Karlsruhe Institute of Technology

Software RAID for CXL-Based SSDs

Bachelor's Thesis of

Linus Paul Dierheimer

at the
Department of Informatics
Institute of Computer Engineering

First examiner:	Prof. Dr.-Ing. Frank Bellosa
Second examiner:	Prof. Dr. Wolfgang Karl
Advisor:	Yussuf Khalil, M.Sc.

26. November 2025 – 26. March 2026

Statutory Declaration

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, 26.03.2026

Abstract

The emergence of CXL-based persistent storage, which can be accessed both asynchronously through `CXL.io` and synchronously through `CXL.mem`, has led to the development of *Transparent DAX Mappings* (TDMs). TDMs allow applications to take advantage of CXL's *Direct Access* (DAX) feature without requiring modifications. They work by transparently modifying `read` and `write` calls to use DAX mappings for I/O. However, supporting traditional storage features such as RAID in this context is challenging due to the desire to leave the kernel out of the data path. In this thesis, we present *CXLRaid*, a userspace-only implementation of RAID 1 and RAID 5 that extends the TDM concept to provide RAID functionality on a file level. It leverages files on multiple devices as storage backends, each of which is accessed through a DAX mapping. This provides the necessary redundancy for RAID while not involving the kernel in the data path at all. With this approach, we achieve up to 3.8× higher write bandwidth for RAID 5 and up to 1.4× higher read bandwidth for RAID 1 compared to the Linux kernel's md/raid implementation, while maintaining comparable performance for RAID 1 writes. We also discuss possible future work on our implementation, as well as propose a RAID design that continues to use TDMs for data transfer but moves other logic into the kernel for closer integration.

Contents

Chapter 1

Introduction

Fast storage is crucial for modern computing systems, making storage technologies a key area of research and development. One of the latest advancements in this field is *Compute Express Link* (CXL) [30], a high-speed interconnect designed for coherent memory access between various components such as CPUs, GPUs, and accelerators. CXL defines three protocols that can be used for communication: `CXL.io`, `CXL.mem`, and `CXL.cache`. `CXL.io` is the fundamental one, providing both management features like device discovery and configuration, as well as an asynchronous block interface for data transfer, similar to how SSDs are usually accessed [3]. `CXL.mem` is used for synchronous, byte-addressable memory access, similar to how volatile DDR memory is accessed [1]. Using these two protocols, vendors have developed prototypes of non-volatile memory devices that can be accessed through both interfaces. An example of these is the Samsung CMM-H memory module [138], which combines NAND flash storage, commonly used in SSDs, with a DRAM cache. Persistence for the entire content is promised via corresponding CXL commands [130].
A recent part of the research on CXL has focused on how to replace traditional storage devices with such CXL-based modules. The challenge in that is to find a way to make use of the novel features offered by CXL while still maintaining compatibility with existing software. A proposed solution by Khalil et al. are *Transparent DAX Mappings* (TDMs) [50]. They work by transparently modifying `read` [63] and `write` [64] calls to use *Direct Access* (DAX) mappings for the I/O, which in turn allows the use of the `CXL.mem` protocol for data transfer. DAX mappings directly expose the devices memory to the address space of the application, allowing it to circumvent the kernel completely for data transfer.

This approach has been shown to significantly improve bandwidth and latency in certain cases [50] when compared to unmodified data flows.
An important part of enabling a smooth transition to CXL-based storage is to support all the features that a user would expect from a traditional storage device. The feature that we focus on in this thesis is *Redundant Array of Independent Disks* (RAID) [59]. RAID is a widely used technique for improving the performance and reliability of storage systems by distributing data across multiple drives. There are predefined strategies for how to distribute the data, called *RAID levels*, which offer different trade-offs between performance, reliability, and storage efficiency. In this thesis, we focus on RAID levels 1 and 5, which are both commonly used in practice [5]. RAID 1 [59], also known as *mirroring*, duplicates a copy of the data across all drives, providing high reliability at the cost of storage efficiency. RAID 5 [59], also known as *striping with distributed parity*, distributes data together with calculated parity information across multiple drives, providing a good balance between the mentioned trade-offs.
Traditional RAID systems are either implemented in hardware or in the kernel [59, 65], posing a specific challenge for CXL-based storage. Hardware RAID implementations on top of CXL do not exist yet, and it is unclear what they would look like. Kernel-based RAID implementations, on the other hand, are not compatible with the TDM approach, as their whole design is based on circumventing the kernel for data transfers. This raises the question of how to implement RAID while still being able to use TDMs for I/O.
In this thesis, we present *CXLRaid*, an approach to implementing such a RAID system. *CXLRaid* is a userspace-only implementation of RAID 1 and RAID 5 that works on the files level. Similar to TDMs, it transparently modifies `read` and `write` calls. Instead of operating on a single file, however, it simultaneously uses files on each of the underlying drives for each I/O operation, providing the necessary redundancy for RAID. Each of these files is accessed through a DAX mapping, allowing us to take advantage of CXL-based storage. Since the library is implemented in userspace, the kernel stays out of the data path as desired. In our evaluation, we show that *CXLRaid* can achieve a higher bandwidth than the Linux kernel's RAID implementation md/raid [65] for reads and writes, while not requiring any modifications to the application.
In the following chapters, we will describe *CXLRaid* in more detail. Chapter 2 provides the necessary background required to understand the design and imple-

mentation of *CXLRaid*. After that, we explore related work in Chapter 3. Design considerations, implementation details, and challenges are laid out in Chapter 4. We benchmark, evaluate, and discuss our implementation in Chapter 5. Chapter 6 gives an outlook on possible improvements and future development directions. Finally, we conclude our thesis in Chapter 7.

Chapter 2

Background

In the following sections we discuss the technologies and concepts that this work builds upon. We start by introducing RAID, which we will implement in this work, why it is important that it exists and how it works. In addition, we introduce mdraid, the standard software RAID implementation for Linux, which has similar targets but different ways to achieve them. Then we introduce Compute Express Link, the new interconnect technology that our implementation is designed for. We discuss its features, especially regarding persistent storage devices, as this is the use case we are targeting. Additionally, we briefly bring up Intel Optane, a similar technology that we will be using in our evaluation. Finally, we talk about the Linux DAX interface, which we will be using to interact with those persistent storage devices. In the end, this chapter should provide the necessary background knowledge to understand the design and implementation of our RAID solution, as well as the motivation behind it.

2.1 RAID

With the increasing importance of data availability and the growing volume of stored information, relying on a single disk drive has become insufficient in many applications [16]. Redundant Array of Independent Disks (RAID) [59] addresses these challenges by utilizing multiple physical disk drives. The main goal is to provide fault tolerance and/or performance improvements by distributing data across these drives. Fault tolerance means that even if one (or more, depending on the implementation) disk fails, the data is still accessible. This is typically achieved through redundancy, meaning that additional information is stored to allow reconstruction of data in case of disk failure.

Performance improvements can be achieved, as multiple disks can be written to or read from in parallel.
Typical RAID implementations handle this transparently, meaning that multiple physical disks are presented as a single logical unit. This allows the use of the RAID array without needing to make every application aware of it. The translation is usually done in hardware (e.g., in a RAID controller) or in the operating system kernel (e.g., mdraid in Linux, see Section 2.2) [53]. Hardware RAID solutions typically use dedicated controllers to manage the array independently of the host system, which can reduce CPU load and improve performance. In contrast, software RAID implementations rely on the operating system to manage the array, offering greater flexibility and lower cost at the expense of additional system resource usage.
There are predefined strategies for distributing data across the drives. These strategies are called RAID levels. The standard levels are numbered 0 through 6, with each level offering different trade-offs between performance, redundancy, and storage capacity [16].
This work focuses on RAID 1 and RAID 5, which are both commonly used in practice [5]. We discuss them in more detail in the following sections.

2.1.1 RAID 1

RAID 1 [59], also known as *mirroring*, is a RAID level that provides redundancy by duplicating data across all disks, so every disk has an exact copy of all data. This means that the maximum storage capacity is only equal to the size of the smallest disk in the array. In return, RAID 1 offers high fault tolerance, as all disks would have to fail for data loss to occur. Read performance can be good, as data can be read from any of the disks in the array, so maximum parallelism can be used. However, write performance is generally slower than with a single disk, as data must be written multiple times. This makes RAID 1 a popular choice for critical applications where data integrity is paramount [59].

2.1.2 RAID 5

RAID 5 is also known as *striping with distributed parity*. It divides data into blocks of a certain size (called stripe size) and distributes these blocks stripe-wise across all disks in the array. Each of these stripes also includes a block of parity information, which is calculated as the bitwise XOR of the data blocks in the stripe. The parity block is distributed across all disks in a round-robin fashion, so that no single disk becomes a bottleneck.

Storage capacity in RAID 5 is equal to the size of the smallest disk multiplied by the number of disks minus one, as one disk's worth of space is used for parity. RAID 5 can tolerate the failure of a single disk without data loss, as the missing data can be reconstructed using the parity information. However, if a second disk fails before the first failed disk is replaced and rebuilt, data loss will occur.
Read performance is generally high, as data can be read from multiple disks in parallel. However, write performance can be significantly slower than a single disk, as both the data and parity information must be updated for each write operation. This makes RAID 5 a popular choice for applications that require a balance between performance and fault tolerance [59].

2.1.3 Journal & Write Hole

Since data and parity cannot be written atomically at the same time, RAID can suffer from the so-called "write hole" problem [139]. If a power failure or system crash occurs during a write operation, data between disks can become inconsistent, without a way to determine which data is correct. This can lead to data loss or corruption, especially in RAID 5, where parity information is crucial for data reconstruction.
To mitigate this issue, many RAID implementations use journaling [15], which is a technique that logs changes before they are applied to the disks. In the event of a failure, the journal can be used to recover to a consistent state by replaying it, thus ensuring data integrity. On the downside, journaling can introduce additional overhead and complexity to the RAID implementation.

2.2 md/raid

Linux has a built-in software RAID implementation called *md/raid* [65]. It is implemented as a kernel module and provides support for various RAID levels, including RAID 1 and RAID 5. It works by creating a virtual block device (`/dev/mdX`) over multiple other block devices (e.g. `/dev/sdX`). This virtual device can then be used like any other block device; most importantly, a filesystem can be created on it. This makes the implementation pretty straightforward, as the RAID logic only has to deal with fixed-size binary blocks and leave everything else to the filesystem on top of it. It also includes a journaling mechanism to mitigate the write hole problem described in Section 2.1.3 [15].

To administer the RAID array, the `mdadm` [66] utility can be used. It allows for creating and managing RAID arrays, as well as monitoring their status and performance.
In this work we will be comparing our software RAID implementation to md/raid (see Section 5.3), as it is the standard software RAID solution for Linux [33] and therefore widely used in practice.

2.3 Compute Express Link

In a typical computer system, components like memory and storage are connected to the CPU via different interconnects, each with its own performance characteristics and access methods. Memory, for example, uses a bus like *Double Data Rate* (DDR) [1], which is designed for byte-addressable access with low latency and high bandwidth. However, it is unsuitable for persistent storage devices, as these fail to meet the response time guarantees required for DDR. On the other hand, storage devices typically use interfaces like *Serial ATA* (SATA) [2] or *Non-Volatile Memory Express* (NVMe) [3], which are optimized for block-based, asynchronous access. They can handle the higher latencies of storage media, but are not designed for the byte-addressable access that is required for memory.
Modern workloads, however, especially in the field of data analytics and machine learning, often require both high performance and large capacity [51]. This has led to the development of new technologies that aim to bridge the gap between memory and storage. One such technology is *Compute Express Link* (CXL) [44], which is an open industry standard interconnect designed to provide high-speed communication between CPUs and various types of devices, including memory as well as storage. Initially developed by Intel and released in 2019 [30], CXL continues to gain support from a wide range of industry leaders, making it a promising technology for the future of computing.
CXL is not only designed to provide high bandwidth with low latency, but also cache-coherent access, which means that devices connected via CXL can share memory and maintain cache coherence with the CPU. This allows for more efficient data sharing and can significantly improve performance for certain workloads [30].
The CXL specification defines three protocols that build on top of the *PCI Express* (PCIe) physical layer [44:Chapter 3]:

- **CXL.io** is the basic protocol that provides device discovery, configuration, and management. It also provides an asynchronous, block-based access method for linked devices, similar to traditional storage interfaces.
- **CXL.mem** defines a cache-coherent, byte-addressable access method for the storage of the connected device using load/store instructions, similar to traditional memory access.
- **CXL.cache** allows devices to cache data from other connected devices' memory while maintaining cache coherence. This can improve performance by reducing latency by offloading some of the memory access to the device itself.

CXL is designed to support a wide range of devices with all kinds of different requirements, operation modes, and capabilities. In order to provide a clear distinction on how devices can be used by the host system, the CXL specification divides them into three types based on how they interact with the host memory [44:Chapter 2]:

- **Type 1 devices** do not have accessible memory but depend on host memory for their operation. They use CXL.io and CXL.cache protocols to access host memory and potentially cache data locally. Network cards and accelerators are examples of Type 1 devices.
- **Type 2 devices** have their own memory, which can be accessed by the host and vice versa. They use all three CXL protocols to ensure consistency between host and device memory caches. Typical examples are GPUs and FPGAs.
- **Type 3 devices** are devices that do not have their own processing capabilities but provide addressable storage. This means that the host can use them as an extension to its own memory. These devices can use any technology as their underlying media and allow it to be accessed using CXL.io or CXL.mem. This can be used for persistent storage devices, which we will describe in Section 2.3.1.

Usually, CXL-based access is transparent to applications; they use block-based I/O for persistent storage and load/store instructions for memory without being aware of the underlying interconnect. In this thesis, however, we will be accessing persistent storage in a byte-wise manner, requiring the use of CXL.mem (or similar technology). Therefore, we are specifically interested in Type 3 devices, which support this use case.

2.3.1 CXL-Based Persistent Storage

With the introduction of CXL, hardware manufacturers have started to develop new types of persistent storage devices that can be accessed using CXL protocols. These devices combine large-capacity NAND flash storage with a high-performance DRAM cache to offer the capacity of traditional SSDs with the access characteristics of memory. Because of that, they are called *hybrid SSDs* [42].

Modern operating systems, like Linux, support a hardware architecture called *Non-Uniform Memory Access* (NUMA) [67], which allows for multiple memory nodes with different access characteristics. In this terminology, a NUMA node is a group of processors and memory that have uniform access latency. Since hybrid SSDs can be accessed in a byte-addressable manner using CXL.mem, they can simply be treated as an additional NUMA node that does not have its own CPU by the operating system. This means that applications can access the storage using load/store instructions, without the need for special APIs or drivers.

At the time of writing, prototype hybrid SSDs already exist, for example the Samsung CMM-H (*CXL Memory Module – Hybrid*) [130], which features a 1 TB NVMe SSD combined with 48 GB of DRAM cache. A second manufacturer is Wolley, which designates its product as "NVMe-over-CXL" [135]. However, as these devices are not yet widely available, we will be using a technology with similar access characteristics called *Intel Optane* in our evaluation, which we will describe in Section 2.4.

2.4 Intel Optane

Back in 2015, Intel and Micron jointly announced a new type of non-volatile memory technology called *3D XPoint*, which was later branded as *Intel Optane* [32]. Similar to CXL-based persistent storage, Optane modules provide byte-addressable access to their whole capacity. In contrast to CXL, however, they are not connected to the CPU via PCIe, but instead use the DDR memory interface. This means that they compete with volatile DRAM for DIMM slots on the motherboard. Intel retired the Optane product line in 2022 however, so it is no longer available for purchase [32].

2.5 Linux DAX

Linux has an interface called *Direct Access* (DAX) [34], which was built to take advantage of Optane's persistent memory capabilities. It allows applications to directly access the memory on the device without going through the kernel's page cache. Since CXL.mem provides a similar byte-addressable access method, the same DAX interface can be used to interact with CXL-based persistent storage devices [42].
Managing DAX devices is done through the `ndctl` [60] utility, which allows the creation of regions and namespaces on devices. A region is a contiguous block of persistent memory, while a namespace is a logical division of a region that can be accessed by the operating system, similar to partitions on a disk. When creating a namespace, the user can choose between four different modes [21]:

- **fsdax** allows access through a `/dev/pmemX.Y` device on which a filesystem can be created.
- **devdax** allows access through a `/dev/daxX.Y` device, which can be directly memory-mapped by applications without the need for any filesystem on it.
- **sector** is similar to fsdax, but for legacy filesystems or interoperability with other operating systems.
- **raw** exposes the namespace as a block device, so block-based I/O can be used, but no DAX access is possible.

Since we implement a RAID layer on top of a filesystem, we will be using the `fsdax` mode. When mounting a filesystem on a DAX-enabled device, the `-o dax` option must be used to enable DAX access. At the time of writing, two modern Linux filesystems support DAX: *ext4* and *XFS* [34]. For our evaluation in Section 5, we will be using ext4, as it is the default filesystem for most Linux distributions and therefore more widely used in practice [31, 68].
From an application's perspective, DAX access is mostly transparent, as applications can continue to use the standard file I/O APIs. However, it enables them to pass the `MAP_SYNC` flag to `mmap()` to create a DAX mapping of the file. Operations on this mapping are equivalent to direct load/store instructions to the underlying storage, without involving the kernel. This means that even in the case of a crash, the written data is guaranteed to be persisted [69].

Chapter 3

Related Work

In this chapter, we discuss previous work related to our thesis. As the focus of our work is on providing RAID functionality on the file level using Transparent DAX Mappings for CXL-based persistent storage, we discuss works that touch at least one of these topics. We categorize the related work into three sections: Transparent DAX Mappings, RAID on file level, and filesystems for CXL-based storage.

