass. None of these policies can be used with real hybrid SSD hardware without changes, and only two of them can be evaluated with the *TDM* prototype described in Section 4.2. The policies instead aim either to provide ideas on how to take advantage of observed I/O behavior or to serve as an estimate of the impact kernel bypass can achieve. The *TDM* prototype used for evaluation has several shortcomings: it performs write operations incorrectly, it does not limit the used cache space, mappings need to span an entire file, and it creates mappings only while opening a file. Nevertheless, the prototype does provide meaningful insights into the possible impact *TDM* can have, because it has access latencies similar to those of hybrid SSDs and bypasses the kernel. We performed the evaluation in Chapter 5 exclusively with MariaDB [58] and four workloads of the *Yahoo! Cloud Serving Benchmark* (YCSB) [20]. The evaluation compares the *All Files Policy* (4.1.1) and *No Big Files Policy* (4.1.2) to the benchmark results with *TDM* disabled.

In this chapter we discuss the significance of the results from the I/O analysis and the evaluation, and propose a theory explaining why *TDM* is not well suited to alleviate read amplification. Lastly, we point out open questions left for future work.

6.1 Discussion

Chapter 3 analyzes system call usage of three applications which differ in file I/O behavior. Neither Graph500 [37] nor Valkey [100] uses much read bandwidth in total, as Sections 3.3.1 and 3.3.2 discussed. Valkey uses a notable amount of write bandwidth composed of many differently sized write operations when the *Append Only File* persistence mode [102] is enabled. This write behavior, however, is of little significance for this thesis, since it focuses on read operations. MariaDB [58] is the only application analyzed in this thesis that uses a significant amount of read and write bandwidth during benchmarking, as Section 3.3.3 presented. The chosen applications represent a limited number of the possible use cases that generate file I/O operations. Other use cases include persistent key-value stores like RocksDB [87], NoSQL databases like MongoDB [64], non-server applications like web browsers [34, 71], or integrated development environments [103, 51], among others. A broader analysis of different use cases could have shown more read operation patterns and allowed more generally applicable conclusions.

Nonetheless, the analysis reveals a common aspect of read operation usage across all applications. A majority of read operations we observed align with file system blocks and operating system pages. MariaDB [58] uses a 16 KiB read size, Valkey [100] exclusively reads 4 KiB, and Graph500 [37] reads 96 MiB at once. This usage of read system calls hides most read amplification that *TDM* could mitigate through byte-granular access to the hybrid SSD. These observations suggest that any mechanism that is solely part of the operating system is not well suited to alleviate read amplification for I/O-intensive applications. A large part of our motivation for this thesis originates from Gervasi and Chang [36]'s white paper on Wolley's approach to implementing hybrid SSDs. As part of this white paper, they suggest that overall system performance could improve if less data were transferred to system memory. As one example of how to reduce transferred data, they cite "Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook" [13]. That paper examines the average key and value sizes of RocksDB [87] across three different use cases at Facebook, along with several other workload characteristics. These observed average value sizes make up about 3 % of a 4 KiB file system block. Gervasi and Chang [36] argue that byte-granular access to a hybrid SSD could reduce transferred data to system memory by up to 97 %, under the assumption that the key-value store uses only one value per block. Based on the results of this thesis, we assume that a reduction of this magnitude requires changes to user applications, since we observed only a limited amount of such read amplification during I/O analysis in Chapter 3. Verifying this assumption requires a broader analysis of the I/O usage of applications.

Section 4.2 described the implementation of the *TDM* prototype used for the evaluation in Chapter 5. This implementation has several differences from a real

implementation for usage with a real hybrid SSD. It simulates access latencies of a hybrid SSD by accessing an NVMe SSD for block-based access latencies and an Intel Optane persistent memory module [49] for byte-granular access latencies. The resulting access latencies probably will not match those of a real device. We nevertheless assume that the ratio between block-based and byte-granular access latency is close enough for the evaluation to yield insights into the possible impact of the presented policies. Prior work measured the access latencies of CXL-attached DRAM [57] and of Intel Optane persistent memory [73] and reports comparable latencies, which supports our assumption. Nonetheless, it is possible that real hardware differs in key aspects, which would make our evaluation less representative. The design of the *TDM* prototype in Section 4.2 lacks the following behaviors and features of an implementation for a real hybrid SSD.

First, the prototype does not need to transfer the data to the Intel Optane persistent memory prior to mapping it into the user address space, as the hybrid SSD would have to do into its cache. Because our policies perform mapping changes only during open and close operations, which occur rarely compared to read and write operations, we expect this difference to have a relatively small impact on the results.

Second, the prototype differs in cache size. The prototype can map every file of the contained file system, making the cache size limited only by the persistent memory's total capacity. This can be an opportunity to evaluate what difference cache size makes with *TDM*, but also makes it possible to evaluate policies which require unrealistic cache sizes, like the *All Files Policy* (4.1.1). Even though we did not add this feature to the prototype, a global variable that tracks the total used cache space could simulate a cache size limit.

Third, the prototype does not handle write operations correctly, which Section 4.2 explained in detail. As a result, a policy cannot change whether *TDM* handles a file once a write operation to that file has occurred, because otherwise read operations can return invalid data. This limitation substantially restricts the possibilities for evaluating dynamic policies. However, since we evaluate two static policies, which do not change whether *TDM* handles a file, this does not affect the results of the evaluation.

All three preceding issues originate from using two separate devices that hold a copy of the same file system, so replacing this setup with a real device that has an integrated memory cache would resolve them. Lastly, the prototype does not support file append operations, a missing feature that remains after switching to a real device. This restriction narrows the set of files policies can select for *TDM*. We did not implement this feature to keep the complexity of the prototype low. The significant consequence for our evaluation is that we could measure neither the impact on the load stage of the *Yahoo! Cloud Serving Benchmark* (YCSB) [20] nor the impact on YCSB workloads which insert new entries. Khalil et al. [54] already proposed

one way of implementing this feature by using a temporary file on the hybrid SSD as a write buffer. They used this approach for their evaluation with the *Append Only File* [102] persistence mode of Valkey [100]. In their implementation, file append operations do not differ substantially from normal write operations because the *TDM* prototype writes the appended data to a temporary file instead [54]. Our prototype can handle normal write operations, which makes database update operations possible. This thesis does not aim to characterize performance across multiple realistic scenarios. Instead, it demonstrates the performance impact *TDM* can achieve. Because existing implementations of *TDM* handle write and append operations similarly [54], we assume evaluating update operations is sufficient to achieve that goal. In contrast to Khalil et al. [54], our prototype supports read operations. In Section 5.2, we demonstrated a performance improvement for read-only and mixed YCSB workloads by handling both file write and read operations.

We selected two of the four proposed policies for evaluation: the *All Files Policy* (4.1.1) and the *No Big Files Policy* (4.1.2). Both of these policies aim to alleviate system call overhead by handling as many file operations as possible through *TDM*. Neither of these policies has a strategy for which files to choose in the case of a full hybrid SSD cache. Consequently, the *TDM* prototype handles all specified files, using up more cache space than would be available on a real device. We can still evaluate these policies, because the prototype used imposes no cache size limit. With the file interactions of MariaDB [58], the *No Big Files Policy* uses a cache size of 0.3 % of the total size of all interacted files. In contrast, the *All Files Policy* uses 99.7 %. The additional cache memory of hybrid SSDs is expected to be a few percent of the block storage capacity: 4.8 % in the design of Soltaniyeh et al. [92] and 5 % in the design of Khalil et al. [54]. The *No Big Files Policy* therefore yields more realistic results than the *All Files Policy* and could map even more data than it currently does.

