Karlsruhe Institute of Technology

Reducing Synchronous Write Latency With a PMEM Write Cache in the Device Mapper Layer

Bachelor's Thesis
submitted by

Ilia Bozhinov

to the KIT Department of Informatics

Reviewer:	Prof. Dr. Frank Bellosa
Second Reviewer:	Prof. Dr. Christian Wressnegger
Advisor:	Lukas Werling, M.Sc.

13. Dezember 2021 – 13. April 2022

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, April 12, 2022

Abstract

Persistent Memory (PMEM) is a new non-volatile byte-addressable storage technology. While slower than DRAM, it offers significantly higher bandwidth and lower latencies when compared to SSDs. These characteristics make it a good option for improving I/O and in particular synchronous write performance. There have already been many projects working in different layers of the I/O stack aiming to achieve that goal. Most of them, however, fail to utilize the full hardware potential, require non-trivial changes to the kernel or replace existing file systems, thus resulting in increased complexity and feature duplication.

In this thesis, we present DPWC, a Device Mapper module for the Linux kernel which uses PMEM as a write cache for a regular SSD device. In contrast to many other works in the field, DPWC easily integrates with unmodified existing file systems. We use a fast on-PMEM caching structure inspired by ZIL-PMEM [40] and optimize it specifically for Intel Optane hardware. DPWC achieves significant speed-ups (1.5-2.05x) in most multi-threaded write workloads compared to the standard write cache implementation (dm-writecache) included in the Linux kernel. Unfortunately, these gains are made at the expense of performance in mixed read and write scenarios, but further optimizations for these cases may be possible.

Contents

Chapter 1

Introduction

Persistent data storage is one of the major tasks of an operating system [29]. The implementation of this functionality is often complicated as one needs to consider two mutually exclusive requirements - speed and reliability. Traditional storage devices like HDDs and SSDs are orders of magnitude slower than the processor and main memory. This necessitates the usage of volatile caches, for example the page cache in Linux [2]. There are also internal non-persistent caches in the disk devices themselves [13].

However, some applications like databases have need to safely persist their data even in cases like power outages. In these cases, storing the data only in a volatile cache is not enough. This problem is typically solved by using APIs like Linux' `fsync()`. When using `fsync()`, the calling thread is blocked until all written data has been flushed from the caches and stored on a non-volatile storage medium [44]. Unfortunately, frequent calls of `fsync()` mean an increase in write latencies and therefore a lower throughput.

Persistent memory (PMEM) is a promising new technology which can help in cases like these [32]. In the moment, a commonly available implementation - Intel Optane - provides non-volatile storage with speeds close to those of RAM [54]. These devices also require special access patterns to achieve the best performance [54], which makes device-specific software optimizations necessary.

To this date, there have been multiple projects focused on providing better support for persistent memory devices. One direction of research are new file systems like NOVA [53] and Strata [21], which are developed to maximize the performance of systems with PMEM modules. This approach is, however, not without its drawbacks. The development of new file systems for new hardware comes with a high implementation cost. In addition, Optane's capacity is currently limited to 512 GB per module [18]. PMEM is also more costly than comparable Optane SSDs [35, 36], which makes it less suitable for storage of larger data volumes.

Another approach is to use PMEM as a cache for larger and slower storage devices (typically SSDs). In Linux, this is achievable for example via the Device Mapper framework. It allows the creation of virtual block devices which can redirect read and write operations to any device. dm-writecache [50] is a Device Mapper module included with the Linux kernel, which can use PMEM as a write cache for a second, slower block device. A rather similar project is ZIL-PMEM [40]. However, it implements the caching on the file system level and manages to achieve better performance than dm-writecache. Unfortunately, its integration in ZFS makes it impossible to reuse ZIL-PMEM for other file systems.

In this bachelor thesis, we propose DPWC, a Device Mapper module which implements a write cache for traditional file systems like Ext4. Our primary motivation are the findings of ZIL-PMEM's author that dm-writecache is suboptimal for many write-dominated workloads [40]. We developed a caching mechanism similar to the one used in ZIL-PMEM and extended it so that it is applicable in the Device Mapper context. In the process we optimized DPWC specifically for Intel Optane hardware and multi-threaded write-dominated workloads. We also designed DPWC so that it is able to recover cached data in the case of a system crash.

Benchmark results show that our module achieves an 1.5-2.0x speedup compared to dm-writecache in multi-threaded write benchmarks. It is, however, able to utilize the full hardware potential only in some cases, as we found out that Device Mapper subsystem incurs significant overhead when working with fast storage devices. DPWC also performs up to 43% better than dm-writecache for some database benchmarks, but also 5-30% slower in others. Our tests also showed that while DPWC performs worse in most workloads with mixed read and write requests, there are ways to improve it.

The next few chapters describe our project in detail. After discussing relevant background information and related work in Chapter 2, we give a detailed description of DPWC's design and the motivation for it in Chapter 3. In Chapter 4 we go into several important implementation details of the module. Finally, in Chapter 5 we present the benchmark results of our module and finish in a conclusion in Chapter 6.

Chapter 2

Background and Related Work

In this chapter, we will briefly describe the Linux kernel I/O stack and identify characteristics which are important for the design and the implementation of DPWC. Previous evaluations of Intel Optane will be presented as a motivation for the design of our on-PMEM data structure. We will also take a look at previous works in the field of persistent memory, in order to understand how DPWC compares to them.

2.1 Linux Kernel I/O Architecture

The Linux I/O stack loosely follows a layered architecture [22]. When an application makes a system call for disk I/O, this request is then passed through and potentially transformed by the various layers. An overview of the important storage I/O subsystems in the kernel can be seen in Figure 2.1.

2.1.1 Applications and File Systems

Applications usually work with files via system calls like `read()` and `write()`. The former operation typically blocks the application until the data is available. In contrast, `write()` and similar system calls often do not block until the data is persisted on non-volatile storage. Instead, the data first lands into the page cache in RAM and is written back to the storage devices at a later point in time [2]. If this is not desirable, applications can use the `fsync()` system call, or open the file with the `O_SYNC` flag. The application will block during `fsync()` (or during `write()` if `O_SYNC` is used) until the data has reached persistent storage [44, 28]. This brings performance disadvantages, since device access is usually orders of magnitude slower than RAM access in the page cache.