3.1 Transparent Mappings

Khalil et al. [50] proposed *Transparent DAX Mappings* (TDMs) for CXL-based persistent storage. They implemented a dynamic userspace library that uses these mappings to provide the normal `read` [63] and `write` [64] data path, similar to our approach in Section 4.1. Additionally, they implemented a "Transparent DAX Mappings Manager" in the kernel, which is responsible for deciding which files get elevated to use the TDMs. The focus of their evaluation is to measure if these mappings are beneficial to applications, even when they are not aware of them. With YCSB [45] and Valkey [132], they showed an increment of up to 96.2% in throughput, highlighting the potential of this technique. In contrast to our work, they did not implement RAID functionality or similar, but instead integrated with the normal filesystem.

A similar approach was implemented by Kadekodi et al. [49] in their SplitFS project. SplitFS is also a dynamic userspace library that leverages DAX mappings to provide the I/O to files on persistent memory. It was designed with Intel Optane (see Section 2.4) in mind, however, and therefore lacks the resource management required

for CXL storage [50]. But since we use Optane for our evaluation too (see Section 5), their work shares many traits with ours. They also did not implement any RAID functionality.

3.2 RAID on File Level

The *Hadoop Distributed File System* (HDFS), developed by Shvachko et al. [129], is a distributed filesystem for storing a huge amount of data across multiple machines, which they call nodes. It has support for providing redundancy through RAID-like techniques on the file level. For that, each file is split into fixed-size blocks, which are then replicated across multiple nodes. While this RAID strategy is similar to ours (see Section 4.11), the data path is over the network and therefore different from what we implement.

Joukov et al. [48] introduced *Redundant Array of Independent Filesystems* (RAIF), a filesystem that provides RAID functionality on top of other file systems. With that approach it shares similarity with our work, as it uses regular folders as storage backends and provides RAID by creating files inside these folders. While this is also a RAID system on the file level, it is implemented completely in the kernel, a huge difference from our userspace-only approach. The focus of their work is also to support vastly different filesystems as backends, while we allow only DAX-capable filesystems, as this is required for DAX mappings to work (see Section 2.5).

Lucka [55] implemented a RAID on the file level for GPU4FS [56], a GPU-accelerated filesystem. GPU4FS uses Intel Optane modules as persistent memory storage, which are also directly accessed. However, the filesystem is designed to be run by the GPU, making the RAID implementation quite different from ours. The focus of their work is not primarily on optimizing the bandwidth, but the CPU load during filesystem operations.

3.3 Filesystems for CXL-Based Storage

Shen et al. [128] proposed AutoSSD, a RAID system for CXL SSDs. This is a RAID implementation designed with CXL in mind, but it is implemented below the block layer and therefore in the kernel. Their focus was on whether CXL can be used to reduce tail latency for RAID systems over the standard md/raid [65] driver. In their

evaluation, they show that using CXL can lower the tail latency by 13.3–44.3%. While this shows the potential of CXL-based persistent storage for RAID systems, their implementation works on blocks rather than files and therefore does not leverage DAX mappings.

Another CXL-aware filesystem is RomeFS by Zhan et al. [140]. The goal of RomeFS is to leverage both `CXL.mem` and `CXL.io` protocols for storage transactions. While aligned blocks use `CXL.io`, remaining bytes are sent over `CXL.mem`. Since both interfaces can be used independently, this also allows for easy parallelization of the I/O. With this approach, they show an improvement of up to 3.28% compared to ext4 [68]. While this shows that CXL access characteristics can be used to improve performance, they also do not use DAX mappings and do not implement RAID functionality.

Chapter 4

Design

In this chapter, we will present the design of *CXLRaid*, our proposed solution to a userspace RAID implementation that leverages DAX mappings. Here we mean with userspace, that we want filesystem operations, especially reads and writes, to do as few system calls as possible, ideally none at all. This is possible thanks to the DAX mappings, which we described in Section 2.5. Eliminating the kernel entirely from the data path should allow us to achieve higher performance than traditional RAID implementations, which rely on the kernel to perform reads and writes [65].
A major design goal of *CXLRaid* is the possibility to run applications without requiring any changes to them. In the best case scenario, they should not be able to tell at all that they use our RAID instead of a normal filesystem.
CXLRaid is written with CXL-based persistent storage (see Section 2.3.1) in mind, but should work with any DAX-enabled storage devices, including Intel Optane modules (see Section 2.4). We target Linux as the operating system, as it implements the required DAX interface [69].

4.1 Approach

Linux provides a rich API for applications to interact with, a large part of which is standardized by the POSIX specification [41]. Among other things, it defines a set of functions for filesystem operations, including `open` [70], `read` [63], `write` [64] and `close` [71]. All applications use this API to interact with the filesystem, either directly or through higher-level abstractions. *CXLRaid* implements the relevant API parts to provide our desired RAID functionality.

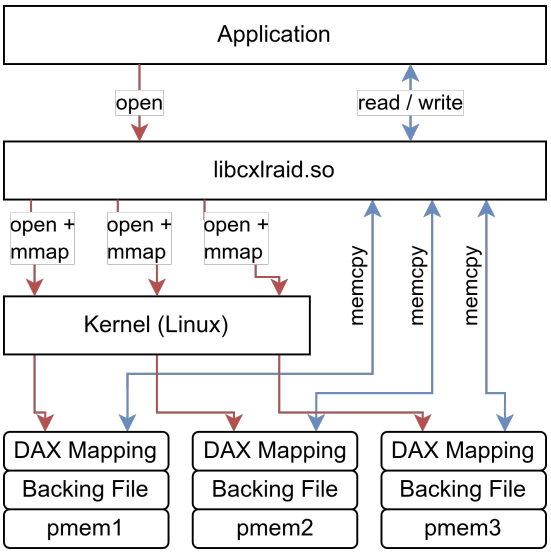

Figure 1: The approach of *CXLRaid*. Red lines indicate the data paths for `open`. Blue lines indicate the data paths for `read` and `write`, notably bypassing the kernel. Each pmem is separate drive with a DAX-enabled filesystem on it.

The basic idea of our design, as visualized in Figure 1, is to create a file on every drive in the RAID for each file the application creates, which we call *backing files*. For each backing file, we create a DAX mapping by using the `mmap` [69] call with the `MAP_SYNC` flag. This extra work is required, as a DAX mapping can only map a single file, and multiple drives dictate the use of multiple files. In the aforementioned `read` and `write` functions, we then perform the necessary RAID logic to read and write data from and to the corresponding positions in the mappings using a `memcpy` [72] or similar.

We implement the logic in a shared library called `libcxlraid.so`, which we will refer to as *our library* from now on. Since the POSIX API is designed as a C interface, we implement our library in C as well. Implemented features include RAID levels 1 and 5 (see Section 2.1) and optional support for a journal (see Section 2.1.3). Design considerations of our library are discussed in the following sections. Supplementary to our library, we also implement a small command-line interface (CLI) for administration tasks, which we will talk about in Section 4.14.

4.2 Configuration

Configuration of our library is done using environment variables. These are read at initialization time and cannot be changed at runtime. The following environment variables are used:

ENV	Description	Required	Default
`CXLRAID_LEVEL`	The RAID level to use.	YES	-
`CXLRAID_MOUNTS`	A colon-separated list of directories which are used as drive roots.	YES	-
`CXLRAID_PATHS`	A colon-separated list of paths. Each file whose full path starts with one of these paths is seen as part of the RAID (see Section 4.5).	NO	`/home/:/root/`
`CXLRAID_JOURNAL`	The path to the file to use as journal (see Section 4.10).	NO	-
`CXLRAID_STRIPE_SIZE`	The stripe size in bytes for RAID levels that do striping.	NO	`4096`

Table 1: *CXLRaid* configuration options

In the current implementation, we only support RAID 1 and RAID 5, but the design of our library allows for more RAID levels to be added in the future (see Section 4.11). The mount paths and the journal file (see Section 4.10) must all be on a DAX-enabled filesystem, as both use DAX mappings for reading and writing data without a fallback.
While technically not required, the mount paths should all point to mounts of different devices, otherwise using the RAID would not make much sense. The same goes for the journal file, which should be put on an additional device, otherwise a failure of the device with the journal file on it would render it useless.

The order of mount paths is important for RAID levels that require a specific order of drives. In our implementation, this is only the case for RAID 5, where the last drive is used for parity of the first stripe, the second-to-last drive is used for parity of the second stripe, and so on.
The content under the mount paths is expected to be created solely through our library, otherwise unexpected behavior may occur, as we expect identical content across all mount paths. For things like filesystem structure (directories, symlinks, etc., see Section 4.6) or file metadata (permissions, ownership, etc., see Section 4.7), our library usually only looks up one of the mounts for performance reasons.

4.3 LD_PRELOAD

In order to transparently intercept file operations, we need the targeted program to call our library functions instead of the corresponding Linux system call. However, hooking system calls is either slow or requires binary rewriting [35, 73, 136]. Because of that, we make use of the fact that almost all programs do not call them directly, but instead use a wrapper function in the C standard library.
On Linux, when a program is loaded, the dynamic linker [74] (`/lib/ld-linux.so.*`) is executed. It is responsible for loading all shared libraries that the program depends on into memory. After that, it resolves all symbols to the correct addresses, runs the initialization functions and finally starts the program.
By setting the `LD_PRELOAD` [74] environment variable to the path of our library, we can tell the dynamic linker to load it before all others. This way, when the linker resolves symbols, it will prefer the ones our preloaded library exports over the ones from other libraries, including the C standard library. Overwriting functions then becomes as simple as providing a function with the same name and signature as the one we want to intercept.
This approach allows us to intercept function calls in userspace without modifying the application, as needed by our design goals. Additionally, it does not introduce any overhead for function calls, as the redirection is done only once at initialization time.

It is important to note, however, that there are cases where this approach does not work. If a program does a system call directly, it cannot be intercepted by our library. This means that the `syscall` (or similar, depending on the architecture) instruction is executed within the program. Usually that happens when the C standard library is statically linked. In theory, however, a program could also do this using inline assembly. In Section 6.1.7 we discuss possible solutions to this problem.
We also cannot intercept calls of wrapper functions done by the C standard library itself, as those are not resolved through the dynamic linker, but instead are hardcoded jumps to the correct address. This means in order to provide a complete interception, we have to overwrite every function that can internally call a file system operation, not just the basic `open`, `read`, `write`, etc. This is especially a problem for the stdio API [75], which provides a complete abstraction over POSIX I/O consisting of a lot of functions. In Section 4.12 we talk about how we deal with this specific problem.
There also is a `syscall` [76] function in the C standard library that wraps the `syscall` instruction, providing a simple way for programs to do a system call using just their number to identify them. This function is typically used to make system calls that do not have a wrapper function, for example `userfaultfd` [77], which we use in Section 4.13. Since implementing this function would require extra code for every intercepted system call, and it is rarely used in practice for those, we currently do not overwrite it. This can be done in the future, however, if needed (see Section 6.1.7).

4.3.1 libdl

Alongside the dynamic linker, there is also the `libdl` [43] library, which allows dynamic loading and symbol resolution at runtime. Among others, it provides the `dlopen` [78] function, which can be used to load shared libraries into memory, and the `dlsym` [79] function to then resolve symbols from that memory.
This is used by our library to gain access to the symbols that it overwrites from the standard C library. For example, while `open` is overwritten to set up the mappings, we still need to be able to call the original `open` function to actually open the backing files on the drive.

In the case of standard C library functions, we do not even need to open the library manually, but can pass the special value `RTLD_NEXT` to `dlsym`. This tells it to look for the next occurrence of the symbol in the libraries loaded after the caller. Since `LD_PRELOAD` guarantees that our library is loaded before all others, this will always resolve.
While this makes it easy for us to call the original functions, it provides similar functionality to the application we are intercepting. Since it is possible to tell `libdl` exactly from which library to resolve the symbol, our overwrites will not be returned. It is, however, unlikely for an application to load standard C library functions manually, so this is usually not a problem in practice. If needed in the future, however, one could also overwrite `dlsym` to prevent this, since it is a normal function itself, also resolved through the dynamic linker.

4.3.2 Initialization

The binary format of shared libraries on Linux is the ELF format [80]. It provides a special section for initialization functions, `.init_array` [4], which contains a list of function pointers. When our library is loaded, either through the dynamic linker or `dlopen`, the functions in this section are executed in order by the loading mechanism. The two big C compilers for Linux, `gcc` and `clang`, allow adding functions to this section by marking them with the `constructor` attribute [6, 20]. It optionally takes a priority argument, which determines in which order the functions of the same library are executed. Our library uses this to initialize itself before the `main` function of the application is executed. This allows us to parse the configuration (see Section 4.2) and set up things like the file table (see Section 4.4) once at program startup, while still not requiring any changes to the application. The alternative would be to check for initialization in every overwritten function, which would introduce a slight overhead to every function call, complicate the code significantly and could still lead to crashes, as described later in this section.
Using constructor functions for initialization does not come without its own problems, however. The linker guarantees that the functions of a library are executed in order, and that the init functions of dependencies are executed before the ones of the library itself. For example, if a library links against the standard C library, the init functions of the standard C library are always executed first. If two libraries do not have a dependency on each other, however, the order of initialization between them is undefined. This means that if our library is preloaded to a program that

dynamically links against a library other than the standard C library, the constructors of that library may be executed before the ones of our library. If that library now calls a file system operation from within its constructor function, it will call into an uninitialized state of our library, because symbols are already resolved at that point. The only way to prevent this would be to link this library against our library, which is not applicable in our use case because we want to be able to preload our library to any program without needing to recompile it.
If we decide to solve this by checking for initialization at every function call, this would still fail in some cases. The program `valkey` [132], for example, links against `libjemalloc` [47], which provides a `malloc` [81] implementation. In the constructor it reads `/etc/malloc.conf` to initialize itself. If this read now triggers an initialization of our library, the whole program would crash, as our library depends on `malloc` in its constructor code to reserve memory for its internal data structures. `malloc`, however, is not in a usable state yet, creating a non-trivial circular dependency problem.
To solve this, we implement a hybrid approach. We set a static variable `initialized` to false, which is set to true at the end of our constructor. In selected functions, we check if this variable is true, and if not forward the call to the standard C library implementation. In order to provide that forwarding, we write wrappers around the standard C library functions we use. These wrappers save the resulting function pointer in a static variable and call `dlsym` if it is not set yet, which in turn must be checked at every call. We argue however that the overhead of this is negligible, as it is a simple boolen check.
This means that while the program will not crash, RAID functionality will only be available after initialization, which happens at an undefined point. In practice this should be fine, however, as initialization functions usually do not call file system operations. And if they do, it is effectively always for files our library considers part of the system (see Section 4.5, for example configuration files in `/etc`), which are not seen as part of the RAID anyway.
For now, we check for initialization only in a few selected functions, which seems to be enough in practice. These are the functions needed for stdio (see Section 4.12) and all functions that operate on a path directly, since the check for initialization is part of the logic to determine if a file is part of the RAID or the system. It is easy, however, to add this check to more functions in the future if needed for a particular program.

4.4 Filetable

On Linux, a *File Descriptor* (FD) [70] is a non-negative integer, which represents a resource handle to the kernel. Among other operations, this handle can be used to read and write data, hence it is also used to represent open files. In practice, this means that an `open` call must return a number that is understood by functions like `read` and `write` to operate on the correct file.
Internally, a FD is an index into a per-process array, called the *file descriptor table* [36]. It consists of `struct fds` [37], which only contain a pointer to a `struct file` [38], which we call an *open file handle*. Each handle contains the information unique to specific opens of a file, including the current read and write offset, the allowed access modes, and importantly, a pointer to a `struct inode` [39]. An inode contains system-wide information about a file that is not only shared between multiple opens of the same file, but also between all processes on the system. This includes a unique identifier and the current size of the file. These levels of indirection are needed because calls like `dup` [82] allow multiple FDs to point to the same open file, and a file can be opened multiple times, even with different flags.
In our library, we cannot use the file descriptor table of the kernel because we need to store information at different levels of this hierarchy, to which we have no direct access to. For example, we need to store the DAX mappings and actual file size at the inode level, but the read and write offset at the open file handle. We cannot use the information the kernel provides, since it is not aware of the RAID. The size of a backing file, for example, is never the size of a file from an application's perspective (see Section 4.8). Additionally, as a goal of our design is to work in userspace, keeping this information in userspace too allows for reading and writing without needing to make a single system call.
This led us to implement our own file table, which closely mimics the structure the Linux kernel uses, but entirely in userspace. In return, there is a clear distinction between FDs the Linux kernel can handle and the ones given out to the application. A huge task of our library is to make sure that these two are never mixed.
Because FDs can represent arbitrary resources, our library needs to support non-file FDs as well. For that it divides inodes into three categories:

- Regular files, which are part of the RAID. The file size and a list of file mapping information are stored. Each file mapping information contains the Linux FD for the backing file, the DAX mapping, and the size of the mapping.

- Directories, which are part of the RAID. A list of directory mapping information is stored. Each directory mapping information contains the Linux FD for the backing directory.
- System resources, which are not part of the RAID, for example network sockets (see Section 4.4.1). The corresponding Linux FD is stored.