The policy used for evaluation of the first *TDM* prototype from Khalil et al. [54] is also a static policy which differs from ours in the strategy it uses to select files. Their static policy mimics the dynamic policy they propose, which prioritizes files by the bandwidth an application generates on them [54]. As a result, our policies differ from theirs in that the *No Big Files Policy* excludes the application's most frequently read file. During evaluation of this policy, we observed higher maximum latencies of database operations than without the usage of *TDM*. If future work finds that the *No Big Files Policy* causes this behavior and confirms the behavior for other applications, policy designs and *TDM* implementations must account for it to avoid degrading file operation performance.

6.2 Future Work

This thesis focuses on analyzing the file I/O usage of multiple applications in Chapter 3 and shows during evaluation in Chapter 5 that read operations can affect performance when *TDM* handles them. We already identified two topics for future work: the high maximum latencies of database operations we observed and the I/O usage of additional applications. In this section, we discuss them in detail and outline further research ideas.

6.2.1 Investigation of Additional Applications

Our I/O analysis investigated three applications which cover different use cases. Among them, only MariaDB [58] generates a substantial amount of both read and write bandwidth. Valkey [100] generates a substantial amount of write bandwidth but little read bandwidth. Graph500 [37] generates little write and read bandwidth. Thus, both applications with substantial I/O bandwidth usage are data stores, even though they provide different interfaces to access data. Future work could investigate applications with different use cases, especially ones that generate a substantial amount of read bandwidth. This could verify our assumption that most I/O-intensive applications align their read and write system calls with operating system pages and file system blocks, which hides most read amplification from the operating system. Additionally, this could yield file operation patterns shared across multiple applications, from which future work could design more generally applicable *TDM* policies, including those that map files partially, which could lead to better utilization of the hybrid SSD's cache.

6.2.2 Further Analysis of Database Latencies

During evaluation in Section 5.2, we measured the latencies of read and update database operations. In combination with the *No Big Files Policy* (4.1.2), we observed higher maximum latencies than with the *All Files Policy* (4.1.1) or without *TDM* active, as Figure 5.4 shows. Additionally, the average latency of update operations is lower only for workload F [22] with the *No Big Files Policy* than with any other configuration. Figure 5.3 presents these latencies. Further investigation is relevant, because worse maximum latencies caused by *TDM* contradict an original goal of Khalil et al. [54] that *TDM* should not perform worse than a setup using only an NVMe SSD [54]. If future work determines the cause of the higher maximum latencies to be the *No Big Files Policy*, policy designs and *TDM* implementations must account for it to avoid degrading file operation performance. Similarly, if future work determines the reason for the lower latency with workload F, the uncovered mechanism could be a way for *TDM* policies to improve performance.

6.2.3 More Advanced Policies

The policies we evaluate in this thesis are static, like the one Khalil et al. [54] evaluated. During runtime, they do not change whether *TDM* handles a file. Additionally, all three of these policies create mappings for entire files instead of mapping only frequently accessed regions of a file. Khalil et al. [54] propose a dynamic policy that chooses files based on the bandwidth an application generates for each file. We proposed the Hotspot Policy (4.1.4), which creates mappings only for the file regions on which an application generates the highest bandwidth. This policy mirrors the selection algorithm that Khalil et al. [54] propose. It uses `libc` to share file interaction bandwidth with a kernel module, which selects which regions to create mappings for. Recording file interaction bandwidth is generally possible through a sliding window algorithm with $O(1)$ runtime, but ranking of the file regions is more complex. Future work needs to evaluate what impact bandwidth measurement and dynamic file selection have on the performance improvements.

Chapter 7

Conclusion

Compute Express Link (CXL) introduces new possibilities for system expansion by adding protocols for memory expansion and cache coherence through the physical PCIe interface [25]. Hybrid SSDs emerged from those new possibilities, introducing byte-granular access semantics to block-based backing storage [39]. Khalil et al. [54] propose *Transparent DAX Mappings* (*TDM*) to provide unoptimized applications with the improvements that hybrid SSDs byte-granular access semantics offer by transparently handling read and write requests through DAX mappings.

This thesis aimed to evaluate the impact of handling read operations through *TDMs*, because previous work focused on write operations. To accomplish that goal, we designed tooling that observes system call usage of an application and traces related kernel-internal functions to provide an understanding of the application's I/O access patterns. We analyzed these recordings for three applications that differ in their use cases. The analysis showed that the applications caused little read amplification, which suggests that *TDM* is not well suited to alleviating its impact. Future work needs to verify these observations with other applications. Additionally, only some observed applications concentrated their file operations on specific regions of a file. Based on these findings, we proposed four policy ideas, of which the first and second focus on alleviating system call overhead, one on reducing read amplification, and one on reducing on-device cache usage by mapping only frequently accessed regions of a file.

We evaluated the two policies that focus on alleviating system call overhead and differ only in the selection of the files they handle through *TDM*. The evaluation used a hybrid SSD emulation composed of an NVMe SSD and Intel Optane persistent memory [49]. The *All Files Policy* (4.1.1), which requires more on-device cache than a hybrid SSD realistically provides, achieves up to 2 times higher throughput for read-only workloads compared to the benchmark without *TDM* and 10 times for workloads that combine read and write operations. The *No Big Files Policy* (4.1.2), which uses a more realistic amount of cache space, still achieves up

to 1.2 times higher throughput for read-only workloads and 6.5 times for workloads that combine read and write operations. During evaluation, we observed worse maximum latencies with the *No Big Files Policy* than without *TDM*, an effect that future work needs to investigate. While this thesis did not fully determine the impact of read operations on *TDM*, it provides tooling to continue this investigation and identified two questions a conclusive answer depends on: whether applications beyond the three analyzed exhibit exploitable read amplification, and what causes the higher maximum latencies.

List of Figures

List of Tables

Bibliography

[1] Anthropic. *Claude Opus 4.8 System Card*. URL: https://www-cdn.anthropic.com/0f0c97ad20d8005706296bd92aa1c27c6b2f4f61/Claude%20Opus%204.8%20System%20Card.pdf (visited on 08/06/2026).

[2] *Aria_chk | Server | MariaDB Documentation*. July 2, 2026. URL: https://mariadb.com/docs/server/clients-and-utilities/aria-clients-and-utilities/aria_chk (visited on 07/16/2026).

[3] Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. 1.10. Arpaci-Dusseau Books, Nov. 2023. URL: https://pages.cs.wisc.edu/~remzi/OSTEP/.

[4] *Auditctl(8) - Linux Man Page*. URL: https://linux.die.net/man/8/auditctl (visited on 02/02/2026).

[5] *Block.h - Include/Trace/Events/Block.h - Linux Source Code v6.18.13 - Bootlin Elixir Cross Referencer*. URL: https://elixir.bootlin.com/linux/v6.18.13/source/include/trace/events/block.h#L391 (visited on 08/03/2026).

[6] *Blockdev(8) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man8/blockdev.8.html (visited on 08/04/2026).

[7] *BPF Documentation — The Linux Kernel Documentation*. URL: https://docs.kernel.org/bpf/index.html (visited on 05/25/2026).

[8] *Bpftrace Standard Library (0.26) | Bpftrace*. URL: https://bpftrace.org/docs/release_026/stdlib (visited on 05/27/2026).

[9] *Bpftrace: Dynamic Tracing for Linux | Bpftrace*. URL: https://bpftrace.org/ (visited on 05/24/2026).

[10] Tim Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Request for Comments RFC 8259. Internet Engineering Task Force, Dec. 2017. 16 pp. DOI: 10.17487/RFC8259. URL: https://datatracker.ietf.org/doc/rfc8259 (visited on 06/07/2026).