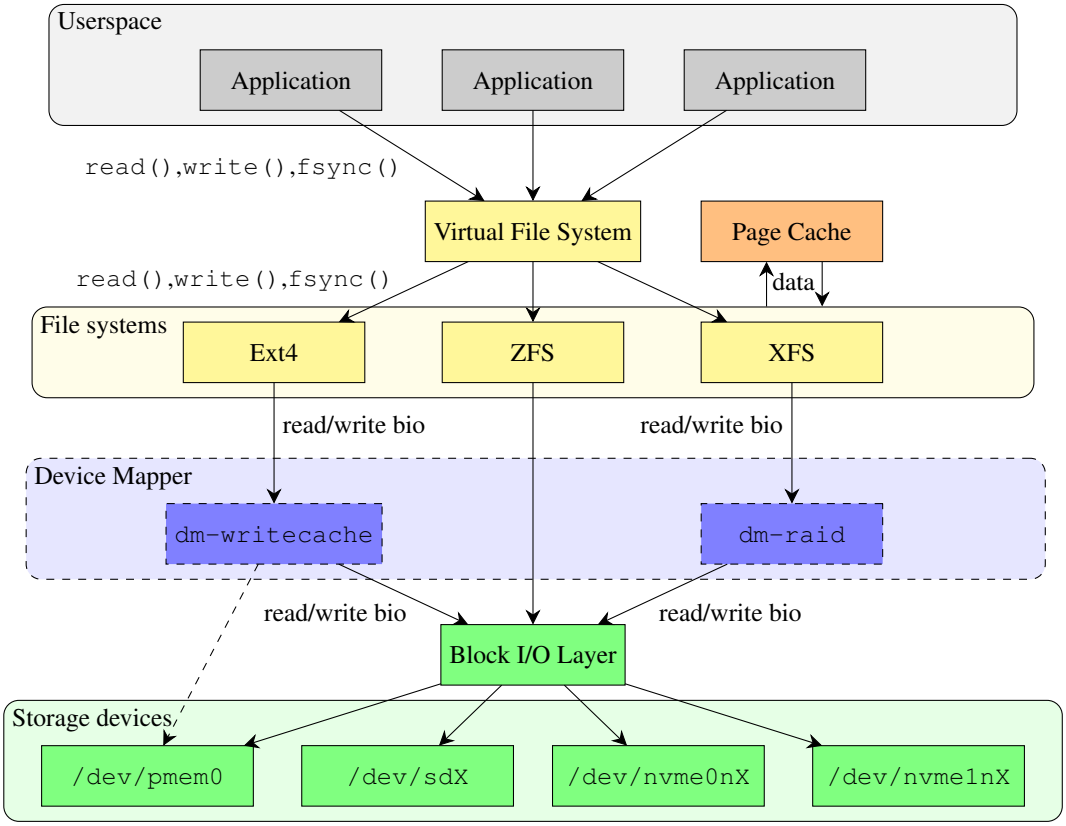

Figure 2.1: Overview of the Linux I/O stack with some example filesystems and devices. Optional components are rendered dashed.

When the application makes any of the aforementioned system calls, the request first reaches the Virtual File System (VFS). The VFS exposes different file systems (FS) to the userspace and allows multiple FS to coexist [30]. The VFS determines the correct FS for the requested file operation(s) and forwards them to the actual FS implementation.

The file system is the abstraction which provides files, directories and operations on them [30]. The FS keeps track of the layout of files and directories on the disk. When a file operation (read, write, create, etc.) comes from the VFS layer, the FS uses this information to transform it into read or write requests for the individual blocks of the underlying storage device and forwards these requests to the lower I/O layers. File systems also manage the page cache, where data is cached after it was read from the disk, or before it is written to it [2]. The FS needs to handle `fsync()` calls and synchronous write requests, too. In these cases, the FS needs to wait (and therefore block the application) until the lower I/O layers

notify that the operations are complete [13, 43].

2.1.2 Lower I/O Layers

Lower I/O layers receive only *bio requests* from the file system layer. A bio request, or just a bio, typically has information about a range of blocks (also called sectors) of the storage device [3] which are to be accessed. It also contains a buffer where data should be read or which should be written to the device. There are also other types of control requests, which are not relevant for our project.

Bios may first land in the Device Mapper (DM) subsystem, which is an optional subsystem of the Linux kernel. It makes it possible to implement mapped block devices (MBDs), which have the same interface as regular block devices [4]. This way, MBDs are transparent to the file system layers, and thus any combination of a FS and an MBD is possible. Upon receiving a bio request, the MBDs can remap it to another block device (which can be an MBD as well) or handle them in a specialized way. Typical examples of Device Mapper devices are:

dm-raid: software implementation of Redundant Array of Independent Disks [11]

dm-zero: a block device which acts exactly like `/dev/zero` - the whole disk is filled with zeroes, writes are silently ignored [12].

dm-writecache: a wrapper around a block device, which is the primary storage medium. A second device, typically a fast SSD or a PMEM module, is used as a persistent cache for write requests [50].

After bio requests have gotten their target device assigned, they are forwarded to the Block I/O scheduler, if present. The block I/O scheduler may delay, reorder and/or merge requests, so that they may be executed more efficiently [19]. Lastly, bios are passed on to the device drivers. Once the storage device has carried out the request, that information is propagated back to the higher layers via a special callback in the bio request. This kind of notification is called an endio notification.

There may be multiple software and hardware submission queues at the lower I/O layers and on the storage device itself. For example, NVMe SSDs support up to 64 K queues, each of which may hold up to 64 K requests [27]. There is no ordering defined for the requests submitted to these queues [14]. As a result, these lower I/O layers cannot give any guarantees about the order in which overlapping bios are processed. File Systems and Device Mapper devices need to wait for an endio notification before submitting further requests, if they need to ensure sequentiality.

2.2 Intel Optane

Our project, DPWC, targets Intel Optane memory modules as cache devices. Intel Optane is built on 3D Xpoint technology and provides fast byte-addressable non-volatile memory. Multiple authors have discovered several important limitations when working with Optane devices. We will briefly summarize these findings, as they have a direct influence on the design of DPWC.

Intel Optane devices may be byte-addressable, however, internally they work with blocks blocks of size 256 bytes [54]. As a result, we can only achieve the maximum possible data write bandwidth if our requests are aligned to 256 bytes. In addition, using non-temporal store instructions and store fences is more efficient than using regular store instructions and cache flushes [47]. Another important limitation of Intel Optane devices is multithreaded scalability. It has been shown that while a moderate degree of parallelism increases the total write bandwidth, increasing the number of threads writing in parallel even further is detrimental to performance [40, 51, 54].

2.3 Similar projects

There have already been multiple attempts to utilize PMEM in order to speed up file system operations. In the following sections we will present a few of them and take a look at their design, in order to be able to compare DPWC to them and understand what differentiates DPWC from each of them.

2.3.1 dm-writecache

dm-writecache is a closely related project and a direct inspiration for our work. It manages two devices: an *origin* block device, where the data is stored and read from in the long term, and a *cache* device, which can be an SSD or a PMEM device, which is used for speeding up handling of write bio requests [50]. The following is a rough description of our understanding of how dm-writecache works, based on the source code available in the Linux kernel.

At runtime, dm-writecache keeps a cache structure in the form of a red-black tree stored on the PMEM. The entries in this tree roughly correspond to (groups of) sectors on the origin device whose contents are not up-to-date. When a new read or write request arrives, dm-writecache proceeds with the following steps:

1. Obtain a lock on the cache data structure.
2. Compare the new request with entries in the cache:
    - In case that the request can be serviced by reading data from the cache or updating cache entries, dm-writecache uses the cache entries.