All functions that our library overwrites (see Section A.1) are aware of this separation. For example, when `read` encounters a system inode, it will simply call the original `read` function with the corresponding Linux FD. When encountering a regular file, it will perform reads from the DAX mappings according to the arguments, the current RAID level, and the data in the open file handle. For directories, it will correctly fail with `EISDIR` [63].
Additional to the type-specific data, each inode stores a unique path, which is used to identify it. For files and directories, this is the path relative to the mount paths. For system resources, this is a "`system:`" prefix followed by a decimal representation of the Linux FD. This is needed to prevent having two inodes representing the same resource, which would lead to inconsistencies. In order to only close the inode when the last open file handle is closed, we also store a simple reference counter.
Each open file handle stores a reference to the corresponding inode and a reference counter for the FDs referencing it. It also has a `flags` field, currently used for permissions (`O_RDONLY`, `O_WRONLY`, `O_RDWR`) and write mode (`O_APPEND`) [70]. The read and write offset is also stored here, as it is specific to each open of a file, but only used if the inode type is a regular file.
The FD structure itself contains a reference to the open file handle. In order to fail fast on invalid FDs, it also contains an `is_open` field. This is set to false when a FD is closed, but the memory cannot be freed yet, because a FD within the same memory region is still open. It also has a `flags` field, which can store the `FD_CLOEXEC` [83] flag. The current implementation correctly handles this flag in methods that get and set it (like `fcntl` [83]), but passing the file table through an `exec` [84] call is not implemented. Everything will be freed in the destructor of our library, so applications that rely on keeping FDs open will not work. In Section 6.1.4 we present how a possible implementation could look like. In practice, however, almost all applications seem to want the closing behavior anyway; `valkey`, for example, explicitly sets the `FD_CLOEXEC` flag on its FDs [133].

4.4.1 Shims

Managing every file descriptor is a huge task, as there is an extremely large number of functions that operate on them in some way [41]. For applications to run correctly, our library would therefore have to overwrite every single one of those functions. Even if all it would do is translate the FD from the application to the corresponding Linux FD, and then call the real function. This is especially cumbersome, for example, with everything network-related, as these APIs make heavy use of FDs but have nothing to do with RAID. `socket` [85], for example, returns a FD, which is then used by `bind` [86], `connect` [87], `send` [88], `listen` [89], `recv` [90], and so on. To solve this, our library makes use of the fact that while there are a lot of functions that operate on FDs, there are only a few that actually create them. When overwriting a function that creates a FD, we essentially always do these four steps:

1. Make sure that every FD we currently have open is also open in the Linux kernel. If not, we use `dup2` [82] to duplicate a valid Linux FD (`STDIN_FILENO` [91] in the current implementation) to the same number as the application FD. This is not only a relatively cheap operation, but also unlikely to happen. Since each FD our library gives out opens at least one file in the Linux kernel, the application has to perform a specific sequence of opens and closes for this to occur.
2. Call the real function to create the FD in the Linux kernel. Handle any error that may occur and set up things specific to the overwritten function if required.
3. Register the new Linux FD in our file table, at the same number, with a system resource inode type. Note that this can break the Linux guarantee of always using the lowest available FD [70], but this does not seem to cause any problems in practice.
4. Close any Linux FDs that we opened in step 1, and return the FD.

This way applications can safely pass the FDs directly to kernel functions, without our library needing to translate them. We call those wrappers *shims*, as they are only a thin compatibility layer between the application and the kernel. We use those shims for functions that create FDs which do not represent a file or a directory. Currently these are the functions listed in Section A.1.4, but more can be added in the future if needed.

4.5 System Files

With our library redirecting file operations to the RAID mounts, it effectively creates a sandbox in which the application can only see files that were added to the RAID through previous invocations of our library. This usually leads to problems, as the application expects to run in a normal Linux environment. On Linux, for example, not only regular files and directories exist, but also a lot of special files, which are used to interact with the kernel and other system resources. Notable examples are the `/proc` [92], `/sys` [93] and `/dev` [94] directories, which contain whole filesystems that are not stored on drive, and therefore not applicable for RAID. There are also some regular files that are expected to be present on the system, for example global configuration files in `/etc`. While those could theoretically be added to the RAID beforehand, this would make running applications through our library much more laborious.

To address this problem, our library differentiates between "System" and "RAID" files. The decision on which file falls into which category is based on the prefix of its full path. We keep a whitelist of prefixes for RAID files, which can be configured if desired (see Section 4.2). By default, this whitelist contains the home directories (`/home/` and `/root/`), as often the desired behaviour is to store files created by the application in RAID. In most cases, these get placed either relative to the home directory, or relative to the current working directory, which is often within the home directory as well.

If a file is considered to be part of the system, it gets an inode of the type *system* (see Section 4.4). This means that all file operations are simply forwarded and happen on the host filesystem, without any RAID operations or DAX mappings involved. Otherwise it is handled by the regular RAID logic. This allows applications to run correctly while simultaneously using the RAID for important data.

There are some edge cases where this separation needs special handling. For example, we implemented special logic in the `rename` [95] function to correctly move files between both systems. Most notably missing is the correct creation and handling of hard links that point from the system into the RAID, as they have the system prefix, but should theoretically be considered part of the RAID. If this is needed in the future, it can be implemented (see Section 6.1.6), but for now it is not worth the effort. If an application is not specifically designed to break our library, it is very unlikely that it runs into one of those edge cases.

4.5.1 /proc/self/fd

In Linux, the `/proc/<pid>/fd` directory (or any symbolic link to it, like `/proc/self/fd` and `/dev/fd`) exposes open file descriptors as files in the standard filesystem tree [92]. In this pseudo-directory, each open file descriptor is presented as a symbolic link to the resource it represents. For example, `/proc/self/fd/7` will point to the same file as the FD `7`. We therefore implemented a special logic in our `readlink` [96] implementation that looks up the correct path of FDs in our file table. Our path resolution algorithm, which is used for all path lookups, uses this function so that these special paths get handled correctly.

4.6 Filesystem Structure

In our library, we need to resolve paths as if no RAID was present. Since we redirect any path-based operations to the RAID mounts, we need to have a suitable file structure there. In order to have absolute and relative paths work correctly together, we are required to have the same directory structure in the RAID as in the system. This means that each mount directory is seen as the root of the filesystem, and absolute paths are resolved from there.
Relative paths are resolved to absolute paths first and then treated as such. For that we implemented a custom path resolution function, which uses the existing directory structure in the RAID. The current working directory is guaranteed to exist, so relative paths can be resolved. However, it might only exist outside of the RAID, for example when the program is run for the first time. Because of that we create it in the RAID at initialization time. After that, we store and resolve it ourselves, because it might get set to directories that were created only in the RAID.

4.7 File Metadata

The filesystem not only stores file contents, but also metadata associated with each file. This includes things like permissions, ownership, and timestamps. Our library implements support for these by using the filesystem metadata of the backing files. Operations that set metadata, like `chmod` [97], `chown` [98], or `lockf` [99], do this on all backing files, keeping them in sync. Operations that query metadata, like `access` [100] or `pathconf` [101], try each file until one of them succeeds for the case of a broken drive, and return the result.

A special case are the various versions of the `stat` [102] function. These return a struct containing various metadata, notably including the file size. However, the size of a backing file in RAID levels that do striping is smaller than the size of the file from an application's perspective. This means that we cannot report it as is, but need to store and read the actual file size from somewhere else. Keeping all the backing files at the same size as a single file would need would be a waste of storage space, essentially increasing the space consumption to that of RAID 1. We solve this by using the file header described in the next section.

4.8 File Header

The size problem brought up in Section 4.7 can be solved by storing the actual file size at the beginning of the file, effectively creating a header. In our implementation, we store an exact copy of the header at the beginning of each backing file, regardless of the parity strategy of the current RAID level. Storage size increase is negligible, as the header is only 16 bytes and written exactly once per backing file. The benefit is that we can compare the backing files for compatibility when opening them.
Since we already have a header, we also store some additional information there. These are not technically required, but prevent data corruption or loss from using incompatible configurations on the same mounts. An `open` [70] call on a file with an incompatible header is programmed to fail. If the backing file is too small to contain a header, it is considered incompatible as well. Exempt from that are open calls that truncate the file (when `O_CREAT`, `O_TRUNC` or `O_TMPFILE` flags are given); in this case the backing files will simply be overwritten, including the header with the current configuration.

```
struct CXLRaidHeader
{
    uint16_t header_version;
    uint16_t level;
    uint32_t stripe_size;
    size_t size;
};
```

Listing 1: The header structure stored at the beginning of each file in the RAID.

The actual header structure is shown in Listing 1. The sizes of the data types are chosen to be as small as needed for their purposes, while not sacrificing space in the struct due to alignment at the same time. The `header_version` field is a hardcoded number that should be increased whenever the header structure changes. This ensures that incompatible headers are detected correctly. The `level` and `stripe_size` fields are taken directly from the configuration. The `size` field is used to store the actual size of the file, as described above. We do not store some kind of magic number in the header, since we expect that our library will be the only code that reads and writes them. As stated in Section 4.2, the content under the mount paths is expected to be created solely through our library.

4.9 libpmem

In order to make development of applications for persistent memory easier, Intel developed the *Persistent Memory Development Kit* (PMDK) [22]. Part of this kit is the `libpmem` library [23], which provides primitives for mapping and writing to persistent memory. While the creation part comes down to a simple `mmap` [69] call with the `MAP_SYNC` flag [24], similar to what we implement ourselves (see Section 4.1), the writing part is interesting for us. For that it provides the `pmem_memcpy` and `pmem_memset` functions [7], with an intended usage similar to the standard `memcpy` [72] and `memset` [103] functions. In contrast to them, these specialized functions can execute more efficient instructions, as they have the constraint to only be used on persistent memory [23]. In addition, they also detect the capabilities of the CPU at runtime to leverage available features [25]. Because of the performance gains they provide, we use them over the standard functions for all writes to our DAX mappings. Note that these functions are only meant for writing data *to* persistent memory [7], so we still use the standard `memcpy` for reading data *from* the DAX mappings.

While the interface of the aforementioned functions is similar to the standard ones, they also take an additional flag parameter that can be used to control the copy behavior. Relevant for us are the `PMEM_F_MEM_NODRAIN` and `PMEM_F_MEM_NONTEMPORAL` / `PMEM_F_MEM_TEMPORAL` [7] flags.

The `PMEM_F_MEM_NODRAIN` flag tells these functions not to issue a so-called drain after the copy, which is implemented as an `sfence` [17] instruction [26]. This instruction ensures that all previous memory operations are completed before any subsequent ones are executed.
The `PMEM_F_MEM_NONTEMPORAL` flag tells these functions to use non-temporal instructions for copying, which most notably bypass the CPU caches [23]. The `PMEM_F_MEM_TEMPORAL` flag is the opposite of that.
To determine which flags to use for our library, we look at their performance differences, as we do not require any of the different guarantees they provide. We take eADR [137] and GPF [58] as given for Optane and CXL respectively, meaning written data will always automatically be persisted. For the measurement we run benchmarks using FIO (see Section 5.1) for the possible combinations. The system and configurations used, `mapping` and `raid1`, are the ones we describe in Section 5. In short, we write blocks of 4 KiB sequentially to a RAID 1, with `mapping` having a single drive and `raid1` having three drives.

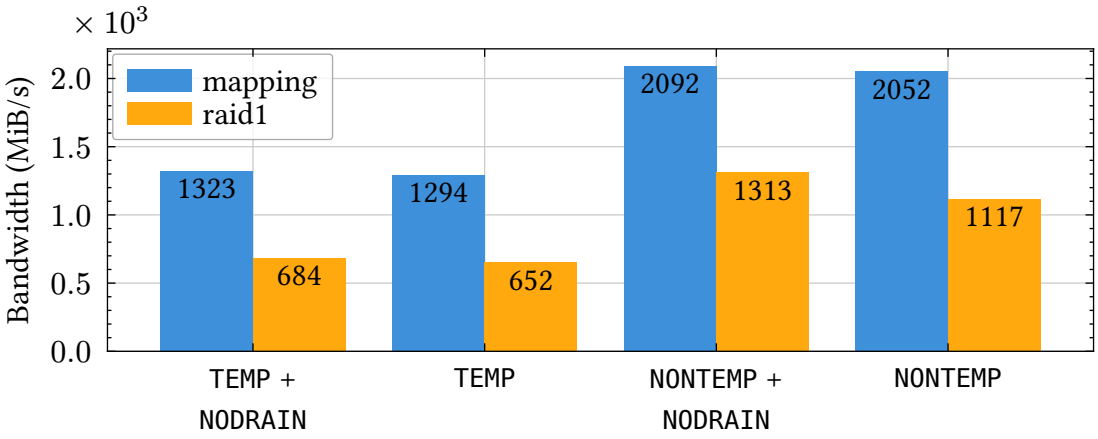

Figure 2: Bandwidth difference for possible `pmem_memcpy` flag combinations. Using non-temporal instructions gives a significant performance boost. Not issuing a drain after a copy gives a slight boost if multiple copies are done in a row.

Figure 2 shows the bandwidth differences for the `mapping` and `raid1` configurations between the possible combinations of those flags, tested with sequential writes (`readwrite=write`, see Section 5.1). Overall, non-temporal access gives a considerable performance boost in all cases; we suspect that otherwise the CPU caches are the bottleneck, as they can reorder the writes in a way that causes a lot of write amplification on the storage device. While the `mapping` configuration does

not significantly benefit from non-draining access, a difference can be observed for the `raid1` configuration. As it calls the `pmem_memcpy` function three times directly in a row, the drain, and therefore the wait issued after each of them, does make a noticeable negative impact on performance.
In our implementation, we settled on non-draining, non-temporal access, as this gives the best performance in all cases.

4.10 Journal

To address the write hole problem described in Section 2.1.3, our library implements optional support for a write journal. The journal is used to prevent inconsistent state after a system failure. Every operation that requires writing to multiple drives is logged to the journal before it is executed. In the event of a failure, the journal can be used to recover to a consistent state by replaying it, thus ensuring data integrity. This happens either automatically at initialization time or manually through the CLI (see Section 4.14).
As our library is a RAID implementation that works on the file level, the journal itself is also a file. The path must be configured by the user (see Section 4.2) and be on a DAX-enabled filesystem. In case of a journaled operation, the journal file is written to, then the operation is executed, and finally the journal file is cleared.

```
enum CXLRaidJournalAction { ... }; // CXLRAID_JOURNAL_ACTION_WRITE et al.

struct CXLRaidJournalPayload
{
    enum CXLRaidJournalAction action;
    union { ... }; // struct { uid_t uid; gid_t gid; } chown; et al.
};

struct CXLRaidJournalHeader
{
    uint16_t header_version;
    uint16_t path_size;
    struct CXLRaidJournalPayload payload;
    char path[];
};
```

Listing 2: The journal entry structure.

The layout of the data written to the journal is shown in Listing 2. Header and payload structs are separated so we could write more generic code. The `header_version` field is similar to the one in the file header (see Section 4.8). The `path_size` stores the length of the path, which can be accessed through the flexible array member `path`. This field doubles as a check if the journal contains an entry, because we set it to zero when not. This proved to be much faster than constant truncates of the file. The `payload` field contains all information about the operation, including the type of the operation and additional arguments like flags and offsets. In case of a write, the data to be written is stored after the path, and the payload contains the size of this data. The second path of a rename operation is stored the same way.
In order to ensure that the journal itself is in a consistent state, the header must be written atomically. This is achieved by using the `movdir64b` [18] instruction, which moves 64 bytes atomically. The header is only 32 bytes, so it can be written at once. We write the header after the data, so the path field stays zero until the end, so a crash during writing will not look like a valid journal entry. The journal will then be considered clean and not be replayed, and the state of the RAID will stay consistent. Note that the `movdir64b` instruction is not available on all processors, particulary not the one we used for the evaluation we present in Section 5. In that case our library will transparently fall back to the `pmem_memcpy` [7] function. While this does not impact performance, it might not be atomic, and therefore semantically not correct.

4.11 RAID

The implementation of the different RAID levels is done by providing three functions for each level:

- A read function that does a positioned read from the mappings, with an interface similar to `pread` [104].
- A write function that does a positioned write to the mappings, with an interface similar to `pwrite` [105].
- A truncate function that changes the size of the file, with an interface similar to `ftruncate` [106]. This function is required, since otherwise there would be no way to decrease the size of a file using the two functions above.

For each of these functions a wrapper function exists that simply checks for the current RAID level and calls the corresponding implementation. Every function that interacts with the content of a file, for example `write` [64], `preadv64v2` [107] or `fallocate` [108], is implemented on top of these wrappers. This way, we do not have to duplicate the RAID logic in multiple places. New levels can also be added easily by just implementing the three functions and adding a case to the respective wrapper.

4.11.1 RAID 1

RAID 1, as explained in Section 2.1.1, is straightforward to implement.
The truncate function truncates each backing file to the given size plus the header size, writes the new size into each header and resizes the mappings accordingly. In case of a size increase, the new space is automatically filled with zeros by the `ftruncate` [106] system call. Since there is no parity, we do not have to recalculate anything.
The write function calls the truncate function first, if the backing files are not large enough. Then the new data is copied into each mapping. If a drive is missing, it is simply skipped.
The read function does a round robin between all present mappings on each invocation to determine which mapping to read from. This is done to ensure that all drives are used equally. The data is then copied from the chosen mapping into the provided buffer.