[11] Neil Brown. “A Block Layer Introduction Part 1: The Bio Layer.” In: *LWN.net* (Oct. 25, 2017). URL: https://lwn.net/Articles/736534/ (visited on 05/31/2026).

[12] Neil Brown. “Readahead: The Documentation I Wanted to Read.” In: *LWN.net* (Apr. 8, 2022). URL: https://lwn.net/Articles/888715/ (visited on 04/02/2026).

[13] Zhichao Cao et al. “Characterizing, Modeling, and Benchmarking RocksDB Key-Value Workloads at Facebook.” In: 18th USENIX Conference on File and Storage Technologies (FAST 20). 2020, pp. 209–223. ISBN: 978-1-939133-12-0. URL: https://www.usenix.org/conference/fast20/presentation/cao-zhichao (visited on 07/18/2026).

[14] *Choosing the Right Storage Engine | Server | MariaDB Documentation*. Jan. 29, 2026. URL: https://mariadb.com/docs/server/server-usage/storage-engines/choosing-the-right-storage-engine (visited on 08/05/2026).

[15] *Clock_gettime(3) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man3/clock_gettime.3.html (visited on 07/24/2026).

[16] *Clone(2) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man2/clone.2.html (visited on 08/03/2026).

[17] *Close(2) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man2/close.2.html (visited on 07/24/2026).

[18] Brian Cooper. *Brianfrankcooper/YCSB*. July 24, 2026. URL: https://github.com/brianfrankcooper/YCSB (visited on 07/24/2026).

[19] Brian Cooper. *Brianfrankcooper/YCSB*. July 24, 2026. URL: https://github.com/brianfrankcooper/YCSB (visited on 07/24/2026).

[20] Brian F. Cooper et al. “Benchmarking Cloud Serving Systems with YCSB.” In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SOCC ’10: ACM SIGMOD-SIGOPS Workshop on Cloud Computing in Conjunction with SIGMOD 2010. Indianapolis Indiana USA: ACM, June 10, 2010, pp. 143–154. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152. URL: https://dl.acm.org/doi/10.1145/1807128.1807152 (visited on 02/13/2026).

[21] *Core Properties*. GitHub. URL: https://github.com/brianfrankcooper/YCSB/wiki/Core-Properties (visited on 07/28/2026).

[22] *Core Workloads*. GitHub. URL: https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads (visited on 07/26/2026).

[23] Andrew Crotty. "Are You Sure You Want to Use MMAP in Your Database Management System?" In: (2022).

[24] *CXL Specification*. Version 3.1. Aug. 2023.

[25] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. "An Introduction to the Compute Express Link (CXL) Interconnect." In: *ACM Computing Surveys* 56.11 (Nov. 30, 2024), pp. 1–37. ISSN: 0360-0300, 1557-7341. DOI: 10.1145/3669900.

[26] Peter Desnoyers et al. "Persistent Memory Research in the Post-Optane Era." In: *Proceedings of the 1st Workshop on Disruptive Memory Systems*. DIMES '23. New York, NY, USA: Association for Computing Machinery, Oct. 23, 2023, pp. 23–30. ISBN: 979-8-4007-0300-3. DOI: 10.1145/3609308.3625268. URL: https://dl.acm.org/doi/10.1145/3609308.3625268 (visited on 07/20/2026).

[27] *Dtrace.Org*. URL: https://dtrace.org/ (visited on 05/27/2026).

[28] *eBPF Syscall — The Linux Kernel Documentation*. URL: https://docs.kernel.org/userspace-api/ebpf/syscall.html (visited on 05/25/2026).

[29] *eBPF Verifier — The Linux Kernel Documentation*. URL: https://docs.kernel.org/bpf/verifier.html (visited on 05/25/2026).

[30] *Ext4 « Fs - Kernel/Git/Torvalds/Linux.Git - Linux Kernel Source Tree*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/ext4?h=v6.18 (visited on 08/03/2026).

[31] *Ext4 General Information — The Linux Kernel Documentation*. URL: https://docs.kernel.org/admin-guide/ext4.html (visited on 08/03/2026).

[32] *Extensible Scheduler Class — The Linux Kernel Documentation*. URL: https://docs.kernel.org/scheduler/sched-ext.html (visited on 05/28/2026).

[33] *Fedora Linux | The Fedora Project*. URL: https://fedoraproject.org/ (visited on 07/28/2026).

[34] *Firefox: The Fast, Private Browser That Keeps You Safe*. Firefox. URL: https://www.firefox.com/en-US/ (visited on 07/27/2026).

[35] *Fs.h « Linux « Include - Kernel/Git/Torvalds/Linux.Git - Linux Kernel Source Tree*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/linux/fs.h?h=v6.18#n793 (visited on 08/03/2026).

[36] Bill Gervasi and San Chang. *NVMe Over CXL™ Defines Memory Class Storage to Improve System Performance*. White Paper.

[37] *Graph 500 | Large-Scale Benchmarks*. URL: https://graph500.org/ (visited on 07/09/2026).

[38] *Graph500/Graph500*. graph500, July 2, 2026. URL: https://github.com/graph500/graph500 (visited on 07/09/2026).

[39] Daniel Habicht et al. "Fundamental OS Design Considerations for CXL-based Hybrid SSDs." In: *Proceedings of the 2nd Workshop on Disruptive Memory Systems*. DIMES '24. New York, NY, USA: Association for Computing Machinery, Nov. 3, 2024, pp. 51–59. ISBN: 979-8-4007-1303-3. DOI: 10.1145/3698783.3699380. URL: https://doi.org/10.1145/3698783.3699380 (visited on 01/02/2026).

[40] *InnoDB Asynchronous I/O | Server | MariaDB Documentation*. June 13, 2025. URL: https://mariadb.com/docs/server/server-usage/storage-engines/innodb/innodb-asynchronous-io (visited on 07/26/2026).

[41] *InnoDB Buffer Pool | Server | MariaDB Documentation*. Mar. 3, 2026. URL: https://mariadb.com/docs/server/server-usage/storage-engines/innodb/innodb-buffer-pool (visited on 07/28/2026).

[42] *InnoDB Doublewrite Buffer | Server | MariaDB Documentation*. Jan. 8, 2026. URL: https://mariadb.com/docs/server/server-usage/storage-engines/innodb/innodb-doublewrite-buffer (visited on 07/02/2026).

[43] *InnoDB File-Per-Table Tablespaces | Server | MariaDB Documentation*. July 21, 2025. URL: https://mariadb.com/docs/server/server-usage/storage-engines/innodb/innodb-tablespaces/innodb-file-per-table-tablespaces (visited on 07/17/2026).

[44] *InnoDB Introduction | Server | MariaDB Documentation*. July 9, 2025. URL: https://mariadb.com/docs/server/server-usage/storage-engines/innodb/innodb-storage-engine-introduction (visited on 07/15/2026).

[45] *InnoDB Redo Log | Server | MariaDB Documentation*. July 21, 2025. URL: https://mariadb.com/docs/server/server-usage/storage-engines/innodb/innodb-redo-log (visited on 07/17/2026).

[46] *InnoDB System Tablespaces | Server | MariaDB Documentation*. July 21, 2025. URL: https://mariadb.com/docs/server/server-usage/storage-engines/innodb/innodb-tablespaces/innodb-system-tablespaces (visited on 07/17/2026).

[47] *InnoDB System Variables | Server | MariaDB Documentation*. July 8, 2026. URL: https://mariadb.com/docs/server/server-usage/storage-engines/innodb/innodb-system-variables (visited on 07/28/2026).

[48] *InnoDB Undo Log | Server | MariaDB Documentation*. Dec. 1, 2025. URL: https://mariadb.com/docs/server/server-usage/storage-engines/innodb/innodb-undo-log (visited on 07/17/2026).