- Otherwise, for write requests dm-writecache may block if there is no free space in the cache. It then waits for a background writeback thread to free up space. Then, dm-writecache adds new entries to the cache. Read requests are directly forwarded to the origin device, if they cannot be read from PMEM.

3. Finally, dm-writecache unlocks the cache.

In the background, two workqueues are running. One of them takes requests from the cache structure and submits them to the origin device, in case that the free space in the cache falls beyond a certain threshold. dm-writecache takes special care to throttle the requests in order to not overload lower I/O layers. The second workqueue is responsible for handling notifications of completed bio requests. It forwards these notifications to the higher I/O layers and and marks the cache entries as free. This operation also requires obtaining the same global lock on the caching structure to remove the written-back entries from it.

dm-writecache suffers from lock contention in multi-threaded workloads [40]. The global locking strategy makes it impossible for multiple threads to write to the caching structure at the same time. In turn, this means that we cannot utilize the full bandwidth that the Intel Optane devices can provide. Solving this problem is the main goal of this bachelor thesis.

2.3.2 ZIL-PMEM

Another attempt to use Intel Optane devices as a cache for file system operations in the context of the Zettabyte File System (ZFS) is ZIL-PMEM [40]. ZFS stores synchronous writes in the ZFS Intent Log (ZIL), whose entries can be replayed in the case of a crash to restore the potentially missing data [6]. The ZIL can also be stored on a separate device, which is usually chosen to be faster than the main storage device and thus acts as a write cache for ZFS [41].

ZIL-PMEM is a specialized implementation of the ZIL which uses Intel Optane to store the log operations. By using a specially crafted data structure, ZIL-PMEM is able to achieve a good utilization of the PMEM log device. It works especially well in multi-threaded write-dominated workloads. Unfortunately, ZIL-PMEM is tightly integrated into ZFS. This means that it cannot be easily reused in any other file systems. In this bachelor thesis, we develop a caching structure very similar to the one used in ZIL-PMEM, but put it in the context of the Device Mapper subsystem. We also extend it with mechanisms for handling read and overlapping write requests, which is not necessary in ZIL-PMEM, as it is only a file system log.

2.3.3 Specialized Filesystems

There have been many other attempts to create an efficient file system which can work with PMEM devices. A typical example is the NOVA file system [53]. While providing impressive performance, it utilizes PMEM as the main storage medium. However, as mentioned in Chapter 1, Optane's capacity is limited and costs more than comparable SSDs [18, 35, 36]. This is less of an issue with cross-media file systems like Strata [21] and Ziggurat [55]. Ziggurat stores files on multiple storage devices organized in tiers (both PMEM and SSDs) and migrates files between tiers based on different policies and heuristics. Strata, on the other hand, uses a userspace component allowing applications to use PMEM, and a specialized kernel file system for asynchronously writing back data to the slower storage devices.

By working in the file system and application layers, these projects have more possibilities for determining which data should be cached. They also have to deal with less software management overhead, which can have a big impact when working with fast storage devices [42]. As a result, approaches like these are able to provide very good performance with minimal overhead.

Nevertheless, correctly implementing a whole file system is a complex and bug-prone undertaking. Many of these projects require big changes to the Linux kernel or special support in user space applications. In the words of NOVA's authors, "We have run complex programs on NOVA. There are, of course, many bugs left to fix." [26]. In addition, a file system often provides its own unique features, which have to be reimplemented in a new file system. On the other hand, a solution in a lower I/O layer like the Device Mapper subsystem can be reused with existing file systems and requires no changes to existing software stacks. For these reasons we believe a more generic approach than the aforementioned projects would be preferable.

Chapter 3

Design

Taking the information from Chapter 2 into account, we designed DPWC, a potential alternative to dm-writecache. DPWC manages two devices, the *origin device*, which is the main storage medium, and the *cache device*. Broadly speaking, DPWC receives read and write requests from higher I/O layers of the OS. It then stores write requests on the cache device, and at a later point on the main storage device. DPWC forwards read requests to the main storage device. Depending on which component handled the request, DPWC or the storage device notify the higher I/O layers after the request has been handled.

During the design, we had the following goals and were willing to make the following trade-offs:

- DPWC should be able to extract the maximum performance possible from the cache device when caching write requests. We are willing to use more RAM or PMEM space in order to achieve that.
- DPWC should work well in highly multi-threaded workloads. A global locking strategy of all necessary data structures similar to dm-writecache is not acceptable.
- We will optimize for write-dominated workloads. This means trading off read performance for write performance, if necessary.
- DPWC must ensure that no data on the cache device is lost in the event of a system crash or a power outage.
- DPWC should handle all operations correctly. This means that deterministic applications can read the same data from the disk after executing any sequence of operations, regardless of whether DPWC is used or not. In particular, DPWC should not reorder two write requests or a read and a write request to the same sector.

- We should write data efficiently on the origin device. This way, we will be able to free up space on the cache device faster and cache more requests in total. Nevertheless, in case that the cache capacity is not enough, we still need to guarantee correctness, as far as possible.

Before diving into the details of our design, we will take a look into the general components of DPWC and the interaction between them. Figure 3.1 provides a visual overview of these components and their integration with the Linux kernel.