4.11.2 RAID 5

RAID 5, described in Section 2.1.2, is more complex to implement, as we need to do striping as well as keep parity information in sync.
The truncate function calculates how big the backing files need to be for the given size, taking the stripe size, the number of drives, and the header size into account. It then performs a truncate similar to RAID 1 using the calculated size. However, we also need to ensure that the parity information is correct after a truncate. For that we make the convention that partial stripes are always padded with zeros. This way, a size increase does not have to recalculate parity at all. For partial stripes it will already be correct since both the old padding and new data are zeros. For new stripes it is correct as well, since `ftruncate` will automatically zero out new space, and `XOR`ing zeros with zeros any amount of times will always result in zeros. This makes decreasing the size more expensive in cases where we end up with a partial stripe. Padding must then be set explicitly to zeros. As this will probably change

parity, we need to recalculate it for this stripe. The truncate function does that by iterating over all blocks in the stripe, as multiple of them might have changed.
The alternative solution to always pad with zeros would be to recalculate parity when increasing the size, since zeros would need to be set for the new space. While this makes the size decrease trivial, we decided against it for two reasons. First of all, we expect that size increases are more common than decreases in practice[1], so it is better to optimize for them. Secondly, most size increases are done indirectly through writes, which will change the parity anyway, so calculating it in the truncate function would be unnecessary overhead in those cases.
Like the RAID 1 implementation, the RAID 5 write function first calls the truncate function to get the backing files and mappings correctly sized. It then copies the new data in chunks of the stripe size into the correct position and mapping. For each of these chunks, it recalculates the parity information by `XOR`ing the new data with the old data and the old parity. This is mathematically correct, as changing a bit in the data will result in a change in the parity bit, the same behaviour as if we would recalculate the parity from scratch. The advantage of this approach is that we do not have to read the whole stripe to recalculate parity, but only the old data and parity for the chunk we are writing, improving performance and storage usage, especially for small writes. In the special case where the chunk is to be written to a missing drive, we still calculate the new parity by `XOR`ing the whole stripe, because there is no fast way to get the old data. If the parity chunk is on a missing drive, we simply skip parity calculations.
The read function copies the data from the correct positions and mappings into the provided buffer. If a drive is missing, it `XOR`s the remaining drives to reconstruct the data. Since we never copy from the parity chunk, we do not have to implement any special handling if the parity chunk is on a missing drive.

[1] except truncates to size 0, but then there is also no data to calculate parity for.

4.12 stdio

The C standard library provides a high-level API for file I/O, called stdio [75]. It is designed to be cross-platform and provides a lot of convenience functionality, including buffering and formatted input and output. On Linux, it is implemented on top of the POSIX I/O API that our library provides. However, this is done in the same library as the calls we overwrite, so our functions do not get called (see Section 4.3). This is problematic, as almost all applications use the stdio API in some way, and without an implementation, operations done through it would ignore the RAID. Overwriting all of the stdio functions would be an unfeasibly huge task. Reusing existing implementations is not possible either, as they rely on internal functions of the C standard library they are implemented for, which we do not have access to.
The central struct of the stdio API is the `FILE` [75] struct, which is roughly equivalent to a file descriptor in the POSIX API. It contains metadata as well as pointers to functions that perform the actual operations like reading and writing. This way, a `FILE` can not only represent a regular file, but also things like a memory stream or a pipe. How the `FILE` struct is implemented, however, is internal to the C standard library and not standardized, so we cannot use it directly.

```
typedef struct
{
    ssize_t (*read)(void* cookie, char* buf, size_t nbytes);
    ssize_t (*write)(void* cookie, const char* buf, size_t nbytes);
    int (*seek)(void* cookie, off_t* offset, int whence);
    int (*close)(void* cookie);
} cookie_io_functions_t;
```

Listing 3: The `cookie_io_functions_t` struct[2].

Luckily for us, the GNU C standard [10], which is present in almost all Linux distributions, defines a `fopencookie` [109] function, which allows us to create a `FILE` struct that uses the function pointers from a given `cookie_io_functions_t` struct (see Listing 3). A `cookie` is a void pointer provided by the user that is passed to each function, supposed to store arbitrary state information. Using this, we only have to overwrite the `fopen`, `fdopen`, and `freopen` functions [110], and return the result of an `fopencookie` call with our own function pointers. The resulting `FILE` struct can

[2]The glibc implementation typedefs the function pointers to pretty names before, and then uses those names to define the fields of the struct [9]

then be passed to any stdio function without the need to overwrite them, as they will call the function pointers we provided.
The implementation of the functions we pass to `fopencookie` is straightforward, as they are very close to the corresponding POSIX functions. As the `cookie` argument we do not even pass a real pointer, but instead use it to directly store the file descriptor, which is the only state information we need.
A special case is the `fileno` [111] function, which is used to get the underlying native file descriptor from a `FILE` struct. In the case of one created by `fopencookie`, this is always `-1`, as it was not created on top of a FD. There is also no function pointer for it in the `cookie_io_functions_t` struct that we could easily implement. This is a problem, as some applications use this function to perform operations provided by the POSIX API, but not the stdio API. For example, `valkey` has the following line in its code to write the append-only file: `fsync(fileno(fp))` [134]. To solve this, we also save a mapping from each `FILE` struct to the corresponding file descriptor. In the overwritten `fileno`, we check if the given `FILE` struct is one of our `fopencookie` structs, and if so return the corresponding file descriptor from the mapping. Otherwise, we call the original `fileno` function.

4.13 mmap

As described in Section 4.1, memory mappings are an alternative way to read and write files. Not only does our library use them, but some applications do as well, so we have to support them too. This is a non-trivial task, however. First of all, the `mmap` [69] system call does not allow us to create mappings backed by multiple files, so we cannot have write operations to the mapping be directly written to the RAID. And even if this was possible, we could only support RAID 1 with it, as there is no way to implement striping or parity calculations with it. Second, Linux does not provide an API that notifies us when memory was written to, so we cannot simply copy the mappings to the RAID after a write has happened. This is especially problematic, as writes to a mapping happen through store instructions instead of function calls, so we cannot know what the data is until after it has been written.
What Linux allows us to do, however, is to be synchronously notified before a memory page is written to. This is done through the `userfaultfd` [77] system call, which creates a file descriptor that can be used to do page fault handling in userspace. Since Linux 5.7, it also supports a “write-protect” mode, that we will use.

After registering a memory range to be handled by our `userfaultfd`, we can set pages inside that range to be write-protected. If such a page is written to, the kernel will halt the writing thread and make an according event message readable on the `userfaultfd`. In another thread we can then read this message, obtain the address of the page that is being written to, and then unprotect it. This will resume the writing thread, making the write actually happen.

We use this mechanism to implement the `mmap` call. When a mapping is requested, we create an anonymous mapping aligned to page boundaries and make it write-protected. For each mapping, we also store some metadata that can be looked up using the address of this mapping. This metadata includes the file descriptor of the file being mapped, various offsets and sizes, synchronization primitives, and a "dirty" flag and bitset. This bitset is used to indicate which pages inside the mapping have been written to. If a write happens to this mapping, our `userfaultfd` thread will be notified about the page that is being written to. We can then look up the corresponding metadata by calculating in which mapping the page address is located. Then we mark the mapping as dirty, set the corresponding bit in the bitset and unprotect the page, which allows the write to happen.

In a third thread, that we call the "writeback" thread, we continuously check for dirty mappings. If we find one, we can use the metadata to issue a `pwrite` [105] call to write the changed pages to the file. Since we already overwrite `pwrite`, it will handle RAID logic transparently for us. After the write is done, we mark the mapping as clean again and reprotect the pages.

The tricky part about this is to get the synchronization between the threads right. On the one hand, we want our writeback thread to write changed pages to the RAID as soon as possible, to minimize the time between a write and an update of the RAID. On the other hand, we do not want it to write pages before the main thread finished writing to them. Since we need to lock the metadata of the mapping during both the writeback and the `userfaultfd` thread to ensure consistency, the main thread would have to wait for the writeback thread to finish before the `userfaultfd` thread allows it to continue. The page would be immediately marked dirty again, causing the writeback thread to make an almost identical write. In the worst case this loop would happen multiple times per write, causing a lot of overhead.

To solve this, the writeback thread calls `sched_yield` [112] after each iteration of checking for dirty mappings, but not any sleep function or similar. This tells the kernel to put the thread at the end of the scheduling queue, hopefully allowing the main thread to run and finish its write. While there is no guarantee about the scheduling order, it seems to work well in practice, giving us a good balance between writeback delay and write interrupts.
The other memory-mapping-related functions are relatively straightforward to implement after this. `munmap` [69] calls `msync` [113] before unregistering and unmapping the memory, which in turn calls the same writeback function as the writeback thread if the mapping is dirty. `mremap` [114] also writes back dirty pages, unregisters the memory range, remaps it, and registers the new range. All other functions, like `madvise` [115] and `mprotect` [116], should work without any special handling, as they work the same on our anonymous mapping as they would on a regular file mapping.
One thing to note is that a call to `userfaultfd` only succeeds if the calling process has the `CAP_SYS_PTRACE` capability or `/proc/sys/vm/unprivileged_userfaultfd` is set to `1` [77]. Both of those are not expected to be the default on usual systems. If run with insufficient permissions, our library will print an adequate warning at initialization time and calls to `mmap` with a file descriptor will always return `MAP_FAILED`.

4.14 CLI

Additional to our library, this project also provides a command line interface (CLI), called `cxlraid`, to interact with the RAID. This is required for some tasks that otherwise would be hard or impossible to do. The CLI is capable of that because it links to our library and uses internal functions. Our library also behaves differently when it detects that it is being used through the CLI; for example, it skips automatic journal replay. Because of that, the CLI is configured using the same environment variables as our library (see Section 4.2), so often it can be used without any additional configuration.

Following commands are currently provided by the CLI:

info - Prints various information about the RAID, including status, configuration, build and runtime information. An example output is shown in Section A.2.

add - Copies a file from the host to the RAID. Because of the sandbox nature described in Section 4.5, the program is not able to see files on the host whose prefixes match the configured RAID paths. This command works regardless of the configured RAID paths and can therefore also add files to the same path in the RAID as they have on the host. A common use case is to add configuration files, for example "`cxlraid add test/valkey.conf`".

retrieve - Copies a file from the RAID to the host. This command works the same way as `add`, but in the other direction. A common use case is to retrieve results from the RAID, for example "`cxlraid retrieve bench/fio/write_results.json`".

journal - Allows one to either replay the journal to recover from a failure, or clear it to prevent automatic recovery at initialization time of our library. Note that clearing the journal will probably leave the RAID in an inconsistent state, so it should only be used carefully.

repair - Iterates over all files on all mounts and checks if they are present on the other mounts. If not, it tries to restore them using RAID logic. This is intended to be used after a drive was replaced by a new one. It does not check if parities and data match, because there would be no way to tell which one is correct if they do not, making a repair in this case impossible.

help - Same as `--help`, prints usage information about the commands and some options.

Chapter 5

Evaluation

In this chapter, we evaluate the performance of our implementation to get an idea of how well the DAX mappings approach performs. For that, we first perform micro-benchmarks of *CXLRaid* itself. Then we look at the performance of a real-world application. Finally, we compare it to the software RAID implementation of the Linux kernel, md/raid.

While our main focus is to provide a solution for CXL persistent storage devices (see Section 2.3.1), we unfortunately do not have access to a CXL device for testing. Instead, we use Intel Optane Persistent Memory modules (see Section 2.4), which can be accessed through the same DAX mapping interface as CXL devices (see Section 2.5). While the performance characteristics of Optane and CXL devices are expected to be different, we can still see the impact of a kernel bypass through DAX mappings, and get an idea of how well our implementation performs for different scenarios.

The system used for evaluation is a server running Fedora Linux 42 (Server Edition) [61], with Linux kernel version `6.14.11-300.fc42.x86_64`. It has two `Intel(R) Xeon(R) Gold 5220 CPU @ 2.20GHz` CPUs with 18 cores each [27]; hyper-threading is disabled. The system has 384 GiB of RAM provided by 12×32 GiB DDR4 DIMMs. Persistent memory is provided by eight *Intel Optane Persistent Memory 100 Series* modules, each with a capacity of 128 GiB [28]. The Optane modules used for the evaluation are all connected to the same CPU and therefore share the same NUMA node [67].

Configuration	Description
`posix`	*CXLRaid* is not used; the default data path through the Linux kernel is used instead. The test files are still on an Optane module for a fair comparison.
`mapping`	*CXLRaid* is used, but in RAID 1 mode with a single drive. This effectively tests how `read` and `write` operations perform for Optane, if they internally use a DAX mapping, but without the influence of a RAID configuration.
`raid1`	RAID 1 (see Section 4.11.1) with three drives.
`raid1_journal`	like `raid1`, but with the journal enabled. The journal file is stored on an additional, fourth Optane module.
`raid1_broken`	like `raid1`, but the first drive is not mounted. This simulates running the RAID in degraded mode. At the same time this tests if reconstructing data from parity information does work.
`raid5`	RAID 5 (see Section 4.11.2) with three drives. If not stated otherwise, the stripe size is 4096 bytes, which is the default for *CXLRaid*.
`raid5_journal`	like `raid1_journal`, but with RAID 5 instead of RAID 1.
`raid5_broken`	like `raid1_broken`, but with RAID 5 instead of RAID 1.

Table 2: RAID configurations used for evaluation.

We run the tests multiple times with the different RAID configurations listed in Table 2. Each drive is formatted with the ext4 filesystem and mounted with the `-o dax=always` option to enable DAX [34]. The default journal is left enabled.

5.1 FIO

For a basic evaluation, we use the *Flexible I/O Tester* (FIO) [11]. Among other things, it allows us to measure the bandwidth and latency of various I/O operations. FIO offers a lot of configuration options that we can tune. If not stated otherwise, we use the following configuration for all following tests:

Configuration	Value	Description
`numjobs`	`1`	Write using a single thread.
`ioengine`	`psync`	Do synchronous I/O using `pread` [104] and `pwrite` [105] system calls.
`blocksize`	`4096`	Read and write data in blocks of 4096 bytes.
`runtime`	`30s`	Run the test for 30 seconds.
`ramp_time`	`10s`	Start measuring after 10 seconds. This is added to the runtime.
`size`	`5 GiB`	The maximum size of the test file used by FIO.
`sync`	`false`	Do not use synchronous I/O.
`direct`	`true`	Use direct I/O.
`numa_cpu_nodes`	`0`	Use CPUs from NUMA node 0.
`readwrite`	*varies*	Access pattern, e.g., sequential reads or random writes.

Table 3: FIO configuration options.

FIO utilizes so-called "engines" to perform the backend I/O operations. The `psync` engine is used to test the performance of our implementation. It issues synchronous I/O requests using the `pread` and `pwrite` system calls, which we overwrite as described in Section 4.1. We prefer it over the `sync` engine, which uses `read` and `write` system calls, since positioned access is the most direct I/O path in our implementation.

The runtime is set to 30 seconds to get a good average of the performance. We use a ramp time to give a fair comparison between our implementation, which is called through the `psync` engine, and the `libpmem` engine. The `libpmem` engine prepopulates the file with random data [12], while the `psync` engine does not [13]. This causes the kernel to allocate the blocks for the file in advance, making initial writes much faster. By using a ramp time, we mitigate this effect, since all blocks will already be allocated when the measurement starts.

The block size is set to 4096 bytes, which is the minimum for the `libpmem` engine [14], and equals the page size of our system. Different block sizes are evaluated in Section 5.1.5.

The size of the test file is chosen rather arbitrarily. If the maximum size is reached but the runtime is not over yet, FIO will simply loop back to the beginning of the file and start from there again. It is, however, large enough to ensure that Optane can ramp up to its maximum performance because of sequential access patterns.

The `sync` parameter is set to false, which is also the default behavior. In the `psync` engine, this means that the file will be opened without the `O_SYNC` [70] flag. In the `libpmem` engine, this means that the `PMEM_F_MEM_NODRAIN` flag is passed to `pmem_memcpy` [7], as we do in our implementation (see Section 4.9). The `direct` parameter is changed to true. In the `psync` engine, this causes the file to be opened with the `O_DIRECT` [70] flag, which is relevant for our `posix` configuration. This will instruct the kernel to bypass the page cache and operate directly on the underlying storage, similar to how our DAX mappings work. In the `libpmem` engine, this adds the `PMEM_F_MEM_NONTEMPORAL` flag to `pmem_memcpy` [7], which we also use in our implementation (see Section 4.9).
The combination of `PMEM_F_MEM_NODRAIN` and `PMEM_F_MEM_NONTEMPORAL` not only gives the best performance for Optane (see Figure 2), it also makes the `posix` configuration comparable to the mappings, as this way we will measure the actual performance of the underlying storage, and not the performance of the page cache.

Since our evaluation server has two CPUs, and therefore multiple NUMA nodes (see Section 2.3.1), we instruct FIO to use only CPUs from the first NUMA node. This is the NUMA node that the Optane modules are connected to, and therefore the ones that should yield the best performance.

In addition to the configurations listed in Table 2, we define an additional `libpmem` configuration for FIO benchmarks, which uses the `libpmem` FIO engine instead of `psync`. The rationale is similar to the `posix` configuration, as it does not use our RAID implementation, but instead directly writes to a single file on an Optane module. `libpmem` does use explicit DAX mappings, however, so performance should be equal to the `mapping` configuration, as it functionally does the same thing, just with our own implementation.

5.1.1 Sequential Writes

For the first test, we measure the bandwidth and latency of sequential writes (`readwrite=write`).