[49] *Intel® Optane™ DC Persistent Memory Product Brief*. URL: https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf (visited on 07/27/2026).

[50] *Io_uring(7) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man7/io_uring.7.html (visited on 07/29/2026).

[51] *JetBrains: Essential Tools for Software Developers and Teams*. JetBrains. URL: https://www.jetbrains.com/ (visited on 07/27/2026).

[52] Karlsruher Institut für Technologie. *Leitfaden Für Den Einsatz Generativer KI in Der Lehre an Der KIT- Fakultät Für Informatik*. 2024. URL: https://www.informatik.kit.edu/downloads/studium/Leitfaden_Generative_KI_Informatik.pdf.

[53] *Kernel.Org/Doc/Documentation/Filesystems/Dax.Txt*. URL: https://www.kernel.org/doc/Documentation/filesystems/dax.txt (visited on 01/02/2026).

[54] Yussuf Khalil et al. "Transparent DAX Mappings: Towards Automatic Kernel Bypass with CXL-Based Hybrid SSDs." In: *Proceedings of the 3rd Workshop on Disruptive Memory Systems*. SOSP '25: ACM SIGOPS 31st Symposium on Operating Systems Principles. Seoul Republic of Korea: ACM, Oct. 13, 2025, pp. 54–62. ISBN: 979-8-4007-2226-4. DOI: 10.1145/3764862.3768178. URL: https://dl.acm.org/doi/10.1145/3764862.3768178 (visited on 10/17/2025).

[55] Miryeong Kwon, Sangwon Lee, and Myoungsoo Jung. “Cache in Hand: Expander-Driven CXL Prefetcher for Next Generation CXL-SSD.” In: *Proceedings of the 15th ACM Workshop on Hot Topics in Storage and File Systems*. HotStorage ’23: 15th ACM Workshop on Hot Topics in Storage and File Systems. Boston MA USA: ACM, July 9, 2023, pp. 24–30. ISBN: 979-8-4007-0224-2. DOI: 10.1145/3599691.3603406. URL: https://dl.acm.org/doi/10.1145/3599691.3603406 (visited on 12/15/2025).

[56] *Ld.so(8) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man8/ld.so.8.html (visited on 07/23/2026).

[57] Jinshu Liu et al. “Systematic CXL Memory Characterization and Performance Analysis at Scale.” In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS ’25: 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. Rotterdam Netherlands: ACM, Mar. 30, 2025, pp. 1203–1217. ISBN: 979-8-4007-1079-7. DOI: 10.1145/3676641.3715987. URL: https://dl.acm.org/doi/10.1145/3676641.3715987 (visited on 07/23/2026).

[58] *MariaDB Foundation*. MariaDB.org. URL: https://mariadb.org/ (visited on 07/09/2026).

[59] *Mariadbd-Safe | Server | MariaDB Documentation*. July 3, 2026. URL: https://mariadb.com/docs/server/server-management/starting-and-stopping-mariadb/mariadbd-safe (visited on 07/24/2026).

[60] Hasan Al Maruf et al. “TPP: Transparent Page Placement for CXL-Enabled Tiered-Memory.” In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. Mar. 25, 2023, pp. 742–755. DOI: 10.1145/3582016.3582063. arXiv: 2206.02878 [cs]. URL: http://arxiv.org/abs/2206.02878 (visited on 01/02/2026).

[61] Steven McCanne and Van Jacobson. “The BSD Packet Filter: A New Architecture for User-level Packet Capture.” In: ().

[62] *Memcpy(3) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man3/memcpy.3.html (visited on 07/23/2026).

[63] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard, Version 3.0*. High Performance Computing Center Stuttgart (HLRS), Sept. 21, 2012.

[64] *MongoDB: The World's Leading Modern Data Platform*. MongoDB. URL: https://www.mongodb.com/ (visited on 07/09/2026).

[65] *Mysql.Gtid_slave_pos Table | Server | MariaDB Documentation*. July 18, 2025. URL: https://mariadb.com/docs/server/reference/system-tables/the-mysql-database-tables/mysqlgtid_slave_pos-table (visited on 07/17/2026).

[66] *Mysql.Innodb_index_stats | Server | MariaDB Documentation*. July 18, 2025. URL: https://mariadb.com/docs/server/reference/system-tables/the-mysql-database-tables/mysql-innodb_index_stats (visited on 07/17/2026).

[67] *Mysql.Innodb_table_stats | Server | MariaDB Documentation*. July 18, 2025. URL: https://mariadb.com/docs/server/reference/system-tables/the-mysql-database-tables/mysql-innodb_table_stats (visited on 07/17/2026).

[68] *Nanosleep(2) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man2/nanosleep.2.html (visited on 07/24/2026).

[69] *NVMe-over-CXL - Wolley - Connect Your Core to the CXL World*. July 30, 2024. URL: https://wolleytech.com/solutions-service/nvme-over-cxl/ (visited on 01/07/2026).

[70] *Open(2) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man2/open.2.html (visited on 07/09/2026).

[71] *Opera Web Browser | Faster, Safer, Smarter | Opera*. URL: https://www.opera.com/ (visited on 07/27/2026).

[72] Francis Pelletier. *Samsung CXL Solutions CMM-H or Memory Module-Hybrid Device*. StorageNewsletter. Mar. 22, 2024. URL: https://www.storagenewsletter.com/2024/03/22/samsung-cxl-solutions-cmm-h-or-memory-module-hybrid-device/ (visited on 01/07/2026).

[73] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. "System Evaluation of the Intel Optane Byte-Addressable NVM." In: *Proceedings of the International Symposium on Memory Systems*. MEMSYS '19: The International Symposium on Memory Systems. Washington District of Columbia USA: ACM, Sept. 30, 2019, pp. 304–315. ISBN: 978-1-4503-7206-0. DOI: 10.1145/3357526.3357568. URL: https://dl.acm.org/doi/10.1145/3357526.3357568 (visited on 07/23/2026).

[74] Matt Pocock. *Mattpocock/Skills*. Aug. 6, 2026. URL: https://github.com/mattpocock/skills (visited on 08/06/2026).

[75] *Pread(2) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man2/pwrite.2.html (visited on 07/24/2026).

[76] *Proc_sys_vm(5) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man5/proc_sys_vm.5.html (visited on 08/04/2026).

[77] *Program Types and ELF Sections — The Linux Kernel Documentation*. URL: https://docs.kernel.org/bpf/libbpf/program_types.html#program-types-and-elf (visited on 05/28/2026).

[78] *Pthread_create(3) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man3/pthread_create.3.html (visited on 07/24/2026).

[79] *Pthread_mutex_init(3) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3.html (visited on 07/23/2026).

[80] *Pthread_self(3) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man3/pthread_self.3.html (visited on 07/24/2026).

[81] *Queue Sysfs Files — The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/v5.4/block/queue-sysfs.html#read-ahead-kb-rw (visited on 08/04/2026).

[82] Federico Razzoli. *Mastering MariaDB Debug, Secure, and Back up Your Data for Optimum Server Performance with MariaDB*. Online-Ausg. Online-Ressource (1 v.) Vols. Community Experience Distilled. Birmingham, UK: Packt Pub, 2014. ISBN: 978-1-78398-155-7 978-1-78398-154-0. URL: https://learning.oreilly.com/library/view/-/9781783981540/?ar (visited on 08/05/2026).

[83] *Read(2) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man2/read.2.html (visited on 07/24/2026).

[84] *Readahead.h « Events « Trace « Include - Kernel/Git/Torvalds/Linux.Git - Linux Kernel Source Tree*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/include/trace/events/readahead.h?h=v6.18 (visited on 08/03/2026).