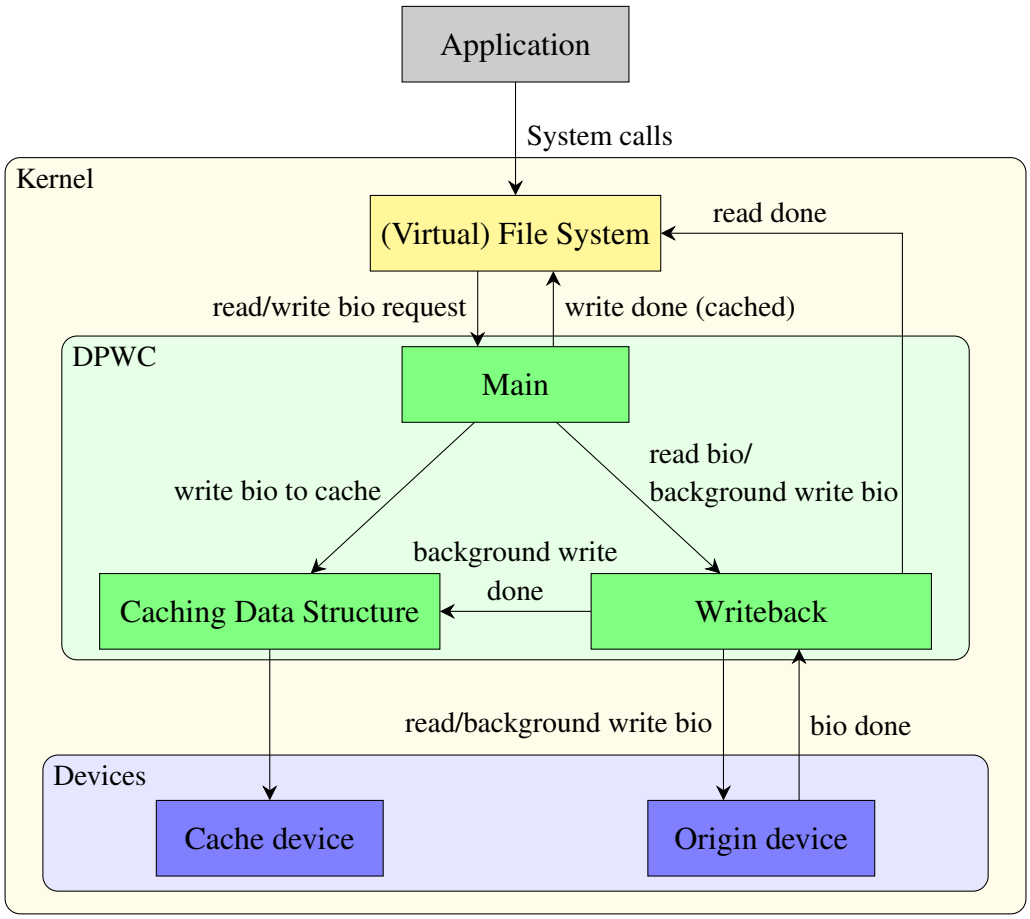

Figure 3.1: Overview of DPWC at runtime in the context of the Linux Kernel I/O stack. The flow of BIO requests and their endio notifications is shown.

3.1 Overview

DPWC consists of four main parts:

The **main module** receives I/O requests (both read and write) and dispatches them to the appropriate place. It forwards read requests directly to the writeback system. Write requests are first cached on the cache device via the caching data structure. Afterwards, the main module passes them to the writeback system.

The **caching data structure (CDS)** is responsible for managing the space on the cache device and storing write requests on it.

The **writeback system** dispatches read and write requests to the origin device. It also keeps track of ordering dependencies between requests and ensures sequentiality of requests where needed. Lastly, it notifies the CDS when a write request has been successfully stored on the origin device.

The **replay module** recovers the data from the CDS in the case of a system crash. The data is then copied, or *replayed*, to the origin device. The replay module is rarely used, but very important for the persistency guarantees DPWC has to make.

3.2 The Main Module

The main module's only responsibility is dispatching new requests. It differentiates three distinct cases:

- If the incoming request is a read request, the main module forwards it to the origin device via the writeback system. The main module does no further handling of this request, as the origin device will notify the higher I/O layers as soon as the request has been completed.
- If the incoming request is a write request and there is enough space on the cache device, then the following steps are taken:
  1. CDS writes the request in the cache. After this point, the request is considered to have been written to persistent storage.
  2. The main module duplicates the request.
  3. The duplicate request is forwarded to the writeback system for long-term storage.
  4. We notify the higher I/O layers that the original request is complete. In the absence of DPWC (or a similar caching mechanism) this notification will be sent by the origin device much later, after it has processed the request. However, the CDS and the replay module guarantee that the data is now persistently stored. In addition, the writeback system guarantees that other requests for the same sectors will not be reordered with the current request. Therefore it is safe to notify the higher I/O layers and the application where the request originated that they do not need to wait any longer in order to submit further requests.

- If the incoming request is a write request and we do not have enough caching space, the request must be forwarded to the writeback system for long-term storage and no caching can be done. The request may have to wait for a long time until it is processed.

An important detail here is the interface between the main module and the writeback module. We use a strictly ordered queue of requests. The main module always pushes new requests to the back of the queue, while the writeback module consumes requests from the front. This queue is needed for two reasons. On one hand, we want to ensure that the request handler does as little work as possible, in order to reduce latencies for the applications. This is why we offload the writeback module on a different thread. On the other hand, the writeback module needs to handle the requests in chronological order, as discussed in Section 3.5, which necessitates the aforementioned queue.

3.3 Caching Data Structure

The main module relies on the caching data structure (CDS) to manage data on the cache device and to store new requests. The CDS needs to be able to manage the concurrency level for PMEM access in order to ensure optimal performance on Intel Optane hardware as described in section 2.2. In addition, the CDS needs to be able to efficiently clear cached data as soon as it is written back on the origin device. Nevertheless, no data should be lost in the case of a system crash. For the purposes of recovery in such cases, we also need to store additional metadata for the use of the replay module.

Taking these requirements, the information in section 2.2 and the caching structure of ZIL-PMEM [40] in account, we designed a very similar data structure as depicted on Figure 3.2. In the following paragraphs, we describe our design in detail and explain how the different parts work together to fulfill our goals from the beginning of this chapter.

The CDS consumes the whole cache device. It partitions the available space into a fixed number of equally-sized physical generations. Each physical generation is in turn partitioned into a fixed number of equally-sized chunks. Each chunk may contain zero or more entries, which represent cached write requests. Each physical generation, chunk and entry have their own header. The information stored in this header is used by the replay module in the case of a crash. Details of the recovery procedure can be found in Section 3.4.

Since we have a fixed (and therefore limited) amount of physical generations, the CDS also keeps track of *virtual* generations at runtime, each having its own number. These numbers act like unique identifiers for virtual generations and impose a total order on them. Each virtual generation is mapped to a single physical