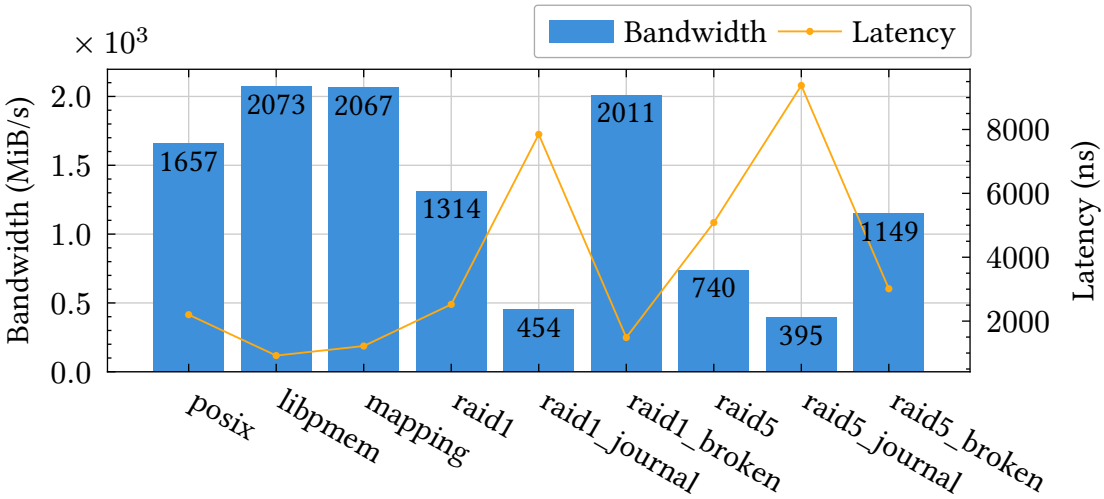

Figure 3: Sequential write performance of different RAID configurations. Using DAX gives a significant performance boost compared to the traditional POSIX I/O path. RAID configurations that have to perform more writes have lower bandwidth.

As Figure 3 shows, the `libpmem` and `mapping` configurations perform very similarly, giving us the highest bandwidth and lowest latency. They are also very close to the 2.14 GiB/s (2.3 GB/s) mark, which is the maximum bandwidth for a single-threaded direct write for our Optane modules [46]. The `posix` configuration performs worse, despite writing the same amount of data to the same underlying storage, showing the overhead of the kernel.

The `raid1_broken` configuration, which is effectively RAID 1 with two drives, performs only slightly worse than the `mapping` configuration, benefiting from the non-synchronous memory copies. The `raid1` configuration, however, which uses three drives, is notably slower. We probably hit the maximum amount of parallel memory copies that the CPU can handle here, which causes the performance to drop.

The RAID 5 configurations perform worse than their RAID 1 counterparts, which is expected since they have to do parity calculation and additional memory copies. The `raid5_broken` configuration performs better than the `raid5` configuration, as the benefit of having to write less data seems to outweigh the cost of the more complex parity calculation (see Section 4.11.2).

The journal also has a noticeable impact on the performance. Not only does it cause additional writes, but the journal is cleared between each block, which causes the higher latency.

5.1.2 Sequential Reads

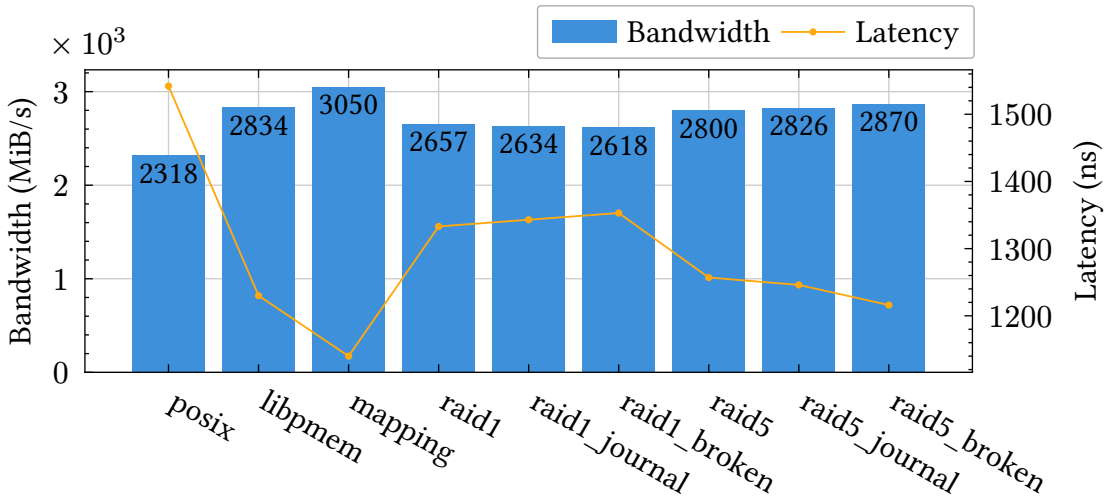

Figure 4: Sequential read performance of different RAID configurations. DAX mappings give a significant performance boost compared to the traditional POSIX I/O path. RAID 5 performs slightly better than RAID 1 because of higher locality.

We test the read performance similar to the write performance, just with `readwrite=read`. The results are shown in Figure 4.
Interestingly, the `libpmem` configuration performs worse than the `mapping` configuration for reads, despite doing the same thing. The performance of the `mapping` configuration is equal to the maximum read performance of our Optane modules, however [46], which is the expected value. The `posix` configuration is not only has a lower bandwidth, but has significantly higher latency, showing the overhead of the kernel for read operations.
All three RAID 1 configurations perform the same, which is expected since data is read only from a single drive in all cases. Which drive is used, however, rotates between each block, causing some minor performance differences.

The RAID 5 configurations perform better than the RAID 1 configuration, which is a bit surprising. We probably benefit from higher locality of the data here, since we read all data from all drives, in contrast to RAID 1 where the rotation between the drives causes more random access patterns. `raid5_broken` also needs to reconstruct data from parity information, but doing this from already read data seems to be ever so slightly faster than reading the data directly from the drive.

5.1.3 Random Writes

FIO also allows us to test random access patterns. First, we test random writes with `readwrite=randwrite`.

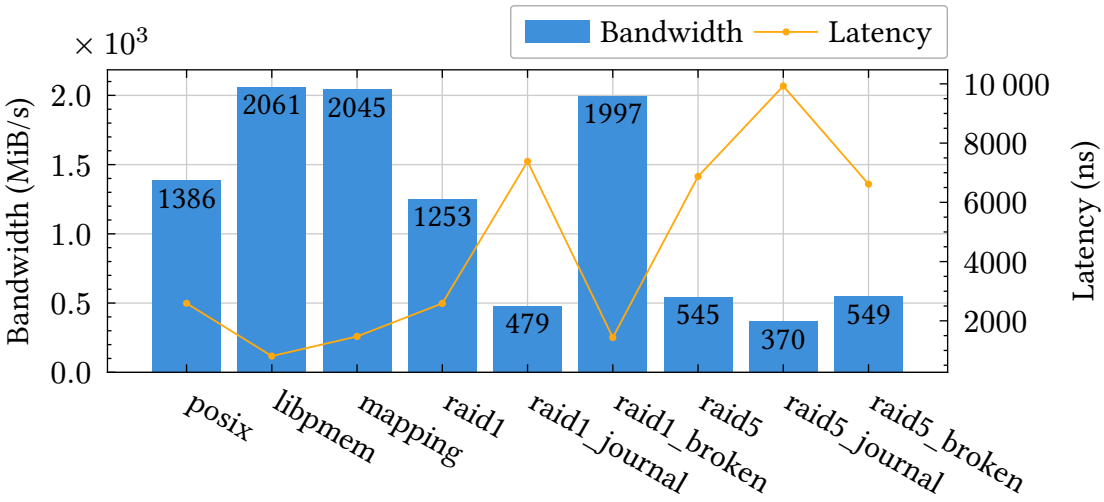

Figure 5: Random write performance of different RAID configurations. Proportionally similar to sequential writes, but generally lower bandwith.

Figure 5 shows the performance results for these random writes. Overall, they look similar to the sequential writes in Figure 3. While the `mapping` and `libpmem` configurations are still limited by the maximum bandwidth of the underlying storage, the `posix` configuration performs worse, making DAX mappings even more beneficial in comparison.

RAID 1 also performs similarly to the sequential writes; RAID 5, however, sees the biggest performance drop. The byte-wise parity calculation seems to suffer more from the random access pattern than the specialized memory copy functions in RAID 1 do.

This is especially noticeable for the `raid5_broken` configuration, which needs to do a more complex parity calculation (see Section 4.11.2), causing the bandwidth to be more than halved in comparison to the sequential writes.

5.1.4 Random Reads

Finally, we test random reads with `readwrite=randread`.

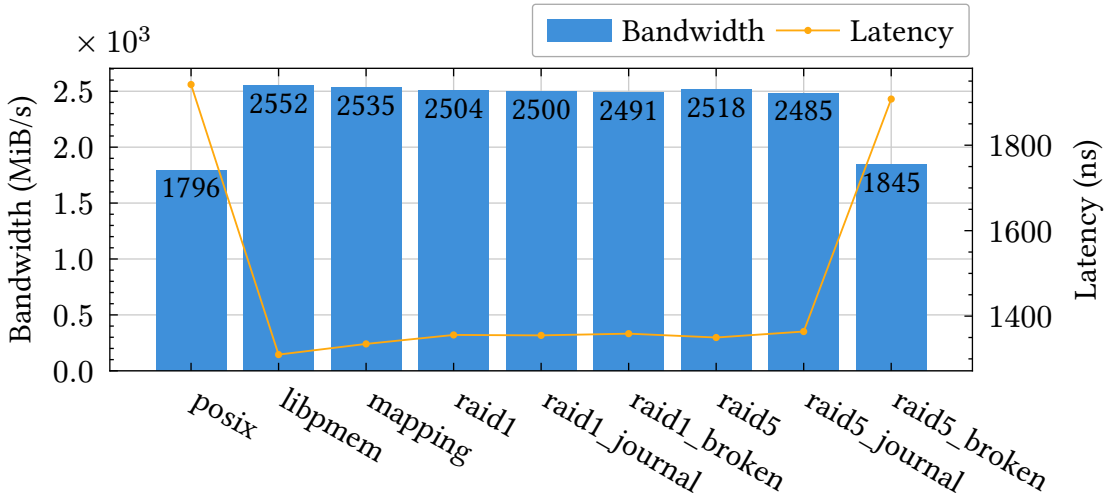

Figure 6: Random read performance of different RAID configurations. DAX gives higher bandwidth than POSIX I/O. RAID 5 in degraded mode performs worse than the other RAID configurations, because of parity reconstruction overhead.

Figure 6 shows that random reads perform equally well for almost all configurations, although slightly worse than the sequential reads in Figure 4. This makes sense, as all these levels directly copy the data from the underlying storage. And since the access is always random, we do not benefit from higher locality as some configurations do for sequential reads.

The `posix` configuration performs notably worse than the other configurations as well as itself for sequential reads, showing that the overhead of the kernel is even more noticeable for random reads.

The one configuration that cannot simply copy the data is `raid5_broken`, as it needs to reconstruct some data from parity information. As already observed for random writes, byte-wise random access is notably slower, causing a significant drop in performance for this configuration.

5.1.5 Impact of Block Size

The block size is the amount of data that is read or written in a single I/O operation. We do not test the `libpmem` configuration here, as it only supports block sizes of 4096 bytes and above, and our `mapping` configuration performs the same. We also only focus on sequential writes here, which should give us a good insight into how the block size affects the bandwidth. We test small values around the internal block size of Optane of 256 bytes [46], as well as 1 KiB and 4 KiB, which is also our stripe size and the page size, and some larger values up to 1 MiB to see how the performance changes for large block sizes. Note that because of these selectively chosen block sizes, the x-axis is not linear here.

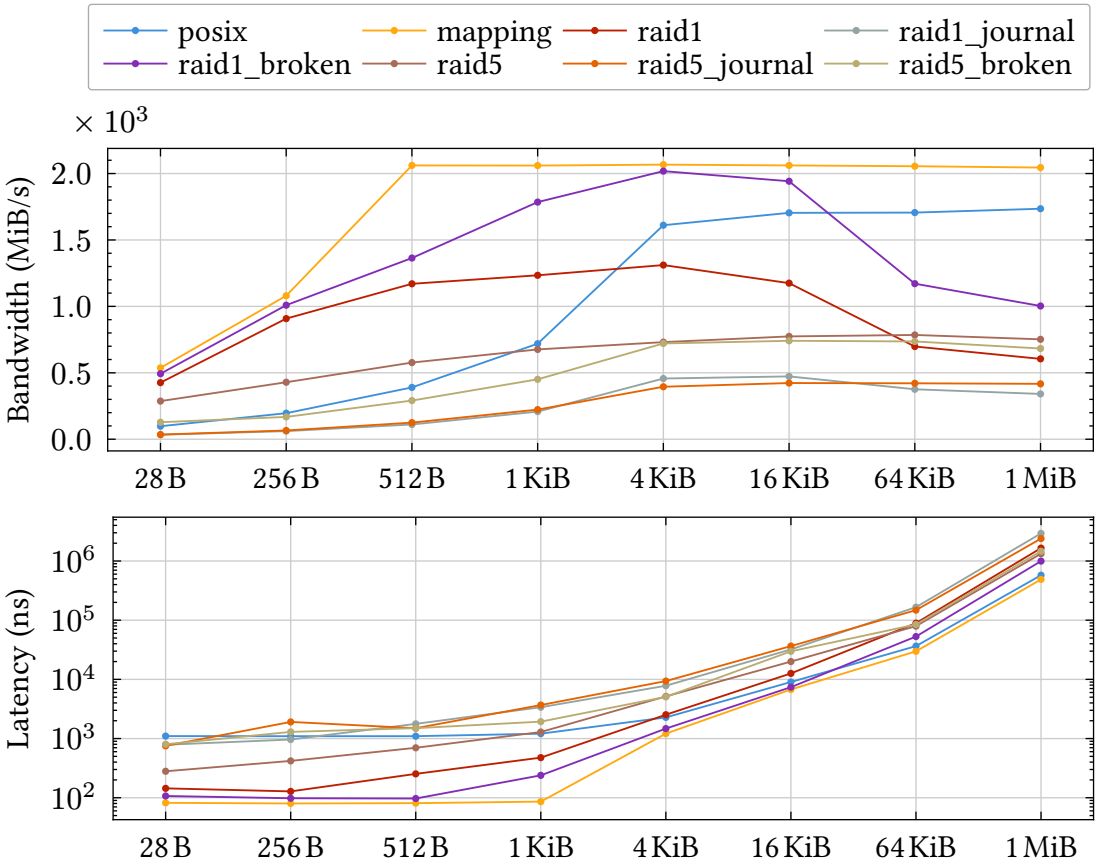

Figure 7: Impact of different block size on sequential write performance. Highest bandwidth is achieved with block sizes around 4 KiB, which equals the page size.

As shown in Figure 7, a greater block size gives a higher bandwidth up until around 4 KiB, after which the performance slightly drops again. The exception to that is the traditional `posix` I/O path, which strictly benefits from larger block sizes, especially once they reach the page size. The biggest drop in performance after 4 KiB can be observed for the RAID 1 configurations. Higher block sizes seem to limit the amount of parallelism the CPU can achieve for the memory copies. The RAID 5 configurations, on the other hand, are quite stable, as they never copy more than the stripe size at once, which is also 4 KiB in this case.
The latency strictly increases with larger block sizes, which is expected since more data needs to be copied. Note that the y-axis is logarithmic here to make differences for smaller block sizes more visible.

5.1.6 Impact of Stripe Size

The stripe size is the amount of data that is written to a single drive before switching to the next drive in RAID levels that do striping, such as RAID 5 (see Section 2.1.2). *CXLRaid* allows for it to be configured freely, with the default being 4096 bytes. In this section, we evaluate how different stripe sizes affect the write performance.

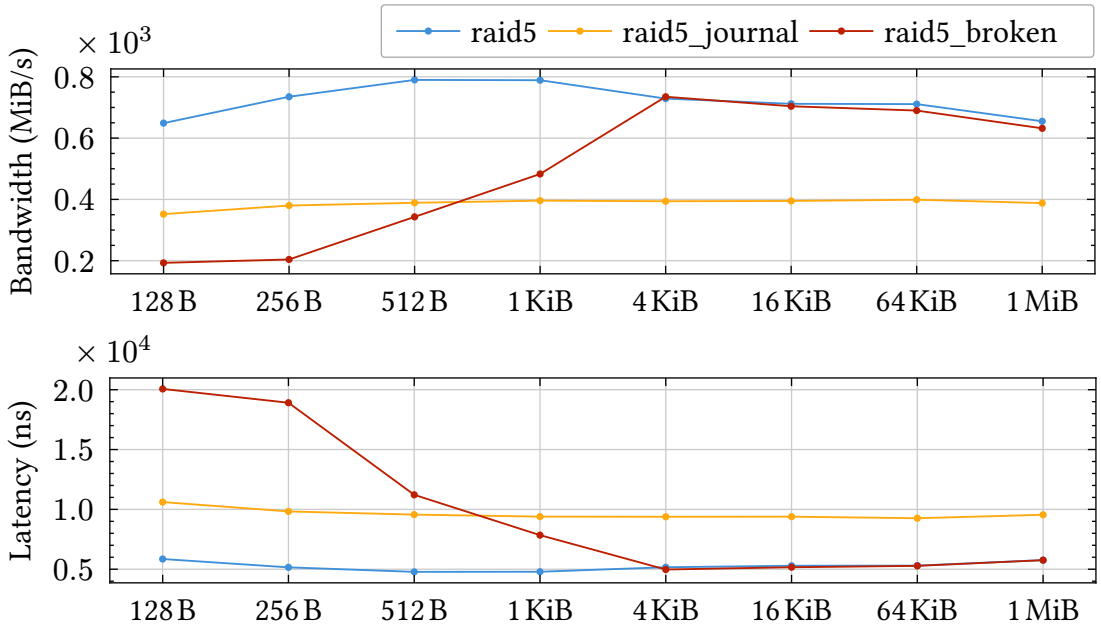

Figure 8: Impact of different stripe sizes on RAID 5 write performance. Stripe sizes around 512 bytes give the highest bandwidth. RAID 5 in degraded mode is significantly worse when the stripe size is smaller than the block size.