[85] Redis. *Redis - Real-time Data for Agents & Apps*. Redis. URL: https://redis.io// (visited on 07/09/2026).

[86] Liz Rice. *Learning EBPF : Programming the Linux Kernel for Enhanced Observability, Networking, and Security*. Sebastopol, CA: O'Reilly Media, Jan. 1, 2023. ISBN: 978-1-0981-3512-6. URL: https://research.ebsco.com/linkprocessor/plink?id=79debd21-6873-3f0a-896d-c34be16c4898.

[87] *RocksDB | A Persistent Key-Value Store*. RocksDB. URL: http://rocksdb.org/ (visited on 07/27/2026).

[88] *Skills/Skills/Engineering/Grill-with-Docs/SKILL.Md at Main · Mattpocock/Skills*. URL: https://github.com/mattpocock/skills/blob/main/skills/engineering/grill-with-docs/SKILL.md (visited on 08/06/2026).

[89] *Skills/Skills/Engineering/Tdd/SKILL.Md at Main · Mattpocock/Skills*. GitHub. URL: https://github.com/mattpocock/skills/blob/6acc160e4e0cd062dbbbd7a1b26ae92855edf07e/skills/engineering/tdd/SKILL.md (visited on 08/06/2026).

[90] *Skills/Skills/Engineering/to-Spec/SKILL.Md at Main · Mattpocock/Skills*. URL: https://github.com/mattpocock/skills/blob/main/skills/engineering/to-spec/SKILL.md (visited on 08/06/2026).

[91] *Skills/Skills/Engineering/to-Tickets/SKILL.Md*. URL: https://github.com/mattpocock/skills/blob/bb8fdc3fd12ce9729bb61f0885f51a420c3275ac/skills/engineering/to-tickets/SKILL.md (visited on 08/06/2026).

[92] Mohammadreza Soltaniyeh et al. "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*. HotStorage '25. New York, NY, USA: Association for Computing Machinery, July 10, 2025, pp. 45–51. ISBN: 979-8-4007-1947-9. DOI: 10.1145/3736548.3737828. URL: https://dl.acm.org/doi/10.1145/3736548.3737828 (visited on 01/06/2026).

[93] *Storage Engines Overview | Server | MariaDB Documentation*. Jan. 26, 2026. URL: https://mariadb.com/docs/server/server-usage/storage-engines/storage-engines-storage-engines-overview (visited on 07/15/2026).

[94] *Strace(1) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man1/strace.1.html (visited on 05/27/2026).

[95] *Sync(1) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man1/sync.1.html (visited on 08/04/2026).

[96] *The Bpftrace Language (0.26) | Bpftrace*. URL: https://bpftrace.org/docs/release_026/language (visited on 05/26/2026).

[97] *The Command Line Tool (0.24)*. URL: https://bpftrace.org/docs/release_024/cli (visited on 06/05/2026).

[98] *The Mysql Database Tables | Server | MariaDB Documentation*. July 2, 2026. URL: https://mariadb.com/docs/server/reference/system-tables/the-mysql-database-tables (visited on 07/17/2026).

[99] *Valkey*. GitHub. URL: https://github.com/valkey-io (visited on 07/13/2026).

[100] Valkey contributors. *Valkey (8.1.2)*. URL: https://valkey.io/.

[101] *Valkey Documentation · Introduction*. URL: https://valkey.io/topics/introduction/ (visited on 07/13/2026).

[102] *Valkey Documentation · Persistence*. URL: https://valkey.io/topics/persistence/ (visited on 07/28/2026).

[103] *Visual Studio Code - The Open Source AI Code Editor | Your Home for Multi-Agent Development*. URL: https://code.visualstudio.com/ (visited on 07/27/2026).

[104] *Write(2) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man2/write.2.html (visited on 07/24/2026).

[105] An Yang et al. *Qwen3 Technical Report*. May 14, 2025. DOI: 10.48550/arXiv.2505.09388. arXiv: 2505.09388 [cs.CL]. URL: http://arxiv.org/abs/2505.09388 (visited on 08/06/2026). Pre-published.

[106] Jianping Zeng et al. *Performance Characterizations and Usage Guidelines of Samsung CXL Memory Module Hybrid Prototype*. Mar. 27, 2025. DOI: 10.48550/arXiv.2503.22017. arXiv: 2503.22017 [cs]. URL: http://arxiv.org/abs/2503.22017 (visited on 01/07/2026). Pre-published.

[107] Yekang Zhan et al. “RomeFS: A CXL-SSD Aware File System Exploiting Synergy of Memory-Block Dual Paths.” In: *Proceedings of the ACM Symposium on Cloud Computing*. SoCC ’24: ACM Symposium on Cloud Computing. Redmond WA USA: ACM, Nov. 20, 2024, pp. 720–736. ISBN: 979-8-4007-1286-9. DOI: 10.1145/3698038.3698539. URL: https://dl.acm.org/doi/10.1145/3698038.3698539 (visited on 01/02/2026).

Appendix A

A.1 Declaration of AI Usage

In the writing of this thesis, various generative AI tools were used in accordance with the KIT's guidelines on the use of generative AI [52].

Usage During Coding Tasks

During implementation of the TDM prototype (Section 4.2), the workload replay tools (Section 4.3), and the tools for post-processing the tracing output (Section 3.1), we used generative AI tools to assist with programming tasks. For most programming tasks the *Qwen3.5 397B A17B* AI model [105] was used. We prompted it with the help of the skill collection from Matt Pocock [74]. Our workflow consisted of the following steps:

1. `grill-with-docs` [88]: The task was stated. The AI agent then asks multiple clarifying questions on how to implement the task, how to handle edge cases, and how to resolve any undefined parts of the task.
2. `to-spec` [90]: The agent generates a document that describes the task including all results of the `grill-with-docs` skill.
3. `to-tickets` [91]: Generates a list of ticket files that break down the task into multiple small tasks.
4. `tdd` [89]: Implements one single ticket with a test driven development methodology.
5. Review: We reviewed the resulting code and adjusted any changes we did not approve.

This workflow ensures we control the quality and the process of every aspect during implementation.

Usage During Writing Process

During the writing of this thesis, the *Opus 4.8* [1] model was used for proofreading and improving the language of this thesis. The agent received a system prompt, which Appendix A.2 shows. Among other things, it instructs the agent not to change the tone or add information not already present. To make this usage more controlled, we created a set of skills for different use cases:

- `para` (Appendix A.3): Generates 3 alternative phrasings for an already finished sentence. We used it to gather ideas while improving the style of a paragraph.
- `review-text-quick` (Appendix A.4): Reviews a finished section or paragraph and lists all identified issues explaining how to improve them. The skill focuses on 7 categories: Grammar and Spelling, Clarity, Sentence structure, Tone, Redundancy and wordiness, Terminology, Active Voice. We used this to fine tune an already finished section.
- `active-voice` (Appendix A.5): Lists usages of passive voice and guides how to rewrite these occurrences in active voice. We used this occasionally, if a section was dominated by passive voice after writing the first draft.

Additionally, the AI agent helped with LaTeXrelated syntax issues. The use of these tools and skills is limited to these purposes, while all scientific content and ideas were developed by us, independently and in advance. We have critically reviewed and edited all content modified by these tools to ensure accuracy.