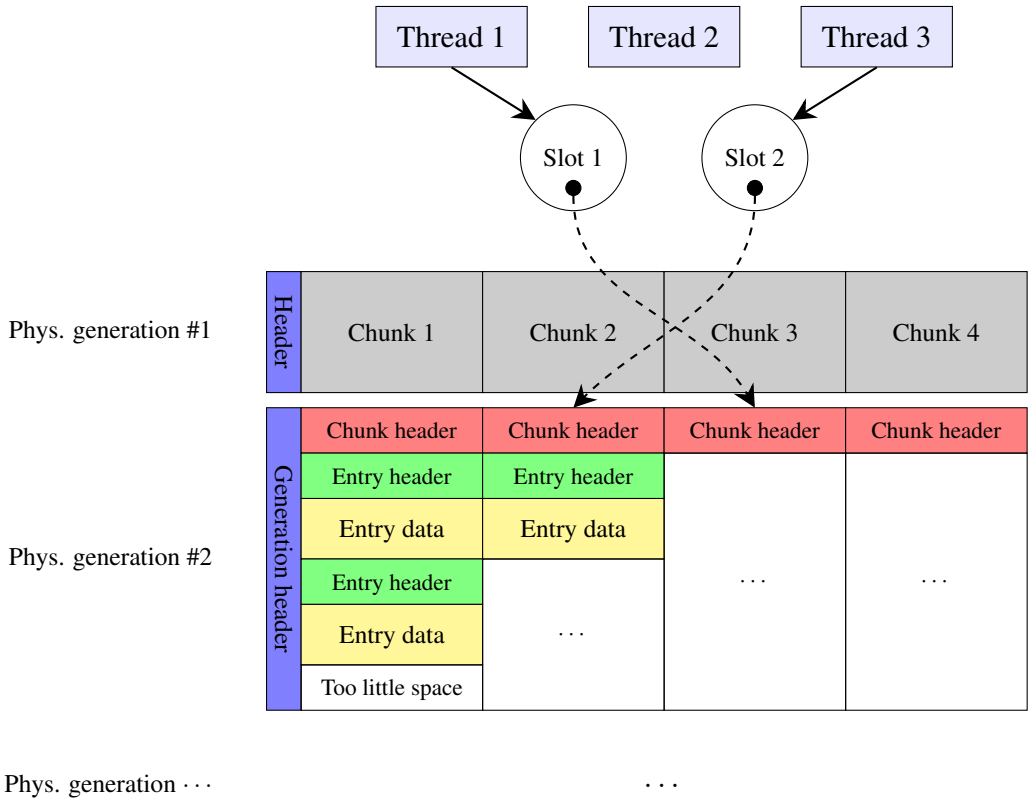

Figure 3.2: Example of the PMEM data structure at runtime, with 3 writing threads. Configuration uses 2 slots, 4 chunks per generation and at least 2 generations in total. The current virtual generation is mapped to physical generation #2. Chunks 2 and 3 are assigned to Slot 1 and 2, respectively. Threads 1 and 3 use the depicted slots, Thread 2 is waiting.

generation. The number of virtual generations is unbounded, and therefore during the course of operation many virtual generations map to the same physical generation. In this way, generations allow us to efficiently recycle PMEM space after the writeback module has processed all entries in it.

At any given point, one virtual generation is marked as the *current* generation. Incoming write requests are always stored in chunks of the *current* generation, that is, in chunks of the corresponding physical generation. When the *current* generation has no more space to store new entries, a different physical generation is mapped to the next virtual generation. Then, the next virtual generation becomes the *current* generation. This operation is described in detail later.

In order to enable parallel writing from multiple threads, we use the slot system from ZIL-PMEM [40]. The slot system also doubles down as a mechanism

for limiting the number of concurrent writers, which is necessary for optimal performance as discussed in Section 2.2

Each thread which needs to store a new entry on the CDS needs to obtain one of the fixed number of available slots. Each slot may point to a chunk of the *current* generation. At any given time, at most one thread can hold a given slot. Thus, a thread gains exclusive access to the chunk mapped to that slot. After acquiring a slot, a thread needs to differentiate the following cases:

1. The slot already has an assigned chunk with enough remaining free space for the new entry. In this case, the thread stores the write request as a new entry and updates the available chunk size, releases the slot it holds, and finally returns. The implementation of this operation needs to take care of crash consistency, as the order of these operations has impact on the working of the replay module, as discussed in Section 4.1.

2. In case the slot does not have an assigned chunk, or that chunk does not have enough free space, the thread differentiates between two cases:

   (a) There is an empty chunk in the *current* generation which has not been assigned to a slot yet. The chunk is assigned to the slot held by the thread, and the thread continues as in case 1.

   (b) There is no empty chunk in the *current* generation. In this case, the *current* generation is *closed-off*, meaning no new writes will happen to chunks in it. This includes both the *current* virtual generation and the corresponding physical generation. All slots' assigned chunks are cleared. Another, empty physical generation is mapped to the next *virtual* generation. The latter then becomes the *current* generation. Finally, the thread may continue as in case 2a). We do these operations in order to prevent other threads from writing to a *closed-off* generation.

This algorithm may fail only if the cache device is nearly full, that is, there are no empty generations and chunks. In these cases, the operation is unsuccessful and the main module handles the request as described in the previous section.

A *closed-off* physical generation P with mapped virtual generation V may be reset and mapped to another virtual generation, as described in case 2b) above. This is possible as soon as there are no pending (i.e. not written to the origin device) requests in V and in older virtual generations. For this reason, the CDS tracks the number of pending requests in each physical generation. Adding a new request to any chunk of the generation increases this number, a request done notification from the writeback module decreases it. If a P (and therefore V) has no more pending requests, it needs to wait for all generations before it to be completely written and then cleared. After that, P can be cleared and may be reused for new virtual generations.

3.4 Replay

The motivating factors for the design of the CDS have been multi-threaded performance (covered by the slot/chunk system) and data loss prevention in extreme cases such as a system crash. In such scenarios, the main module invokes the replay module the next time the system is started. The replay module then uses the information stored in the CDS in order to figure out which requests have not been written to the disk and *replays* them. In this context, *replaying* a request means creating a new request for the origin device with the contents stored in the corresponding entry in the CDS.

For the replay we need to recover all requests stored in the CDS. We also need to know the chronological order of operations, in case different operations overwrite each other. As explained in the next paragraph, the CDS guarantees that the entries it contains after a crash represent all requests which may need to be replayed to prevent loss of data. We need to concern ourselves only with requests which we have previously marked as persistently stored. Moreover, we can obtain the chronological order by sorting the physical generations and entries within them. Replaying the entries in that order is sufficient to restore

To facilitate this, each physical generation on the cache device has a *generation identifier (GID)* stored in its header. The GID is the same as the number of the last virtual generation which was mapped to the given physical one. Each time the *current* generation changes, the GID of the corresponding physical generation is updated. The CDS also guarantees that for two virtual generations with GIDs A and B, $A < B$, any request in A was submitted earlier than any request stored in B. This is because once A is *closed-off*, no further writes to A are possible. Since $A < B$, B is a generation which became current at a later point than A, so entries in B were added after A was *closed-off*, thus after any entries in A.

Each entry in the CDS also gets its *entry identifier (EID)* stored in the entry's header when it is written. The EID is necessary to restore the chronological order of entries within a generation. This information is impossible to recover from simply knowing the chunks and entries present in a physical generation. While entries inside a chunk are stored chronologically, a single thread may write to different chunks. Without a generation-wide identifier like the EID, we cannot know the order of the resulting entries.

We can also guarantee that if there are n entries in the CDS, they represent exactly the last n write requests which DPWC has processed. This is because the CDS caches *all* write operations. We remove entries from the CDS only when clearing a whole physical generation P. The CDS guarantees that when clearing P (and its corresponding virtual generation V), all generations before V we also cleared. It follows that the n entries in the CDS are a suffix of the chronological list of operations, and any other requests are already stored in the origin device.

Lastly, requests are idempotent, that is, executing the same request twice does not change the result. Therefore, even if some of the n entries in the CDS have been already written to the origin device, replaying all of them still produces the correct result.

These considerations enable the replay module to replay all entries on the CDS after a crash. As we have already established, these entries suffice to restore the system to a correct state. Here, a correct state means that the data on the origin device will be indistinguishable from the data which would have been stored there if no crash had happened.

3.5 Writeback Module

The last important component of DPWC - the writeback module (WBM) - is responsible for managing requests to the origin device.

In contrast to dm-writecache, DPWC does not use the cached data in order to speed up read requests. Instead, we rely on the page cache to service most of these requests, as described in Section 2.1. The remainder of the read requests are serviced by the origin device itself.