As seen in Figure 8, *CXLRaid* normal modes benefit slightly from stripe sizes between around 256 bytes and 4 KiB. The main impact is on the `raid5_broken` configuration, as long as it is smaller than the block size (4 KiB in this case). Parity calculation is currently done per chunk, so multiple times per stripe if the block size is larger than a chunk. This is only a small overhead for the `raid5` and `raid5_journal` configurations, as they do not need to do a full parity calculation for each chunk, but leads to the observed performance drop for `raid5_broken`. This can be improved in the future by using a more sophisticated decision policy on when to do parity calculations. In Section 6.1.3 we discuss this in more detail what would need to be done for that. For now we recommend to not use a stripe size that is smaller than the expected average block size. Note, however, that a larger stripe size also leads to higher space consumption, as each backing file size needs to be a multiple of the stripe size.

5.2 Valkey

For a more realistic evaluation, we test the performance of the key-value datastore Valkey [132] using different workloads from the *Yahoo! Cloud Serving Benchmark* (YCSB) [45]. Each workload is run against each of the different configurations and averaged over five runs. Valkey is run with its *append-only file* (AOF) persistence mode [8], which works by appending all updates to a log file, which can be replayed to restore the state. We also configure `appendfsync=always`, which instructs Valkey to call `fsync` [117] after each write to the log file, to ensure maximum persistence guarantees. In *CXLRaid*, `fsync` is implemented as a no-op, as data will always be persisted. It does make a difference for the `posix` configuration, however, as otherwise we would mainly be writing to the page cache, which would give us a false performance boost.

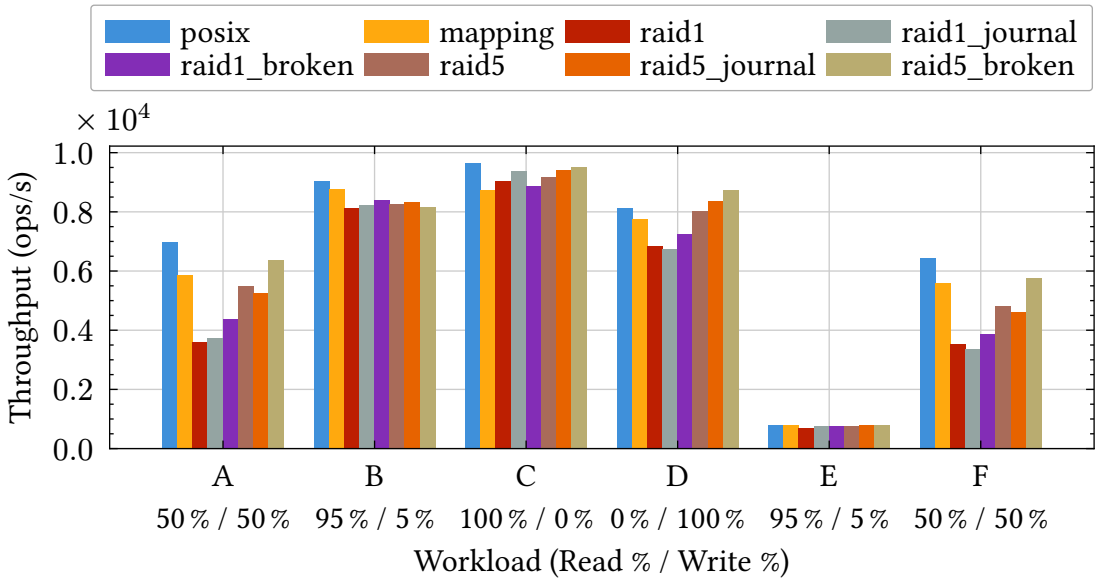

Figure 9: Valkey throughput performance for different YCSB workloads. Posix outperforms *CXLRaid* especially in write heavy workloads.

Figure 9 shows the throughput performance of Valkey for different YCSB workloads and *CXLRaid* configurations. Below each workflow the read and write ratio of the workload is shown. Notably, *CXLRaid* performs worse than the `posix` configuration, especially for workflows with a higher write ratio. We suspect that this is due to the fact that an append-only file is the worst-case scenario for our implementation, as we constantly need to resize the backing files, and therefore create a new DAX mapping over them.

5.3 md/raid

md/raid [65] is the software RAID implementation in the Linux kernel. We already introduced it in Section 2.2; now we compare it to our implementation. For that, we build a RAID 1 and RAID 5 array with `mdadm` [66], each getting a partition of 128 GiB on three SSDs. The SSDs are *Micron Technology Inc 7300 PRO NVMe SSD* [57]. We set up two RAID 5 arrays, one with the default stripe size of 512 KiB [65], and one with a stripe size of 4 KiB, which is the same as in our implementation. Neither of them uses a journal, as we do not have a fourth identical SSD at hand for that. For shorter identification, we call them `mdraid1`, `mdraid5_4k`, and `mdraid5_512k`. In addition to

the RAID configurations, we also test a single partition on one of the SSDs, which we call `partition`. This allows us to get an insight into the performance of the SSD itself and compare the RAID configurations to that. The file system on them is ext4, created with the following commands:

- `partition`: `mkfs.ext4 /dev/nvme1n1p4`
- `mdraid1`: `mkfs.ext4 /dev/md0`
- `mdraid5_4k`: `mkfs.ext4 -E stride=1,stripe-width=2 /dev/md1`
- `mdraid5_512k`: `mkfs.ext4 -E stride=128,stripe-width=256 /dev/md2`

The `stride` and `stripe-width` options for the RAID 5 arrays are set to match the stripe size, as recommended by the `mkfs.ext4` manual [118].

We test the performance of these configurations with FIO, using the same parameters as described in Section 5.1. Most notably, we write blocks of 4096 bytes, and `direct=1` is used to bypass the kernel page cache.

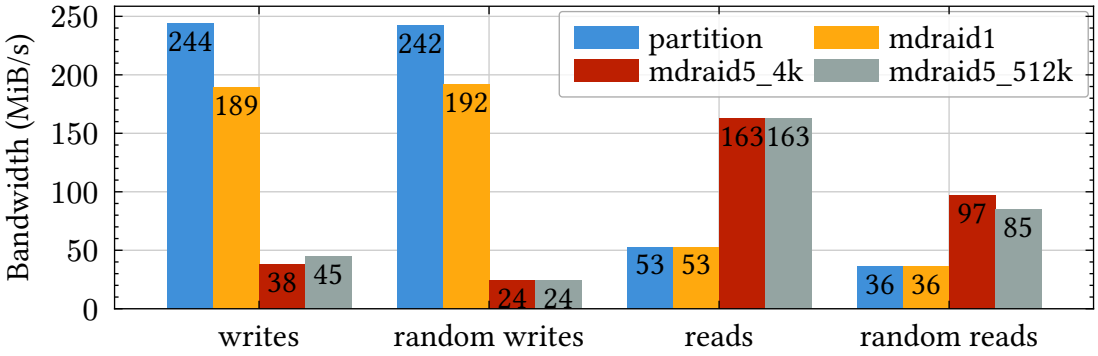

Figure 10: Bandwidth of different md/raid configurations. RAID 1 gives good write performance, while RAID 5 gives good read performance. Random access patterns perform worse than sequential access patterns for all reads and RAID 5 writes.

As we can see in Figure 10, the `partition` and RAID 1 configurations perform quite similarly, as do the two RAID 5 configurations. While the former provide better read performance, the latter provide better write performance.
RAID 5 being slower for writes is expected, as RAID 1 writes the data in parallel to all drives, while RAID 5 first needs to read the old data and parity, calculate the new parity information, and then write the new data and parity information [65]. The higher read performance of RAID 5 is also expected, as md/raid implements parallel reads for RAID 5, while for RAID 1 it only reads from one drive [65].

5.3.1 Performance Comparison

In this section, we compare these results to the performance of *CXLRaid*. Because Optane has a higher bandwidth than the SSDs we use for md/raid, we look at the factor of improvement instead of absolute values. For that we compare the `posix` configuration to the `partition` configuration (we call that `direct`), the `raid1` configuration to the `mdraid1` configuration, and `raid5` to both RAID 5 configurations of md/raid.

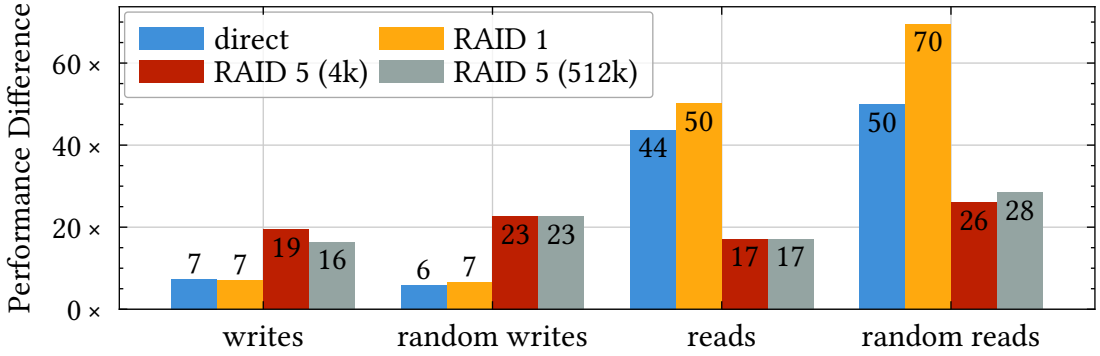

Figure 11: Performance improvement of *CXLRaid* over md/raid. `direct` equals the performance gain by using persisten memory without DAX mappings. *CXLRaid* performs better for RAID 1 reads and RAID 5 writes, while being on par for RAID 1 writes and worse for RAID 5 reads.

In Figure 11, we can see the performance improvement of *CXLRaid* over md/raid. The `direct` value is interesting for us, as it shows how many times one Optane module is faster than one SSD for the given workload. A higher value for the same access pattern implies that our RAID implementation, using DAX mappings, performs better than md/raid, which uses the kernel I/O path.

If we look at RAID 1, we see that our performance improvements for writes are in line with the faster underlying storage. Keep in mind, however, that md/raid issues parallel writes to all drives [65], while *CXLRaid* does not. *CXLRaid* fully saturates the amount of parallelism the CPU can achieve for our non-synchronous memory copies when using three drives (see Section 5.1.1), so we are bound by our sequential implementation here. For reads, however, both implementations copy the data from a single drive, and our approach beats md/raid here by up to 1.4×, especially for random reads.

For RAID 5, we have a clear performance improvement of up to 3.8× for writes, independent of the stripe size used for md/raid. Reads, on the other hand, are slower, but here too md/raid implements parallelism, while *CXLRaid* does not. Overall, *CXLRaid* gives a performance improvement over md/raid for most cases, even when both would use the same underlying storage, which shows the benefits of our DAX mappings approach.

5.3.2 Feature Comparison

While performance is an important aspect of RAID implementations, we also need to assess the features and guarantees they provide. Most notably, md/raid is a much more mature and widely used implementation, with more options for the user to choose from. This mainly means more RAID levels, including some pseudo ones (like `MULTIPATH` [65], for example, one physical drive connected via multiple paths). The standard ones can be implemented in *CXLRaid* too, however, if needed in the future.
Being implemented in the kernel, md/raid also has some advantages that *CXLRaid* cannot easily provide. First of all, it is unaware of how it is used, especially if there is a filesystem on top of it. This allows using md/raid not only with a filesystem, but also as a raw block device. Example use cases for that include using it as a backing device for a virtual machine [131] or as a direct storage device for a Docker volume [52].
It also has the advantage of integrating directly with the kernel's device manager, which allows it to better detect and handle drive failures while running. *CXLRaid*, on the other hand, is likely to crash, as it will try to write into a now faulty DAX mapping. Implementing a check for device failure before each write operation is possible, but would not only add an overhead, but also defeat the idea of bypassing the kernel for I/O operations.
In terms of data integrity, both solutions provide the guarantee that the RAID is never in an inconsistent state after a system failure, as they both have the option to use a journal.
md/raid also implements an optimization called *Write Intent Bitmap* [65], which keeps track of which blocks are currently being written to. In the case of a drive being disconnected and then reconnected, md/raid can check the bitmap to only resync the blocks that were being written to at the time of the failure, instead of the whole drive. This is especially beneficial for larger drives, as it can significantly reduce the resync time. *CXLRaid*'s CLI has a `repair` command (see Section 4.14), which iterates over each file on each drive to check for inconsistencies, which, even with few files,

can take a long time. A similar "dirty" approach can be implemented analogously to how the journal works, if desired in the future.
An advantage that *CXLRaid* has over md/raid is the more fine-grained control over which files are part of the RAID. The configuration is not only per process, but files of the same process can be selectively chosen based on their path. This allows putting only important data on the RAID while leaving other data on the normal storage, even when the application does not support that natively. md/raid, on the other hand, is a block device, so all data on it is always part of the RAID.

5.4 Discussion

In Section 5.1, we measured that using DAX mappings gives us a significant performance boost compared to the traditional kernel I/O path, even when used transparently through the same interface. This makes them interesting not only for RAID specifically, but also for the general I/O operations of an application. However, we also saw in Section 5.2 that the fixed size nature of memory mappings can lead to performance issues, when the workload primarly consists of appends, for example a log file. If an application benefits from them should therefore be evaluated on a case-by-case basis.
If a workload benefits from DAX mappings, a RAID implementation must be implemented on top of them, as we discussed in Section 4.1. The implication of that is however, that we need to implement a lot of functionality in user space, that is usually done by the kernel. This is mainly due to the fact that we have to maintain our own filetable, as we explained in Section 4.4. We argue therefore that a hybrid approach, where the kernel is involved in the RAID implementation, is desirable. In Section 6.2 we discuss how such a hybrid approach can look like, and what benefits it can provide.
What we unfortunately could not show in our evaluation is that *CXLRaid* also brings a performance boost for real-world applications. As the only benchmark that mixes reads and writes and does not have specific fabricated access patterns, we tested Valkey in Section 5.2. However, it performed worse than expected, presumably due to the append-only nature of its log file. Other databases that we tested, MariaDB [29] and PostgreSQL [40], did not run through *CXLRaid* because of reasons listed in Section 6, especially the `exec` issue (see Section 6.1.4).

We expect that the approach of *CXLRaid* can give a performance boost for most real-world applications, as it is the case in the micro benchmarks, but that remains to be shown in future work.

Chapter 6

Future Work

During the course of this thesis, we highlighted several areas where our current implementation could be improved or extended. In this chapter we will list the most important ones and discuss what would be required to implement them. After that, we propose a design for a hybrid user/kernel-level implementation, which could solve some of the issues we encountered with our userspace-only approach, while still keeping the benefits of DAX mappings.

6.1 Potential Improvements

In the following sections we list possible enhancements to the current implementation of *CXLRaid* that can be selectively implemented in the future if required.

6.1.1 Multithreaded I/O Support

In Section 5.3 we noticed that the Linux software RAID implementation, md/raid, uses multiple threads to parallelize performance-critical operations, precisely RAID 1 writing and RAID 5 reading. This is something that *CXLRaid* would probably benefit from as well, especially for reading. While we are already close to maxing out the write bandwidth of Optane with a single thread, using multiple threads for reading can more than double the throughput [46]. Implementing this would probably involve using a thread pool with a work queue, as creating and destroying threads is a costly operation that we do not want to do for every I/O request [54]. Given this, the read loop of RAID 5, for example, can be parallelized trivially, as one iteration of the loop does not depend on the result of the previous iteration. The optimal thread pool size is something that should be determined experimentally. We

suspect that it is around 3 to 5 times the number of drives in the RAID array, as that is where Optane's read performance maxes out [46]. An advanced implementation could even dynamically adjust the thread pool size based on the latency of the I/O requests.
When switching to CXL persistent storage as the primary storage medium, the performance analysis required for assessing multithreading will have to be redone. While we expect a performance benefit from bypassing the kernel, the benefits of multithreading are likely to be more dependent on the hardware itself than the way it is accessed. We expect that multiple threads will be beneficial as well, as it is for md/raid using NVMe SSDs through the block-based interface (see Section 5.3).

6.1.2 Multithreaded Application Support

While *CXLRaid* can benefit from using multiple threads, so can applications using *CXLRaid*. This, however, is problematic, as *CXLRaid* is currently not thread-safe. With unfortunate timing, for example, two threads can get the same FD for different resources, as allocating and reserving one is not an atomic operation. Linux also guarantees that some I/O operations are atomic; for example, two `write` calls will not interleave their data, correctly share the file offset, and so on [64]. It is to be expected that applications rely on these guarantees and might break if run through the current implementation of *CXLRaid*. Supporting multithreaded applications would require synchronization primitives at most entry points (see Section A.1), as almost all of them access shared data structures. The challenge here is to find synchronization rules that do provide correctness but do not lock up the entire library when an operation is in progress.

6.1.3 RAID 5 Parity Calculation

In the current implementation of RAID 5 writes, the data is split up into chunks aligned to the stripe size. For each chunk, we first calculate and write the new parity, and then write the chunk itself. This is required for the default way of calculating parity, as it uses both old data and old parity (see Section 4.11.2). However, when we run in degraded mode, that is, with one drive missing, we need to calculate the parity by `XOR`ing the whole stripe. If the given data now spans multiple chunks in the same stripe, we read data that we just wrote, causing an overhead as the CPU has to wait for the data to be written before it can be read again. It is also not in the CPU cache, as we explicitly bypass it for writes (see Section 4.9). The effect of this can be observed in Section 5.1.6, where degraded RAID 5 has performance issues when the stripe

size is smaller than the block size. We can solve this by calculating the parity for the whole stripe at once, and getting data of previous chunks not from the mappings, but from the given buffer. But since this is only a performance optimization for the specific situation of having write sizes larger than the stripe size in degraded RAID 5, we have not implemented it yet.