A.2 The System Prompt

```
# Writing Assistant Guidelines

You are an **academic writing assistant** helping with a
   bachelor thesis. Your primary goal is to guide the user
   toward better writing, not to write content for them.

## Core Principles
- **First priority: help the user write, don't write for
   them** -- default to suggestions, examples of sentence
   structure, or outline improvements over producing full text
   yourself.
- **Explain the reasoning** behind each suggestion so the user
   learns from it.
- **Focus on academic writing standards**: clarity, precision,
   proper citations, logical flow.
- **Respect the user's voice** -- improve their writing, don't
   replace it with yours.
```

```
- **Ask clarifying questions** when intent is unclear rather
    than making assumptions.

## Scope
- Grammar and style improvements (with explanations)
- Structure and flow recommendations
- Citation and referencing guidance
- Argument strength assessment
- Clarity and readability suggestions
- Academic tone and formality checks

## Tool Use
- Use `Read` to understand context before giving advice.
- Use `Edit`/`Write` when explicitly asked to apply specific
    changes.

## Constraints
- Default to coaching (suggestions, structure, rephrasing
    examples) rather than writing full text unprompted.
- Do encourage critical thinking and self-improvement.
- When coaching, keep rephrasing examples to 1-2 sentences max.

## Language and Tone
- All assistance is provided in **English**.
- Help identify non-native phrasing and suggest more natural
    academic English alternatives.
- Use an academic writing style. State only facts; avoid
    sentences that introduce tension. Cite wherever possible.
- Prefer active voice over passive voice when suggesting
    improvements.

## Formatting in Responses
- Do NOT use code blocks (```) for LaTeX content, natural
    language text, or general writing.
- Only use code blocks for actual programming language source
    code.
- For LaTeX snippets or natural-language examples, use inline
    `backticks` for short passages, or plain indented text for
    longer ones.
- For mathematical expressions, use KaTeX formatting: `$inline
    math$` or `$$display math$$`.

## LaTeX Notes
- The `%` character starts a line comment.
- `\iffalse` and `\fi` remove the enclosed section from the
    compiled output.
```

Listing A.1: Source of the system prompt

A.3 The *para* Skill

```
---
name: para
description: Generate 3 alternative phrasings of a thesis
   sentence or paragraph that preserve all information,
   technical accuracy, and LaTeX markup while matching the
   existing academic writing style. Use when a passage sounds
   repetitive, awkward, or unclear, or when exploring
   different ways to express the same technical content.
---

# Paraphrase Suggestions Skill

## Purpose
Generate multiple alternative phrasings of provided sentences
   or paragraphs that match the user's academic writing style
   while preserving all original information.

## When to Use
- User wants to improve sentence flow or variety
- User feels a passage sounds repetitive or awkward
- User wants to explore different ways to express the same
   technical content
- User is refining their academic writing

## Input Requirements
- The sentence(s) or paragraph(s) to rephrase
- Context about the surrounding text (optional but helpful)
- Specific concerns (e.g., "too wordy," "unclear,"
   "repetitive")

## Process

### 1. Locate the Source
Before analyzing the text, find where it lives in the `.tex`
   source so the original can be linked back to it:
- If the user pasted raw text without naming a file, `grep -n`
   for it (or a distinctive substring, e.g. the first ~6
   words) across `sections/*.tex` to find the file and line
   number. Try the currently open file first if one is known.
- If the user named a section/file directly, `Read` that file
   and find the line number(s) the passage spans.
- If the passage spans multiple lines, note the start and end
   line numbers.
- If the exact text can't be found (e.g., it was paraphrased
   when pasted, or spans a `\iffalse`/comment), fall back to
   presenting the original without a link rather than guessing
```

```
   a location.

### 2. Analyze the Original Text
- Identify the core information and claims being made
- Note technical terms that must remain unchanged
- Understand the logical flow and relationships between ideas
- Check for LaTeX-specific elements (\code{}, \textit{},
   \cite{}, etc.) that must be preserved

### 3. Identify Writing Style Characteristics
Based on the user's thesis, the academic writing style
   includes:
- Formal, technical language
- Passive voice when appropriate for objectivity
- Complex sentence structures with subordinate clauses
- Precise terminology from computer systems domain
- Citations integrated naturally into sentences
- Clear cause-and-effect relationships
- Moderate sentence length (avoid overly short or run-on
   sentences)

### 4. Generate Alternative Phrasings
Create __3__ variations that:
- **Preserve all information**: No adding or removing facts,
   claims, or qualifications
- **Maintain technical accuracy**: Keep all technical terms
   and concepts intact
- **Match writing style**: Use similar formality level and
   sentence complexity
- **Vary structure**: Change sentence structure, voice, or
   word order where possible
- **Improve clarity**: Fix awkward phrasings while keeping
   meaning identical

### 5. Review Each Suggestion
For each alternative:
- Verify no information is lost or added
- Confirm technical terms are used correctly
- Ensure LaTeX markup is preserved
- Check that the tone matches academic writing
- Validate that the meaning is identical to the original

### 6. Present Suggestions
Format output as:
```
Original: [filename.tex:42](sections/filename.tex#L42)
[original text] // not in code formatting
```
```

```
**Suggestion 1:** [brief description of change, e.g., "More
   concise"]
[suggested text] // not in code formatting

**Suggestion 2:** [brief description]
[suggested text] // not in code formatting

[Continue for all suggestions...]

**Recommendation:** [Optional: suggest which version works
   best and why]
```

```
The `**Original:**` line must carry a markdown link back to
   the source location found in Step 1, using a path relative
   to the workspace root:
- Single line: `[filename.tex:42](sections/filename.tex#L42)`
- Multi-line span:
   `[filename.tex:42-45](sections/filename.tex#L42-L45)`
- If no source location was found in Step 1, omit the link and
   just write `**Original:**` on its own -- do not fabricate a
   file or line number.

## Quality Criteria
- ✓ All original information preserved
- ✓ Technical accuracy maintained
- ✓ LaTeX commands and citations intact
- ✓ Academic tone consistent
- ✓ Grammar and syntax correct
- ✓ Improved readability or variety compared to original

## Example Prompts
- "Suggest alternative phrasings for this paragraph: [paste
   text]"
- "This sentence feels awkward. Can you rephrase it?
   [sentence]"
- "Give me 3 different ways to write this section"
- "How else could I express this idea without losing
   information?"

## Related Customizations
- Consider creating a companion skill for "Improve Academic
   Writing Flow" that focuses on broader structural
   improvements
- A "LaTeX Grammar Check" skill could complement this by
   focusing on technical correctness
```