This poses a problem for our caching strategy, which notifies the higher I/O layers as soon as a write request is stored in the cache. An application could submit a read request or even more write requests for the same sector immediately after it receives this notification. In both cases, the WBM would receive these requests from the main module, potentially before the origin device has processed the first request. The WBM needs to ensure that the new request(s) are blocked until the first request completes. Otherwise, we might submit the two requests to the origin device in parallel. The ordering of the two requests is then undefined and may result in wrong data on the disk, or the wrong data being read by the application (see Section 2.1.2).

The main module already serializes requests for the WBM as discussed in Section 3.2. However, modern NVMe SSDs benefit from having multiple requests in their queues [27]. Therefore, we use a data structure (an interval tree ordered by the sectors accessed by a request) in order to detect conflicts between two requests. Two requests are conflicting if they access the same sector(s) and at least one of them is a write request. Two read requests are never in conflict, as the order in which they are handled does not change the end result. The exact steps for processing requests are given in Algorithm 1.

New requests are received via a single queue from the main module and are processed sequentially. The algorithm ensures that only non-overlapping requests may be sent to the origin device at the same time. The device may process them in parallel, potentially speeding up the performance of the origin device. On the

Algorithm 1 Algorithm for processing requests in the WBM.

```
1: activeRequests ← IntervalTree of Requests
2: procedure PROCESSNEWREQUEST(req)
3:     allPotentialConflicts ← activeRequests.getIntersections(req)
4:     activeRequests.insert(req)
5:     for all prev ∈ allPotentialConflicts do
6:         if prev.is_write() or req.is_write() then
7:             waitForCompletion(prev)
8:         end if
9:     end for
10:    submit(req)    ▷ At this point, there are no conflicts with earlier requests
11: end procedure
```

other hand, a second request for the same sector as a previous request will be blocked until the older request is submitted and completed. This guarantees the sequentiality of conflicting requests, which is the core goal of the WBM.

Once the origin device tells the WBM that a request is done, it is removed from the interval tree and it is marked as complete. Blocked requests for the same sectors may then be submitted, except if they are not waiting for further requests. Finally, for completed write requests, the WBM notifies the caching data structure that an entry in it has become obsolete. For completed read requests, the WBM notifies the higher I/O layers that the requested data is available.

A drawback of this design is that the head-of-line phenomenon [16] is possible. If a request cannot be submitted immediately, `waitForCompletion()` will block the thread processing new requests. In turn, this prevents all subsequent requests from being submitted, even if they have no conflicts. Fortunately, this problem can be solved by introducing a second queue where blocking requests are transferred to, after they are added to the interval tree. This way, conflicting requests wait without blocking other, non-conflicting requests. Further improvements are also possible and are discussed later in Section 5.2.5.

3.6 Discussion

There are many similarities between ZIL-PMEM [40], dm-writecache [50], and our project. However, our approach differs significantly due to its writeback policy and the way we make use of the PMEM space.

First of all, despite working as a cache, DPWC does not fit into the traditional cache model, where write caches are either write-through or write-back [45]. Write-through caches store data on the cache medium, but also block until the

data is stored on the primary storage medium as well [45]. In contrast, DPWC indicates that the request is complete as soon as it is stored on the cache medium. DPWC also is not a traditional write-back cache. Write-back caches, like DPWC, do not wait for data to be written on the main storage medium [45]. However, write-back caches usually do not update the backing memory immediately. Instead, they hope to cache multiple requests in the same location, and/or service read requests from the cache [45, 8]. DPWC deviates from this: we make no attempts to coalesce requests to the same blocks, and we do not service read requests from the cache.

This nonstandard writeback policy is also the root of the difference between our CDS and ZIL-PMEM's PRB [40]. ZIL-PMEM writes back requests in batches, and services read requests from the page cache. As DPWC works below the file system level, it cannot rely on the page cache, and therefore needs a different strategy for handling conflicting read requests.

Dm-writecache takes a more traditional approach. It works like a write-back cache. Entries in the PMEM are dedicated to particular blocks on the disk. Subsequent write requests to the same disk blocks may be coalesced together. Read requests may be serviced by the entries in the cache, if the data has been cached and not written back yet. Writes to the origin device happen only when the used cache space reaches a particular threshold. The implementation, however, uses a binary tree stored in the PMEM to organize the cached data. This has inherent scalability issues, as the tree cannot be manipulated by multiple threads at the same time. Our approach also requires synchronization between all writer and reader threads when queuing requests to the writeback module. In contrast to dm-writecache, however, there may be multiple cache entries for the same sectors, and writers may write in parallel to the PMEM memory. Contention for DPWC's lock is also lower, because there are no expensive operations in the critical section - only adding an element to a queue.

Chapter 4

Implementation

We implemented DPWC as an out-of-tree Linux kernel module. As our design was made with the Device Mapper architecture in mind, we did not encounter any significant difficulties integrating with the rest of the Linux kernel. Therefore, our module works with the unpatched upstream Linux kernel. The full source code can be found at `https://github.com/ammen99/dpwc`.

During the implementation, we had to make several decisions which impact the performance and reliability of DPWC. In this chapter, we will present the most important among these decisions and discuss their importance. At the end we will also list a few limitations of our particular implementation.

4.1 Data on PMEM and Crash Consistency

In order to support the replay operation, the CDS needs to store the following information in the various headers on the PMEM module in addition to the requests' data itself:

- Generation ID - 8 bytes in the generation header
- Used space in each chunk (including header) - 8 bytes in the chunk header
- Entry ID - 8 bytes in the entry header
- Entry size (including header) - 8 bytes in the entry header
- Entry start sector - 8 bytes in the entry header

All data needs to be correctly and fully written on the PMEM. Otherwise, the replay operation may not be correct. In the following paragraphs, we will analyse DPWC's implementation in order to show that it will not lose any persistently stored data. For the purposes of this analysis, we assume that we can store a 64-bit integer atomically on PMEM. We also assume that the hardware is able to store

and then read data without errors, and assume that there are no implementation bugs in our code or in the Linux kernel.

First, we will take a look at the exact implementation of discovery of entries on the PMEM after a crash. We use a fixed number of generations and chunks per generation. Therefore, we can calculate the offset of each chunk inside the PMEM memory region. Then, we can use Algorithm 2 to process all chunk entries:

Algorithm 2 Algorithm for iterating entries in a chunk on the PMEM

```
1:  procedure REPLAYCHUNK(char *start)
2:      used ← readChunkHeader(start) - CHUNK_HEADER_SIZE
3:      entry ← start + CHUNK_HEADER_SIZE
4:      while used > 0 do
5:          (sector, id, size) ← readEntryHeader(entry)
6:          entryData ← entry + ENTRY_HEADER_SIZE
7:          processEntry(entryData, sector, id, size - ENTRY_HEADER_SIZE)
8:          used ← used - size
9:          entry ← entry + size
10:     end while
11: end procedure
```

The replay algorithm does not have any error checking mechanisms. It is therefore important that the CDS carefully writes data to the PMEM. We need to ensure that all stored entries are discoverable by the algorithm, and that it will not find any incomplete entries (which may wrongly overwrite valid data on the origin device).