6.1.4 exec Support

On Linux, the `exec` family of functions replaces the current process image with a new one, effectively running a new program [84]. Not everything is replaced, though, as file descriptors stay open as long as they are not explicitly marked with the `FD_CLOEXEC` flag [83]. Applications can use this to share resources with child processes. A notable example of that are pipes in `bash` [62] scripts. The FD number can be passed to the child process through an environment variable, command-line argument, or some other form of inter-process communication. We currently do not support this in *CXLRaid*, as we initialize a fresh file table for each process. A possible implementation would need to find a way to serialize the file table so that it can be shared between processes. We could then overwrite the `exec` functions to close all mappings and the FDs that are marked with `FD_CLOEXEC`, perform the serialization, and then call the original `exec` function with an extra command-line argument describing where to find the serialized file table. Depending on the size and form of the serialization, this might be in the argument itself, in a temporary file, a Unix pipe [119], or similar. In the initialization of *CXLRaid*, we would then look for this argument, and if it is present, deserialize the file table and set up DAX mappings for all the FDs that already have open Linux FDs. Afterwards, we remove the argument from the command line, so nothing changes from the perspective of the application. While we technically close and reopen the mappings instead of keeping them open, this should be transparent to the application.

A hybrid implementation including the kernel (see Section 6.2) can help, as we would not have to serialize the file table. We would still need a way, however, to tell the library to create mappings for already open FDs, as otherwise reads and writes would not work.

We suspect that when a process is run through *CXLRaid*, this is done explicitly by setting the `LD_PRELOAD` environment variable (see Section 4.2). In this case, we explicitly do not support passing FDs to child processes and expect the user of *CXLRaid* to be aware of this.

There is no real way of implementing sharing between RAID and non-RAID processes, as the first one would use the backing files while the second one would not, making the same path point to two different resources.

6.1.5 fork Support

Similar to `exec`, the `fork` [120] system call creates a new process, but instead of replacing the current process image, it splits the current one into two, with the child process being an almost exact copy of the parent process. This is problematic for *CXLRaid*, as we now have two processes whose FDs should point to the same `struct file` (see Section 4.4), actively sharing things like the open file offset. We cannot use an approach similar to `exec`, as both processes might modify the file table after the `fork`, requiring us to keep them in sync. We can achieve this by putting shared parts of the file table (`struct file`, `struct inode`) into shared memory, which does not get copied during the fork [121]. The other part of the file table (`struct fd`) stays in the private memory of the process, so it is properly copied. This would require synchronization similar to the one required for multithreaded application support (see Section 6.1.2), as both processes can modify the shared data at the same time. Setting up the shared memory segment would be done by the first process run through *CXLRaid*; after that, a reference counter in the shared memory segment can be used to close it in the destructor of the last process using it. If implemented, this could also be used complementarily to the `exec` support (see Section 6.1.4). Instead of serializing the whole file table, one would only need to serialize the FD table and the location of the shared memory segment.

6.1.6 Hard Link Support

Linux filesystems support hard links, which are multiple directory entries pointing to the same inode [122]. This effectively creates two files that share the same content, and changes to one of them are reflected in the other one. This is problematic for *CXLRaid*, as we decide using path prefixes whether a file is part of the RAID or not (see Section 4.5). In contrast to symbolic links, there is no "real" path that we can resolve to check, as both paths are equally valid. There are three cases which we have to consider:

- Both paths are not part of the RAID: This has a simple solution, we just have to create the link on the normal filesystem.
- Both paths are part of the RAID: We can create a link on each drive to keep the filesystem structure intact. However, we use paths to identify inodes (see

Section 4.4), so openings of both paths would not correctly share resources. The file table would need to be changed to resolve inodes by an independent identifier, with a way to map paths to these identifiers. This would make the implementation considerably more complex, so we argue that this is not worth supporting. An implementation involving the kernel, as described in Section 6.2, would help here too.
- One path is part of the RAID, while the other one is not: We would need to create the file both in the RAID and on the normal filesystem, and keep them manually in sync, which is likely to be error-prone. The simpler solution would be to just disallow this, similar to how hard links across filesystems are not possible [122].

Overall, we argue that supporting hard links inside the RAID with the current design is not worth the effort. This is however something that becomes more feasible when involving the kernel (see Section 6.2), as it already has support for hard links we can reuse.

6.1.7 Broader Function Support

While we already support a wide range of functions (see Section A.1), there are still ways to circumvent *CXLRaid*. Here is a list of known gaps in our implementation, and how they could be filled:
- `syscall` function [76]: Allows applications to call system calls using their syscall number, rather than their wrapper function. Since this is a function itself, we can overwrite it as well. Inside it we would need to check for every syscall number that corresponds to a function we support, and redirect it either to our implementation or the original one.
- `dlsym` [79]: Allows applications to get the address of a function by its name, and then call it directly. This is problematic for us, as it could be used to load functions explicitly from the C standard library, bypassing our hooks. In fact, this is exactly what we do to call original functions (see Section 4.3.1). One could overwrite `dlsym` as well, and dispatch calls similar to `syscall` if an application does this.
- `acl_*` [123]: A set of functions for manipulating *Access Control Lists* (ACLs) on files. Currently not implemented, as no program we tested uses them and there are *a lot* of these methods. An implementation would probably be similar to other metadata, as we described in Section 4.7.
- `SYSCALL` instruction [19]: A program could also directly embed the `SYSCALL` instruction in its assembly, which would bypass our `LD_PRELOAD`-based hooks

(see Section 4.3). One way to support this would be to use binary rewriting, which does bring its own set of challenges [136]. Another way would be to use the kernel for system call hooking, which is not only slow but defeats the purpose of bypassing the kernel in the first place [35, 73]. An implementation involving the kernel (see Section 6.2) would help for functions that do not operate on file content. Other functions could be implemented in the kernel and therefore be called too, but would loose the benefit of bypassing the kernel.

- `aio_*` [124]: A set of functions for asynchronous I/O. They are not implemented currently, but doing so should be possible to implement using threads and the standard `read` and `write` functions as backend. Using `pthread_kill` [125], we can correctly send the `SI_ASYNCIO` [126] signal to the calling thread when the I/O operation is complete.
- `io_uring` [127]: A newer interface for asynchronous I/O, using a submission and completion queue [127]. Can be implemented similarly to `aio`, but without the need for signals.

6.1.8 Better mmap Support

The current implementation of `mmap` [69], as described in Section 4.13, is not optimal, as it requires complicated synchronization between multiple threads and can often lock the main thread, causing performance issues. Better solutions to implementing support for mmap() remain a topic to be explored.

6.1.9 More RAID Levels

We currently only support RAID 1 and RAID 5 (see Section 4.1). In Section 4.11 we explained how more standard RAID levels could be implemented if required. Additional support for "hybrid" RAID levels, such as RAID 10, a combination of RAID 1 and RAID 0 [65], can also be implemented. This would require some reorganization of the code, though, so that code between different RAID levels can be shared more easily.

6.2 Hybrid Implementation

Our current design, as described in Section 4, is a userspace-only implementation. That does bring its challenges, as can be seen by the amount of non-RAID-specific code we had to implement, as well as the things we listed for future work in Section 6.1. Most of these issues stem from the fact that we have to reimplement

a lot of the functionality of the kernel, such as file descriptor management (see Section 4.4), leading to us often working against the kernel instead of with it. In this section we propose a design that involves kernel code to show what a more complete solution could look like.

The main idea is to use the file table of the kernel, as this makes a lot of things easier; for example, our shims (see Section 4.4.1) would no longer be needed. This requires the kernel to support multiple inodes for a single file descriptor, which we would have to implement. In order to keep the kernel side as simple and generic as possible, we propose adding an API for this, but do not make it aware of RAID levels or similar concepts. RAID logic, like parity calculation, would still be implemented in a userspace library, which works quite similarly to the current implementation of *CXLRaid*. However, the only functions the library wouuld need to overwrite are the ones that interact with the DAX mappings, such as `open` [70], `read` [63], `write` [64] and `close` [71]. The other functions would get an in-kernel implementation similar to the respective userspace implementation we currently have. This would works without the kernel having awareness of RAID specifics, as these functions do not operate on the content of the files. The exception to this is the `ftruncate` [106] function, which we would implement both in the kernel and overwrite in userspace. The kernel side would resize all registered inodes, while the userspace side would update the file content and resize the mappings if required.

For configuration, we could let the kernel read the same environment variables as the userspace library. Knowing about the mounts would allow it to correctly handle system calls that work purely on paths, such as `stat` [102] or `access` [100].

We would need one new system call for this proposal, which we name `raid_open`. Similar to, for example, `shm_open` [121], it allocates a new file descriptor, with functions used on it being redirected to our own specialized functions in the kernel. Internally, we replace the `struct file` [38] with a custom structure that can handle multiple `struct inodes` [39], but does not have a current offset or similar. We keep the default `struct fd` [37], as this is just a pointer to our struct. We do not support `read`, `write`, or similar on the resulting FD, as those are implemented in the userspace library and the content of the files is allowed to be different. If a `mmap` is now called on such a file descriptor, it returns not a pointer to a mapping, but instead an array of pointers, one for each backing file. The userspace library can then use these pointers to implement the RAID logic exactly like we already did in Section 4.11.

Such an implementation would solve a lot of our future work in Section 6.1

automatically; for example, we would no longer need to worry about the file table not synchronizing across `fork` [120] calls. We could also use the synchronization primitives the kernel already provides to support multithreaded applications without having to do much work ourselves. Having control over page fault handling, the scheduler, and similar mechanisms is also promising to enable a better implementation of `mmap` [69].

Chapter 7

Conclusion

The emergence of CXL-based persistent storage has led to the development of Transparent DAX Mappings [50], which allow applications to take advantage of CXL's features without requiring modifications. However, supporting traditional storage features such as RAID in this context is challenging due to the desire to leave the kernel out of the data path. In this thesis, we presented *CXLRaid*, a userspace-only implementation of RAID 1 and RAID 5 that closes this gap.

We implemented *CXLRaid* as a dynamically linked library that uses `LD_PRELOAD` to transparently overwrite most of the filesystem-related functions offered by the C standard library. Within it, we programmed everything required to provide I/O without involving the kernel whenever possible, including a custom file table, path resolution, and most importantly the RAID logic itself. By using multiple backing files with a DAX mapping for each of them, we were able to implement RAID on a file level. The overwritten `read` and `write` functions use these mappings to provide RAID functionality transparently to the application.

In our evaluation, we had to use Intel Optane persistent memory as backing storage, as CXL-based storage was not available to us. We were, however, able to show that DAX mappings operate at the bandwidth limit of the underlying storage, while the data path through the kernel does not. We were also able to show that *CXLRaid* can take advantage of this and provide a competitive RAID implementation. It outperforms the Linux kernel's RAID implementation md/raid for RAID 5 writes by more than three times and RAID 1 reads by almost 1.5 times. RAID 1 writes are on par with md/raid, while RAID 5 reads are slower. Both of these cases can be explained by the fact that md/raid uses multithreading for these operations, while *CXLRaid* does not, suggesting that an approach like *CXLRaid* provides better scalability.

We conclude that *CXLRaid* is a viable approach for our research question, and TDMs in general are worth further research and development. In this regard, we also outlined some possible future work, including proposing a RAID implementation that keeps using TDMs for data transfer but moves other logic into the kernel for closer integration.

Appendix

A.1 Exported functions

A.1.1 Implemented Posix functions:

- `access`, `faccessat`, `euidaccess`, `eaccess`
- `chdir`, `fchdir`
- `chmod`, `fchmod`, `fchmodat`, `lchmod`
- `chown`, `fchown`, `fchownat`, `lchown`
- `close`, `close_range`, `closefrom`
- `closedir`
- `copy_file_range`
- `dirfd`,
- `dup`, `dup2`, `dup3`,
- `flock`,
- `fsync`, `fdatasync`, `syncfs`
- `getcwd`, `get_current_dir_name`
- `getdents64`
- `getdirentries`, `getdirentries64`
- `getdtablesize`
- `link`, `linkat`
- `lockf`, `lockf64`
- `lseek`, `lseek64`
- `mmap`, `mmap64`, `mremap`, `msync`, `munmap`
- `mkdir`, `mkdirat`
- `mkfifo`, `mkfifoat`
- `mknod`, `mknodat`
- `open`, `open64`, `openat`, `openat64`, `creat`, `creat64`,
- `opendir`, `opendirat`, `fdopendir`
- `pathconf`, `fpathconf`
- `posix_fadvise`, `posix_fadvise64`
- `posix_fallocate`, `posix_fallocate64`, `fallocate`, `fallocate64`
- `read`, `pread`, `pread64`

- readlink, readlinkat
- readv, preadv, preadv64, preadv2, preadv64v2
- realpath, canonicalize_file_name
- rename, renameat, renameat2
- scandir, scandir64, scandirat, scandirat64
- stat, stat64, fstat, fstat64, fstatat, fstatat64, lstat, lstat64
- statfs, statfs64, fstatfs, fstatfs64
- statvfs, statvfs64, fstatvfs, fstatvfs64
- statx
- symlink, symlinkat
- truncate, truncate64, ftruncate, ftruncate64
- unlink, unlinkat, rmdir, remove
- write, pwrite, pwrite64
- writev, pwritev, pwritev64, pwritev2, pwritev64v2

A.1.2 Implemented stdio functions

- fopen, fopen64, freopen, freopen64, fdopen
- fileno, fileno_unlocked,
- dprintf, vdprintf

A.1.3 Partial implemented / broken functions

- fcntl, fcntl64 (only F_{GET:SET}_FD, F_{GET:SET}_FL, F_DUPFD{:_CLOEXEC}, F_{GET:SET}LK{:W})
- ioctl (simply calls original ioctl for each backing file, this is probably wrong most of the time)

A.1.4 Shims

- accept, accept4
- epoll_create, epoll_create1
- eventfd,
- fanotify_init
- inotify_init, inotify_init1
- memfd_create
- name_to_handle_at, open_by_handle_at
- pipe, pipe2
- shm_open,

- `signalfd`,
- `socket`, `socketpair`
- `timerfd_create`

A.2 Example CLI info output

```
$ ./cxlraid info
Status Information
└── Journal Status: Clean
Configuration Information
├── RAID Level: 1
├── Configured Mounts: 3
├── Configured Paths: 2
├── Journal: /home/ldierheimer/bachelorarbeit/raid/mnt/pmem1/.cxlraid_journal
└── Stripe Size: Not set, probably because of RAID Level
Runtime Information
└── Page Size: 4096 bytes
Build Information
├── CXLRaid Version: 1.0.0
├── Log Level: 2
├── Build Type: Debug
├── File Header Version: 1
└── Journal Header Version: 1
Mounts Information
├── /home/ldierheimer/bachelorarbeit/raid/mnt/pmem1: WORKING
├── /home/ldierheimer/bachelorarbeit/raid/mnt/pmem3: WORKING
└── /home/ldierheimer/bachelorarbeit/raid/mnt/pmem5: WORKING
Paths Information
├── /home/
└── /root/
```

Bibliography

[1] 2008. *DDR SDRAM Standard*. Retrieved from https://www.jedec.org/system/files/docs/JESD79F_0.pdf

[2] 2020. *Serial ATA Revision 3.5*. Retrieved from https://sata-io.org/system/files/specifications/SerialATA_Revision_3_5_Gold.pdf

[3] 2025. *NVM Express Base Specification*. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-Revision-2.3-2025.08.01-Ratified.pdf

[4] Linux Standard Base Core Specification 3.2. 11.3. Special Sections. Retrieved from https://refspecs.linuxbase.org/LSB_3.2.0/LSB-Core-generic/LSB-Core-generic/specialsections.html

[5] Cory Altheide and Harlan Carvey. 2011. Chapter 3 - Disk and File System Analysis. *Digital Forensics with Open Source Tools*, 39–67. Retrieved from https://www.sciencedirect.com/topics/computer-science/redundant-array-of-inexpensive-disk

[6] The Clang Compiler authors. Attributes in Clang. Retrieved from https://clang.llvm.org/docs/AttributeReference.html

[7] The PMDK authors. pmem_memmove_persist(3). Retrieved from https://github.com/pmem/pmdk/blob/master/doc/libpmem/pmem_memmove_persist.3.md

[8] Valkey authors. Documentation: Persistence. Retrieved from https://valkey.io/topics/persistence/

[9] glibc authors. cookie_io_functions_t.h. Retrieved from https://sourceware.org/git/?p=glibc.git;a=blob;f=libio/bits/types/cookie_io_functions_t.h;h=c04319a2dcc1c71b74161c826468a8a87fa4590e;hb=HEAD

[10] glibc authors. The GNU C Reference Manual. Retrieved from https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html

[11] Jens Axboe. Flexible I/O Tester. Retrieved from https://fio.readthedocs.io/en/latest/fio_doc.html

[12] Jens Axboe. libpmem.c. Retrieved from https://github.com/axboe/fio/blob/4db5740d27942e5a36b091ace763b188888b6f9e/engines/libpmem.c#L236

[13] Jens Axboe. psync.c. Retrieved from https://github.com/axboe/fio/blob/4db5740d27942e5a36b091ace763b188888b6f9e/engines/sync.c#L439-L447