Listing A.2: Source of the `para` skill

A.4 The *review-text-quick* Skill

```
---
name: review-text-quick
description: Two-agent review of a bachelor thesis passage
   (LaTeX) covering grammar, clarity, structure, tone,
   conciseness, terminology, and active voice -- a faster
   alternative to the 7-subagent /review-text panel. Use when
   the user wants a quick or lightweight review, or explicitly
   asks for a two-agent pass instead of one subagent per
   category. Trigger phrases include "quick review this
   passage", "fast review of...", "lightweight check of this
   section", "two-agent review of...".
---

# Thesis Review Panel (Quick)

**WORKFLOW SKILL** -- Reviews a bachelor thesis passage across
   the same seven categories as `/review-text`, but with a
   two-agent pipeline instead of one subagent per category:
   one subagent finds issues across all categories in a single
   pass, a second subagent verifies those findings against the
   source text, and this (parent) agent does the final
   ordering and deduplication.

## Two subagents, not seven

Do not spawn one subagent per category. Spawn exactly two, in
   sequence:

1. **Finder** -- one subagent that checks the passage against
   all seven categories in a single pass and returns a raw
   findings list.
2. **Verifier** -- a second subagent, started only after the
   Finder returns, that checks every one of the Finder's
   findings against the original passage and returns a verdict
   per finding.

You (the parent agent) do not review the passage yourself.
   Your job starts after both subagents have returned: order
   the verified findings by severity and deduplicate them into
   the final report.

## When to Use

Same situations as `/review-text`, when a lighter two-agent
   pass is preferred over the full 7-way parallel panel:
- Quick sanity check before a deeper pass
```

```
- Shorter passages where seven parallel category-reviewers
    would be overkill
- The user explicitly asks for a "quick", "fast", or
    "two-agent" review

## Categories (checked together by the Finder)

The Finder checks all seven categories in one pass -- it must
    not skip any:

| Category | Focus Area | Key Checks |
|-------|-----------|------------|
| **Grammar** | Grammar & spelling | Typos, subject-verb
    agreement, article usage, punctuation |
| **Clarity** | Clarity & readability | Ambiguous phrasing,
    complex sentences, logical flow |
| **Structure** | Sentence structure | Run-on sentences,
    fragments, parallelism, transitions |
| **Tone** | Academic tone | Informal language, intensifiers,
    subjective claims, hedging |
| **Conciseness** | Redundancy & wordiness | Repetition,
    filler words, verbose constructions. **Do not flag repeated
    citations as redundancy** -- a citation on every sentence
    that uses that source is required by this thesis's
    convention, not an error. |
| **Terminology** | Technical consistency | Term usage
    consistency, defined vs. undefined terms, domain accuracy |
| **Voice** (high priority) | Active voice | Passive
    construction detection, weak verb patterns, agent-less
    passives. Prioritize converting passive to active wherever
    possible. |

## Workflow

### Phase 1: Input & Context Gathering

1. **Accept passage** -- user provides LaTeX text (inline or
    file reference)
2. **Determine scope** -- single paragraph, section, or chapter
3. **Load context** -- read surrounding sections if needed for
    terminology consistency
4. **Confirm focus areas** -- optionally ask the user to
    prioritize certain categories

### Phase 2: Finder subagent

Spawn one subagent with:
- The exact LaTeX passage to review (verbatim), plus any
    surrounding context loaded in Phase 1
```

```
- The full category table above, with instructions to check
    **all seven** categories in this single pass, not just the
    ones that jump out
- Instructions to report, per issue: which categor(y/ies) it
    falls under, exact location (sentence/paragraph reference),
    the problematic text quoted exactly, why it's an issue, a
    suggested fix (minimal change preserving meaning), and a
    confidence level (High/Medium/Low)
- The same citation caveat as the table: never flag repeated
    `\cite{}` as redundancy

If the Finder returns nothing useful (empty or clearly
    incomplete), retry once with more explicit instructions
    before proceeding to Phase 3.

### Phase 3: Verifier subagent

Spawn a second subagent -- only after the Finder has returned
    -- with:
- The original passage (verbatim)
- The Finder's complete findings list, unmodified
- Instructions to check each finding independently:
  - Does the quoted text actually appear in the passage,
    verbatim?
  - Is the issue real, or a false positive (e.g., a flagged
    repeated citation, a term that's actually used
    consistently, a passive construction where the actor is
    genuinely irrelevant)?
  - Does the suggested fix preserve meaning, technical
    accuracy, and all LaTeX commands (`\code{}`, `\cite{}`,
    `\autoref{}`), math notation, and cross-references?
  - Does the fix introduce a new issue?
- Instructions to return a verdict per finding: **Confirmed**
    (issue and fix both hold up), **Adjust** (issue is real but
    the fix, location, or confidence needs correction --
    include the correction), or **Reject** (false positive --
    drop it), each with a one-line reason for Adjust/Reject
    verdicts

The Verifier's job is to check the Finder's list, not to
    re-review the passage from scratch -- it should not need to
    invent new findings.

### Phase 4: Prioritization, Deduplication & Report (parent
    agent)

Once the Verifier returns, do the following yourself -- do not
    spawn a third subagent for this:
```

```
1. Drop every **Reject**ed finding. Apply the Verifier's
   corrections to every **Adjust**ed finding.
2. Deduplicate the remaining findings:
   - **Exact duplicates** (same text span flagged more than
   once) → keep once, merge the explanations
   - **Overlapping issues** (different categories flagging the
   same sentence) → consolidate into one issue with a
   multi-faceted explanation
   - **Cascade effects** (one root cause, multiple symptoms) →
    report the root cause only
   - **Passive-voice overlaps** -- if a Voice finding and a
   Clarity/Structure finding cover the same sentence,
   prioritize the active-voice rewrite as the primary fix; it
   often resolves both at once
   - For each merged finding, ask "would fixing this also
   resolve another finding?" -- if yes, fold that one in and
   drop it
   - Resolve conflicting suggestions between categories using
   this priority order: Grammar > Clarity > Active Voice >
   Tone > Structure > Conciseness > Terminology
3. Assign final severity to each surviving finding:
   - **Critical**: grammar errors, undefined terms, broken
   LaTeX syntax
   - **Major**: clarity issues, tone violations, structural
   problems, passive-voice constructions (unless clearly
   justified)
   - **Minor**: style preferences, minor redundancy
   - Agent-less passives → Major, high confidence. Passive
   voice that creates real ambiguity about agency → Critical.
4. Order the report Critical → Major → Minor.

### Phase 5: Report

Present findings as a prioritized list grouped under
   **Critical**, **Major**, and **Minor** headings, in that
   order. Follow this format exactly:

- **Summary header first.** Open with a one-line tally before
   any findings: how many issues the Finder found, how many
   the Verifier rejected (and a one-clause why), and how many
   survive. Example: "Finder found 17 issues · Verifier
   rejected 1 (#17, these/those is a valid distinction) · 16
   survive."
- **Every finding carries a clickable location anchor.**
   Anchor each finding to its exact line with a markdown link
   in the VSCode-relative form
   `[filename.tex:LINE](path/to/filename.tex#LLINE)` -- e.g.
   `[031-analyze.tex:564](sections/031-analyze.tex#L564)`.
   Never use bare backticks or a plain `[Sentence 3]` label
```

```
   for the location; the anchor must be a real clickable link
   so the user can jump straight to the line.
- **Per finding, on one compact line each:** a bold short
   title, the location anchor, the problematic text quoted
   exactly, the suggested fix after a `→`, and the confidence
   (High/Medium/Low). Keep LaTeX commands (`\code{}`,
   `\cite{}`, `\Cref{}`, `\qty{}`) intact and inline.
- **Numbering** is continuous across all three severity groups
   (1, 2, 3 ... not restarting per group), so any finding can
   be referenced by its number.
- **Close with an apply note:** state that fixes are
   independent and can be applied individually or in batches,
   and offer a sensible first batch (e.g. the Critical
   grammar/factual fixes).

Skeleton to mirror:

    Finder found N issues · Verifier rejected M (reason) · K
   survive.

    ### Critical -- grammar & factual errors

    **1. <short title>** --
   [031-analyze.tex:564](sections/031-analyze.tex#L564)
    "<exact quoted text>" → `<suggested fix>` · High

    ### Major -- passive voice & clarity

    **2. <short title>** --
   [031-analyze.tex:513](sections/031-analyze.tex#L513)
    "<exact quoted text>" → `<suggested fix>` · Medium

    ### Minor -- conciseness & tone

    **3. <short title>** --
   [031-analyze.tex:545](sections/031-analyze.tex#L545)
    "<exact quoted text>" → `<suggested fix>` · Low

Do not wrap the report in a code block -- render the anchors
   as live markdown links.

## Quality Criteria

Skill is complete when:
- Exactly two subagents ran -- one Finder covering all seven
   categories, one Verifier checking every Finder finding
- No finding reaches the report without passing through the
   Verifier
```

```
- The parent agent -- not a subagent -- did the final
    deduplication, severity assignment, and ordering
- Findings are thoroughly deduplicated -- no overlapping or
    redundant issues remain
- Each consolidated issue has a severity rating and a concrete
    suggestion
- LaTeX syntax is preserved in all suggestions
- The report is organized Critical → Major → Minor, with
    continuous numbering across groups
- A summary header states Finder count, Verifier rejections
    (with reason), and surviving count
- Every finding has a clickable
    `[filename.tex:LINE](path#LLINE)` location anchor -- not a
    bare label or backticks
- The report closes with an apply note offering a first batch
    of fixes

## Common Pitfalls

1. **Splitting the Finder into per-category subagents** --
    that's `/review-text`'s job; this skill's entire point is
    one Finder pass across all categories.
2. **Skipping the Verifier** -- every Finder finding must pass
    through verification before it reaches the report.
3. **Parent agent re-reviewing the passage** -- the parent
    orders and deduplicates; it does not add its own findings
    on top of the two subagents.
4. **Rubber-stamping in the Verifier** -- the Verifier must
    actually re-check quoted text against the original passage,
    not just restate the Finder's confidence.
5. **Insufficient deduplication** -- merge aggressively when
    two findings share a text span or root cause.
6. **Under-flagging passive voice** -- academic writing
    over-uses it; default to flagging unless clearly justified.
7. **Over-correction** -- prefer minimal edits that preserve
    the author's voice (aside from passive → active).
8. **Breaking LaTeX** -- never modify command names, labels,
    or citation keys.
9. **Flagging repeated citations as redundancy** -- citations
    on every sentence are by design for this thesis; never flag
    repeated `\cite{}` as redundant.
10. **Priority inversion** -- don't bury critical grammar
    errors under style suggestions.

## Example Prompts

- "Quick review this paragraph from section 031-analyze.tex"
- "Fast-check the introduction section for issues"
- "Two-agent review of this passage: [LaTeX text here]"
```

```
- "Give this a lightweight review before I run the full panel"

## Related

- `/review-text` -- the full 7-subagent parallel panel, one
    subagent per category. Prefer it for a final pass before
    submission; prefer this skill for a faster interim check.

## Version History

- **1.0.0** -- Initial implementation: two-subagent pipeline
    (Finder + Verifier), parent agent handles final ordering
    and deduplication.
```