This is why when adding a new entry, the CDS does the following operations in the given order:

1. Write the entry header on PMEM.
2. Copy the write request data in the entry data region on PMEM.
3. Update the chunk's header by increasing the used space. Assumption: atomic operation with 64 bits.

Each of these operations is done via non-temporal store instructions because CPU caches might be lost in a system crash scenario. In addition, we use an `sfence` instruction after the second and the third operations, in order to guarantee that the writes have reached PMEM [7]. This way, if the system crashes at any point before operation 3 is complete, the partly written data for the new entry will be ignored by the replay module. Note that this data is lost, however we have never given any notification to the userspace that the data is written.

On the other hand, if we did notify userspace that the operation is successful, the amount of used space marked in the entry's chunk's header may only increase. We also never change the contents of an entry or its header once it is written on the PMEM. This way, the replay module will always be able to discover the request and replay it in case of a crash.

One last consideration is multi-threaded access to the same chunk. However, our slot system guarantees that only a single thread may hold a given slot at a time, and threads do not release their slots until they are done writing. A chunk may only be assigned to one slot. Therefore, no races are possible when updating a single chunk in a thread. The generation header is also correct, as it is written only by the single thread which advances the *current* generation.

4.2 PMEM Space Usage

In Section 2.2 we have already established that data alignment is very important when dealing with Intel Optane memory. We need to ensure that every entry's data is aligned at 256 bytes. This way, we can use an optimized inner loop for data transfer with AVX-512 instructions. For simplicity, we made every generation, chunk and entry header 256 bytes big. We also rounded every generation and chunk's size down to a multiple of 256. In this configuration, we do not need any extra padding for the entry data, as BIO requests in the Linux kernel always access a group of sectors, and each sector has size 512 B. This makes entry data automatically have a length which preserves the 256-byte alignment.

A potential concern with this strategy is the waste of PMEM space. While there is typically only a small number of generations and chunks (in our configuration, we did not use more than 4 generations and 32 chunks per generation), there may be many chunk entries. In the extreme case, each write BIO updates a single sector. In this case, we need $256B$(padding) $+ 512B$(data) $= 768B$ per entry, which means that roughly $\frac{232}{768} \approx 30\%$ of the space is wasted, after accounting for the replay data in the header. However, in practice we usually get a BIO request for a whole page from the page cache. In this case, we need $256 + 4096 = 4352B$ per entry, and only $\frac{232}{4352} \approx 5\%$ is wasted.

The amount of wasted space is further reduced if we reuse some of the padding in the header to store temporary data for the interval tree used in the WBM. In our final implementation, we used an additional $160B$ for this purpose. This brings the wasted space down to just $\frac{72}{4352} \approx 1.7\%$.

4.3 Synchronization

During the implementation of DPWC we had to be careful in order to avoid locking as much as possible. There are, however, multiple places where threads are forced to share resources.

The CDS needs to synchronize writer threads while they store data on the PMEM. Each thread needs to obtain a slot in the CDS. We solve this problem using a similar strategy to ZIL-PMEM [40]. We use a semaphore to limit the number of threads which can obtain a slot at the same time. The semaphore is initialized to the number of slots, as it is impossible to have more threads writing to PMEM than the number of slots. Obtaining a free slot can then be implemented using an atomic compare-and-exchange loop. During our experiments, we observed that in practice, there is low contention at the semaphore, therefore threads are unlikely to enter sleep. Thus, this operation is typically fast.

A thread may need to obtain a new chunk to assign to a slot, because the slot either has no chunk assigned, or the chunk is almost full. As long as the current generation has free chunks left, we do not need any locking. Instead, we keep an atomic integer which indicates the number of the next free chunk in the current generation. To take a new chunk, a thread simply executes an atomic fetch-and-increment operation on that integer.

The synchronization gets more complicated when there are no more free chunks in the current generation. In this case, incrementing the aforementioned counter gives us an invalid chunk. We need to mark the current generation as *closed-off*. However, we may need to synchronize with writeback notification threads, which may want to clear the generation once it has no pending requests. As this happens only once per generation advancement, which is a comparatively rare event, we used a lock to synchronize access here.

The writeback module also has similar issues. We need to synchronize the thread processing new requests with threads handling the completion of requests on the origin device. As this is a background module and therefore not critical for write latencies, we opted for a simpler strategy with locks on the data structure. In addition, operations done while this lock is held do not involve any I/O, and therefore the critical sections are expected to be short and contention to be low.

4.4 Queuing

The Linux kernel provides a workqueue interface which we used in multiple places in our project. In its essence, a workqueue is a queue of work items. Each item is processed asynchronously to the thread(s) which placed it in the queue. There are multiple flavors of workqueues available [9]. The first type are ordered

workqueues. Items in such queues are processed strictly sequentially. This makes ordered workqueues a great fit for the communication interface between the main and the writeback modules, as explained in Section 3.2.

The second type, "regular" workqueues, support the execution of multiple work items in parallel. They can be either bound or unbound [9]. Bound workqueues attempt to process work items on the CPU core they were submitted, to reduce inter-core data transfer. Unbound workqueues may process work items on any CPU core, therefore allowing a higher degree of parallelism in work item handling. This policy, however, may hurt performance if the work items are already submitted from different CPU cores.

We used an unbound workqueue in the writeback module for submitting bios, a slight deviation from our design where bios are submitted directly from the WBM request handler. The benefits from this are twofold. On one hand, our experiments show that the cost of a call to `submit_bio()` is not negligible. By using multiple threads to submit bios, we are able to parallelize this operation. On the other hand, different CPU cores usually use different hardware submission queues (if these are present). Thus, by using multiple CPU cores to submit bios, we are able to utilize the origin device better.

4.5 Known Limitations

As our work was done in the context of a bachelor thesis with a limited time frame, our primary goal were correctness and performance. We did not attempt to implement a production-ready module. Therefore, our implementation has several limitations:

- Our kernel module has a limited number of tunable parameters. Currently, only the number of generations can be changed without recompiling the module. Replay will not work if the module is loaded with a different number of generations or chunks after a crash. Fixing this requires storing these values on the PMEM and reading them before replaying.

- DPWC does only very limited error checking at startup and almost none at runtime. We assume that every input to our module is correct. We also do not handle any potential hardware errors. Instead, we rely on the ECC implemented in Intel Optane to recognize and correct single-bit errors. Adding additional error detection mechanisms is possible by using safer memory reading procedures like `memcpy_mcsafe()` in the replay code.

  A simple error handling mechanism would be stopping the replay procedure as soon as an error is found. We can detect the earliest error, because we

have the chronological order of entries. Errors will show up as duplicated or missing EIDs in the sorted sequence. However, due to the idempotence and independence of requests for different sectors, one can implement a more sophisticated error handling procedure with multiple cases, depending on where an error has occurred.

An error in an entry's data does not prevent any other requests from being properly replayed. In fact, we may cache multiple requests for the same sector. Therefore, it is also possible that this data is not needed at all, in case a later request would overwrite the data anyway.