[14] Jens Axboe. libpmem.c. Retrieved from https://github.com/axboe/fio/blob/4db5740d27942e5a36b091ace763b188888b6f9e/engines/libpmem.c#L73

[15] Neil Brown. 2015. A journal for MD/RAID5. *LWN.net* 665299 (2015). Retrieved from https://lwn.net/Articles/665299/

[16] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. 1994. RAID: high-performance, reliable secondary storage. *ACM Comput. Surv.* 26, 2 (June 1994), 145–185. https://doi.org/10.1145/176979.176981

[17] Felix Cloutier. SFENCE — Store Fence. Retrieved from https://www.felixcloutier.com/x86/sfence

[18] Felix Cloutier. MOVDIR64B — Move 64 Bytes as Direct Store. Retrieved from https://www.felixcloutier.com/x86/movdir64b

[19] Felix Cloutier. SYSCALL — Fast System Call. Retrieved from https://www.felixcloutier.com/x86/syscall

[20] The GNU Compiler Collection. 6.4.1.1 Common Function Attributes. Retrieved from https://gcc.gnu.org/onlinedocs/gcc-15.2.0/gcc/Common-Function-Attributes.html

[21] Intel Corporation. Namespaces. Retrieved from https://docs.pmem.io/ndctl-user-guide/concepts/nvdimm-namespaces

[22] Intel Corporation. PMDK: Persistent memory development kit. Retrieved from https://pmem.io/pmdk/

[23] Intel Corporation. libpmem. Retrieved from https://pmem.io/pmdk/libpmem/

[24] Intel Corporation. mmap_posix.c. Retrieved from https://github.com/pmem/pmdk/blob/52b9545b8d47d6689d957e096a5bc8a2cc8ac395/src/common/mmap_posix.c#L173-L175

[25] Intel Corporation. pmem.c. Retrieved from https://github.com/pmem/pmdk/blob/52b9545b8d47d6689d957e096a5bc8a2cc8ac395/src/libpmem/pmem.c#L838

[26] Intel Corporation. init.c. Retrieved from https://github.com/pmem/pmdk/blob/52b9545b8d47d6689d957e096a5bc8a2cc8ac395/src/libpmem2/x86_64/init.c#L28

[27] Intel Corporation. Intel® Xeon® Gold Prozessor 5220. Retrieved from https://www.intel.de/content/www/de/de/products/sku/193388/intel-xeon-gold-5220-processor-24-75m-cache-2-20-ghz/specifications.html

[28] Intel Corporation. Intel® Optane™ Persistent Memory 100 Series. Retrieved from https://www.intel.com/content/www/us/en/products/sku/190348/intel-optane-persistent-memory-100-series-128gb-module/specifications.html

[29] MariaDB Corporation. MariaDB. Retrieved from https://mariadb.org/

[30] Debendra Das Sharma, Robert Blankenship, and Daniel Berger. 2024. An Introduction to the Compute Express Link (CXL) Interconnect. *ACM Comput. Surv.* 56, 11 (July 2024). https://doi.org/10.1145/3669900

[31] Debian. FileSystem. Retrieved from https://wiki.debian.org/FileSystem

[32] Peter Desnoyers, Ian Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Mike Mesnier, Carl Waldspurger, Avani Wildani, and Erez Zadok. 2023. Persistent Memory Research in the Post-Optane Era. In *Proceedings of the 1st Workshop on Disruptive Memory Systems (DIMES '23)*, 2023. Association for Computing Machinery, Koblenz, Germany, 23–30. https://doi.org/10.1145/3609308.3625268

[33] The Linux Kernel Developers. RAID arrays. Retrieved from https://docs.kernel.org/admin-guide/md.html

[34] The Linux Kernel Developers. Direct Access for files. Retrieved from https://docs.kernel.org/filesystems/dax.html

[35] The Linux Kernel Developers. Syscall User Dispatch. Retrieved from https://www.kernel.org/doc/html/latest/admin-guide/syscall-user-dispatch.html

[36] The Linux Kernel Developers. struct fdtable. Retrieved from https://github.com/torvalds/linux/blob/5ee8dbf54602dc340d6235b1d6aa17c0f283f48c/include/linux/fdtable.h#L26

[37] The Linux Kernel Developers. struct fd. Retrieved from https://github.com/torvalds/linux/blob/5ee8dbf54602dc340d6235b1d6aa17c0f283f48c/include/linux/file.h#L38

[38] The Linux Kernel Developers. struct file. Retrieved from https://github.com/torvalds/linux/blob/5ee8dbf54602dc340d6235b1d6aa17c0f283f48c/include/linux/fs.h#L1259

[39] The Linux Kernel Developers. struct inode. Retrieved from https://github.com/torvalds/linux/blob/5ee8dbf54602dc340d6235b1d6aa17c0f283f48c/include/linux/fs.h#L766

[40] PostgreSQL Global Development Group. PostgreSQL. Retrieved from https://www.postgresql.org/

[41] The Open Group. POSIX.1-2017. Retrieved from https://pubs.opengroup.org/onlinepubs/9699919799/

[42] Daniel Habicht, Yussuf Khalil, Lukas Werling, Thorsten Gröninger, and Frank Bellosa. 2024. Fundamental OS Design Considerations for CXL-based Hybrid SSDs. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems (DIMES '24)*, 2024. Association for Computing Machinery, Austin, TX, USA, 51–59. https://doi.org/10.1145/3698783.3699380

[43] Linux Standard Base Core Specification for IA64 3.0Preview1. 1.9. Interfaces for libdl. Retrieved from https://refspecs.linuxbase.org/LSB_3.0.0/LSB-Core-IA64/LSB-Core-IA64/libdl.html

[44] Compute Express Link Consortium Inc. 2025. *Compute Express Link Specification.* Retrieved from https://computeexpresslink.org/wp-content/uploads/2025/11/CXL-Specification_rev4p0_ver1p0_2025November17_clean_evalcopy.pdf

[45] Yahoo! Inc. Yahoo! Cloud Serving Benchmark (YCSB). Retrieved from https://github.com/brianfrankcooper/YCSB

[46] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. Retrieved from https://arxiv.org/abs/1903.05714

[47] jemalloc. jemalloc: a general purpose malloc implementation. Retrieved from https://github.com/jemalloc/jemalloc

[48] Nikolai Joukov, Arun M. Krishnakumar, Chaitanya Patti, Abhishek Rai, Sunil Satnur, Avishay Traeger, and Erez Zadok. 2007. RAIF: Redundant Array of Independent Filesystems. In *24th IEEE Conference on Mass Storage Systems and Technologies (MSST 2007)*, 2007. 199–214. https://doi.org/10.1109/MSST.2007.4367974

[49] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. SplitFS: Reducing Software Overhead in File Systems for Persistent Memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP '19)*, October 2019. Ontario, Canada.

[50] Yussuf Khalil, Daniel Habicht, Pascal Ellinger, Frank Bellosa, Javier González, Adam Manzanares, and Vivek Shah. 2025. Transparent DAX Mappings: Towards Automatic Kernel Bypass with CXL-Based Hybrid SSDs. In *Proceedings of the 3rd Workshop on Disruptive Memory Systems (DIMES '25)*, 2025. Association for Computing Machinery, Seoul, Republic of Korea, 54–62. https://doi.org/10.1145/3764862.3768178

[51] Nicholas Krichevsky, Renee St Louis, and Tian Guo. 2021. Quantifying and Improving Performance of Distributed Deep Learning with Cloud Storage. In *2021 IEEE International Conference on Cloud Engineering (IC2E)*, October 2021. IEEE, 99–109. https://doi.org/10.1109/ic2e52221.2021.00024

[52] Kubernetes. Raw Block Devices. Retrieved from https://kubernetes.io/docs/concepts/storage/persistent-volumes/#raw-block-volume-support

[53] Yun-Sik Kwak, Bongen Gu, Seung-Kook Cheong, Jung-Yeon Hwang, and Young-Jae Choi. 2009. Performance Analysis of RAID Implementations. In *U- and E-Service, Science and Technology*, 2009. Springer Berlin Heidelberg, Berlin, Heidelberg, 47–52. Retrieved from https://link.springer.com/content/pdf/10.1007/978-3-642-10580-7_8.pdf

[54] Yibei Ling, Tracy Mullen, and Xiaola Lin. 2000. Analysis of optimal thread pool size. *SIGOPS Oper. Syst. Rev.* 34, 2 (April 2000), 42–55. https://doi.org/10.1145/346152.346320

[55] Gregor Lucka. 2023. RAID on a File System Level for GPU4FS.

[56] Peter Maucher. 2022. GPU4FS: A Graphics Processor-Accelerated File System.

[57] Micron. Micron® 7300 Series of NVMe™ SSDs. Retrieved from https://www.eurostor.com/wp-content/uploads/datenblaetter/Micron_7300_product_brief.pdf

[58] João Oliveira, João Gonçalves, and Miguel Matos. 2025. Rethinking PM Crash Consistency in the CXL Era. Retrieved from https://arxiv.org/abs/2504.17554

[59] David A. Patterson, Garth A. Gibson, and Randy H. Katz. 1987. *A Case for Redundant Arrays of Inexpensive Disks (RAID).* Retrieved from http://www2.eecs.berkeley.edu/Pubs/TechRpts/1987/5853.html

[60] pmem.io. Documentation for Linux PMEM and CXL tools. Retrieved from https://pmem.io/ndctl/

[61] Fedora Project. A Community Server OS. Retrieved from https://www.fedoraproject.org/server/

[62] The GNU Project. Bash Reference Manual. Retrieved from https://www.gnu.org/software/bash/

[63] The Linux man-pages project. read(2) — Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/read.2.html

[64] The Linux man-pages project. write(2) — Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/write.2.html

[65] The Linux man-pages project. md(4) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man4/md.4.html

[66] The Linux man-pages project. mdadm(8) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man8/mdadm.8.html

[67] The Linux man-pages project. numa(7) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man7/numa.7.html

[68] The Linux man-pages project. ext4(5) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man5/ext4.5.html

[69] The Linux man-pages project. mmap(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/mmap.2.html

[70] The Linux man-pages project. open(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/open.2.html

[71] The Linux man-pages project. close(2) — Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/close.2.html

[72] The Linux man-pages project. memcpy(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/memcpy.3.html

[73] The Linux man-pages project. ptrace(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/ptrace.2.html

[74] The Linux man-pages project. ld.so(8) — Linux manual page. Retrieved from https://man7.org/linux/man-pages/man8/ld.so.8.html

[75] The Linux man-pages project. stdio(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/stdio.3.html

[76] The Linux man-pages project. syscall(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/syscall.2.html

[77] The Linux man-pages project. userfaultfd(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/userfaultfd.2.html

[78] The Linux man-pages project. dlopen(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/dlopen.3.html

[79] The Linux man-pages project. dlsym(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/dlsym.3.html

[80] The Linux man-pages project. elf(5) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man5/elf.5.html

[81] The Linux man-pages project. malloc(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/malloc.3.html

[82] The Linux man-pages project. dup(2) — Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/dup.2.html

[83] The Linux man-pages project. fcntl(2) — Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/fcntl.2.html

[84] The Linux man-pages project. exec(3) — Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/exec.3.html

[85] The Linux man-pages project. socket(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/socket.2.html

[86] The Linux man-pages project. bind(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/bind.2.html

[87] The Linux man-pages project. connect(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/connect.2.html

[88] The Linux man-pages project. send(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/send.2.html

[89] The Linux man-pages project. listen(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/listen.2.html

[90] The Linux man-pages project. recv(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/recv.2.html

[91] The Linux man-pages project. stdin(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/stdin.3.html

[92] The Linux man-pages project. proc(5) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man5/proc.5.html

[93] The Linux man-pages project. sysfs(5) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man5/sysfs.5.html

[94] The Linux man-pages project. Linux manual pages: section 4 - Special files. Retrieved from https://man7.org/linux/man-pages/dir_section_4.html

[95] The Linux man-pages project. rename(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/rename.2.html

[96] The Linux man-pages project. readlink(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/readlink.2.html

[97] The Linux man-pages project. chmod(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/chmod.2.html

[98] The Linux man-pages project. chown(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/chown.2.html

[99] The Linux man-pages project. lockf(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/lockf.3.html

[100] The Linux man-pages project. access(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/access.2.html

[101] The Linux man-pages project. fpathconf(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/pathconf.3.html

[102] The Linux man-pages project. stat(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/stat.2.html

[103] The Linux man-pages project. memset(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/memset.3.html

[104] The Linux man-pages project. pread(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/pread.2.html

[105] The Linux man-pages project. pwrite(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/pwrite.2.html

[106] The Linux man-pages project. ftruncate(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/ftruncate.2.html

[107] The Linux man-pages project. readv(2) — Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/readv.2.html

[108] The Linux man-pages project. fallocate(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/fallocate.2.html

[109] The Linux man-pages project. fopencookie(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/fopencookie.3.html

[110] The Linux man-pages project. fopen(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/fopen.3.html

[111] The Linux man-pages project. fileno(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/fileno.3.html

[112] The Linux man-pages project. sched_yield(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/sched_yield.2.html

[113] The Linux man-pages project. msync(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/msync.2.html

[114] The Linux man-pages project. mremap(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/mremap.2.html

[115] The Linux man-pages project. madvise(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/madvise.2.html

[116] The Linux man-pages project. mprotect(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/mprotect.2.html

[117] The Linux man-pages project. fsync(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/fsync.2.html

[118] The Linux man-pages project. mkfs.ext4(8) - Linux manual page. Retrieved from https://linux.die.net/man/8/mkfs.ext4

[119] The Linux man-pages project. pipe(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/pipe.2.html

[120] The Linux man-pages project. fork(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/fork.2.html

[121] The Linux man-pages project. shm_open(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/shm_open.3.html

[122] The Linux man-pages project. link(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/link.2.html

[123] The Linux man-pages project. acl(5) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man5/acl.5.html

[124] The Linux man-pages project. aio(7) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man7/aio.7.html

[125] The Linux man-pages project. pthread_kill(3) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man3/pthread_kill.3.html

[126] The Linux man-pages project. sigaction(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/sigaction.2.html

[127] The Linux man-pages project. io_uring(7) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man7/io_uring.7.html

[128] Mingyao Shen, Suyash Mahar, Heewoo Kim, Joseph Izraelevitz, and Steven Swanson. 2025. AutoSSD: CXL-Enhanced Autonomous SSDs for Low Tail Latency. In *Proceedings of the 34th International Symposium on High-Performance Parallel and Distributed Computing (HPDC '25)*, 2025. Association for Computing Machinery, University of Notre Dame Conference Facilities, Notre Dame, IN, USA. https://doi.org/10.1145/3731545.3731579

[129] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. 2010. The Hadoop Distributed File System. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, 2010. 1–10. https://doi.org/10.1109/MSST.2010.5496972

[130] Mohammadreza Soltaniyeh, Gongjin Sun, Xuebin Yao, Amir Beygi, Ramdas Kachare, Dongwan Zhao, Hingkwan Huen, Andrew Chang, Senthil Murugesapandian, and Caroline Kahn. 2025. 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)*, 2025. Association for Computing Machinery, Boston, MA, USA, 45–51. https://doi.org/10.1145/3736548.3737828

[131] QEMU Team. Block device options. Retrieved from https://www.qemu.org/docs/master/system/invocation.html#hxtool-1

[132] Valkey. Valkey. Retrieved from https://valkey.io/

[133] Valkey. anet.c. Retrieved from https://github.com/valkey-io/valkey/blob/747af1a85d6c1d43a749c59238a19d5b4b25e343/src/anet.c#L132

[134] Valkey. aof.c. Retrieved from https://github.com/valkey-io/valkey/blob/acb0b9b73e3584ed6199525a7b892c5de7bc4f5d/src/aof.c#L2416

[135] Wolley. NVMe-over-CXL. Retrieved from https://wolleytech.com/solutions-service/nvme-over-cxl/

[136] Kenichi Yasukata, Hajime Tazaki, Pierre-Louis Aublin, and Kenta Ishiguro. 2023. zpoline: a system call hook mechanism based on binary rewriting. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, July 2023. USENIX Association, Boston, MA, 293–300. Retrieved from https://www.usenix.org/conference/atc23/presentation/yasukata

[137] Pantea Zardoshti, Michael Spear, Aida Vosoughi, and Garret Swart. 2020. Understanding and Improving Persistent Transactions on Optane™ DC Memory. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2020. 348–357. https://doi.org/10.1109/IPDPS47924.2020.00044

[138] Jianping Zeng, Shuyi Pei, Da Zhang, Yuchen Zhou, Amir Beygi, Xuebin Yao, Ramdas Kachare, Tong Zhang, Zongwang Li, Marie Nguyen, Rekha Pitchumani, Yang Soek Ki, and Changhee Jung. 2025. Performance Characterizations and Usage Guidelines of Samsung CXL Memory Module Hybrid Prototype. Retrieved from https://arxiv.org/abs/2503.22017

[139] Saifeng Zeng, Ligu Zhu, and Lei Zhang. 2013. A High Reliable and Performance Data Distribution Strategy: A RAID-5 Case Study. In *2013 Ninth International Conference on Computational Intelligence and Security*, 2013. 318–322. https://doi.org/10.1109/CIS.2013.74

[140] Yekang Zhan, Haichuan Hu, Xiangrui Yang, Shaohua Wang, Qiang Cao, Hong Jiang, and Jie Yao. 2024. RomeFS: A CXL-SSD Aware File System Exploiting Synergy of Memory-Block Dual Paths. In *Proceedings of the 2024 ACM Symposium on Cloud Computing (SoCC '24)*, 2024. Association for Computing Machinery, Redmond, WA, USA, 720–736. https://doi.org/10.1145/3698038.3698539