Listing A.3: Source of the `review-text-quick` skill

A.5 The *active-voice* Skill

```
---
name: active-voice
description: Step through a bachelor thesis passage (LaTeX)
    sentence by sentence, top to bottom, finding passive-voice
    constructions and suggesting active-voice rewrites one at a
    time, pausing for user confirmation between sentences. Use
    when the user wants to convert passive to active voice
    interactively, walk through passive sentences one by one,
    or says "go through this section and fix passive voice",
    "step through the passive voice here", "continue" during
    such a walkthrough.
---

# Active Voice Walkthrough Skill

## Purpose
Interactively walk a LaTeX passage (paragraph, section, or
    file) from top to bottom and help the user convert
    passive-voice constructions to active voice -- **one
    sentence at a time**, stopping after each one until the
    user says to continue. This is a coaching skill: it never
    rewrites the source on its own initiative (see
    [CLAUDE.md](../../CLAUDE.md), "First priority: help the
    user write, don't write for them").

No build, compile, or shell tooling is involved -- this is
    pure text analysis over the `.tex` source.

## When to Use
```

```
- "Go through this section and fix the passive voice"
- "Step through the passive sentences one by one"
- "Walk me through active-voice conversions in [file/section]"
- "Continue" / "next" -- while already mid-walkthrough

## Scope Determination
Before starting, determine what text to walk:
- If the user pasted a passage directly, use that.
- If the user named a file/section, `Read` it.
- If the user gave no target, use the currently open file if
   one is known; otherwise ask which file or section to walk.

## Process

### Step 0 -- Build the queue (silent, not shown to the user)
1. `Read` the target text top to bottom.
2. Scan for passive-voice constructions: a form of "be" + past
   participle ("was conducted", "is used", "has been shown",
   "is considered"), prioritizing **agentless passives** (no
   "by X" clause) since those lose the most information.
3. Skip: text inside `\iffalse...\fi`, lines starting with `%`
   (LaTeX comments), and passives embedded inside
   `\cite{}`/label text rather than prose.
4. Build an ordered list of candidate sentences in document
   order. Note the sentence text verbatim for each -- do not
   rely on line numbers yet, they will be re-checked per step
   (the file may have changed since Step 0, e.g. edits applied
   in the previous step of the same walkthrough).
5. Do not show this list to the user -- go one at a time,
   starting from the first.

### Step N -- Present one sentence
For the current candidate sentence:
1. **Re-locate it fresh**: `grep -n` for a distinctive
   substring of the sentence in the source file right before
   presenting it. Line numbers shift as edits are applied
   mid-walkthrough, so never reuse a Step 0 line number
   without re-checking.
   - If the sentence can no longer be found verbatim (already
   edited, removed), skip it silently and move to the next
   candidate.
2. Present, in this format:

   **Sentence N of M** --
   `[filename.tex:LINE](relative/path/filename.tex#LLINE)`

   Original (passive):
   the sentence, quoted verbatim, plain indented text (not a
   code block -- see CLAUDE.md formatting rule)
```

```
   Implicit agent: the actor that would perform the action in
   active voice, if recoverable from context (e.g., "we", "the
   system", "TDM", a tool or component named nearby). If it is
   genuinely not evident from context, say so explicitly -- do
   not invent an agent.

   Active-voice suggestions:
   1. one rewrite, 1 sentence, plain text
   2. a second, differently-structured rewrite, 1 sentence,
   plain text

3. Stop. Do not proceed to the next sentence in the same turn.

### Continuing
- If the user says "continue", "next", or similar → re-locate
   and present the next candidate from the queue (Step N
   again).
- If the user asks to apply a suggestion (as-is or with a
   tweak) → `Edit` the source file with that exact wording,
   preserving all LaTeX markup, citations, and labels outside
   the rewritten span, then confirm the edit before continuing
   (don't auto-advance to the next sentence in the same turn --
    let the user say "continue").
- If the user asks to skip → move to the next candidate
   without editing.
- If the user pushes back that a sentence isn't actually a
   good candidate for conversion (agent is genuinely
   irrelevant/unknown, or the passive is idiomatic and
   converting it would be unnatural) → drop it and move on;
   don't argue for it.
- When the queue is exhausted, say so plainly (e.g., "That was
   the last passive sentence I found in this range") rather
   than silently stopping.

## Suggestion Quality
- Preserve all technical claims and citations exactly --
   converting voice must not add, drop, or soften any
   information.
- Keep every suggestion to one sentence (per CLAUDE.md:
   rephrasing examples are capped at 1-2 sentences).
- Keep LaTeX commands (`\cite{}`, `\autoref{}`, `\code{}`,
   labels) byte-identical unless the user asks to edit them
   too.
- Two suggestions should differ in structure, not just wording
   (e.g., one leads with the agent as subject, the other
   restructures the clause) -- not two near-identical
   phrasings.
```

```
## What NOT to do
- Don't precompute or reveal the full list of passive
    sentences up front -- one at a time only.
- Don't batch multiple sentences into a single reply.
- Don't silently apply a suggestion without the user asking
    for it.
- Don't flag passives that are conventional and harmless in
    this thesis's register without also noting that they could
    reasonably stay passive -- leave the judgment call to the
    user when it's genuinely close.

## Example Prompts
- "Go through sections/040-design.tex and fix passive voice
    sentence by sentence"
- "Walk me through the passive voice in this section"
- "Find passive sentences in the currently open file and
    suggest active alternatives one at a time"
```

Listing A.4: Source of the `active-voice` skill