Errors in an entry's headers are a bit trickier:

- An error in an entry's sector invalidates the whole entry. We could, however, continue replaying other entries. In this way, we limit the amount of lost data to the set of sectors in actually faulty entries.
- An error in an EID may be completely recoverable. As long as there is at most one wrong EID per generation, we may recover it by looking at the set of EIDs from other entries in that generation. EIDs are sequential, so if there is a missing EID, we know that the faulty entry must have that EID.

  We may also not need to know the EID at all, for example in case there are no other requests writing to the same sectors. In this scenario, we can replay the entry in any order relative to the other entries.

  However, if there are muliple errors in entry headers, we may not be able to recover the data and it will be lost. Similar arguments can be made for errors in the GIDs stored in the generations' headers.
- An error in an entry's size invalidates any entries afterwards. This is due to the entry recovery algorithm 2. The same is true for errors in chunk headers. We can limit the amount of lost data by nonetheless replaying other generations/chunks.

These mechanisms may also be combined with the error correction facilities of the file system. For example, the FS may be able to recover the data in some blocks, or use its journal to repair errors in the entry headers. Other systems may, however, benefit from a fail-fast error detection (for example, to facilitate restore-from-backup to a safe state). Therefore, we believe that an ideal implementation would allow a flexible error handling policy. We cannot foresee any technical difficulties implementing any of these procedures in our project.

- We have not implemented any throttling mechanisms to prevent excessive amount of requests queued on the origin device, or in the request passing

queue between the main module and the WBM. This means that our module may potentially use a lot of system memory (RAM) for buffering requests.

Chapter 5

Evaluation

To validate our approach, we tested DPWC for both correctness and performance. All of the tests were carried out in the same setup inside a virtual machine. Our host system runs a Fedora 35 Linux distribution with a stock `5.15.18-200` kernel. The system has two Intel Xeon Silver 4215 CPUs and 128 GB of RAM. We used a QEMU virtual machine based on Fedora Cloud Base 35 and the stock `5.16.9-200` kernel. The VM uses 8 cores pinned to 8 physical cores of the same host CPU. It also has 16 GB RAM allocated to it. We used the built-in vNVDIMM QEMU virtualization support [46] to passthrough one region of a Intel Optane Persistent Memory 200 Series module with a capacity of 40 GB. We also used the VFIO kernel module [1] on the host system to passthrough a Micron 7300 SSD with a capacity of 1 TB. These two devices were used as the cache and the origin device respectively for DPWC and dm-writecache.

5.1 Correctness Testing

As our module integrates with the rest of the Linux kernel and makes extensive use of synchronization primitives and I/O devices, we opted for a testing strategy based on actual user-space test clients instead of unit tests.

We developed several specialized test clients which stress individual parts of DPWC. One of these clients, simply called `verificator`, works in two phases. It first opens a file and does random one byte writes on it. The operations are generated via a fixed seed and are thus reproducible. `verificator` executes `fsync()` after every operation to trigger actual operations on the disk. In the second phase, `verificator` reads the data from the disk and verifies it. We ran `verificator` multiple times and also with multiple parallel instances and found no errors.

We used the same test client for verification of the replay module. We manu-

ally disabled all writeback functionality in DPWC's code. After that, we ran the `verificator` workloads until the cache device was nearly full. As writeback was disabled, the data was stored only on the cache device, and any read requests (which are serviced by the origin device) returned wrong information. Reloading DPWC triggered the replay procedure. After it completed, all the data on the origin device was correct.

For stress-testing the writeback module, we used a custom Device Mapper virtual device `dpwc-test` as a wrapper around the origin device, as shown on Figure 5.1. `dpwc-test` manages a single block device and forwards all incoming requests unmodified to it. In doing so, it injects configurable (via DebugFS [10]) delays in the submission of these requests. The purpose of these delays is to simulate full hardware submission queues on the origin device, which would normally result in higher latencies for future requests. We set up DPWC with `dpwc-test` configured as a small (1 MB) origin device. Then, we used a modified version of `verificator` which overwrites whole intervals of bytes in DPWC. After a time, the modified version also clears the page cache and validates the written data. The delays introduced by `dpwc-test` resulted in many conflicts in the writeback module. In the end, all of the stored data was correct.

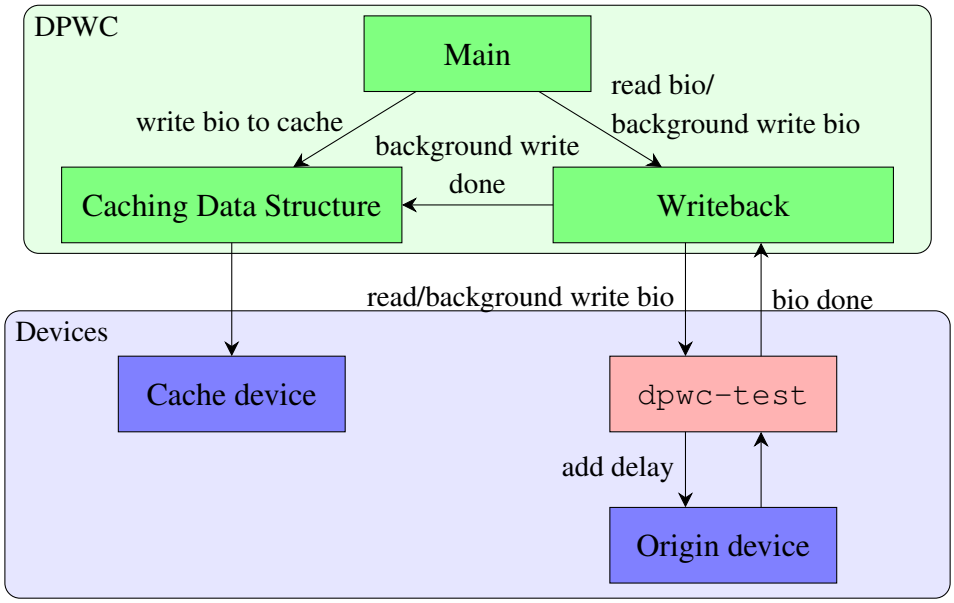

Figure 5.1: Devices used in the `dpwc-test` configuration.

Lastly, we ran xfstests [52]. "xfstests is used as a file system regression test suite for all of Linux's major file systems: xfs, ext2, ext4, cifs, btrfs, f2fs, reiserfs, gfs, jfs, udf, nfs, and tmpfs" [49]. Even though xfstests is aimed towards testing

file systems, many of the tests also require interaction with two block devices - `TEST` and `SCRATCH` devices. We ran all xfstests which belong to the quick group and work with the EXT4 and XFS file systems. These tests were run twice. The first time the `TEST` device was backed by DPWC, and the `SCRATCH` device was a regular partition on our SSD. The second time we switched the places of the two devices. We did not discover any regressions during the tests.

5.2 Performance

After convincing ourselves that DPWC works reasonably correctly, we proceeded to measure its performance in different scenarios.

As a first step, we measured the performance of the Optane module with fio, the Flexible I/O tester [48]. We used the `libpmem` engine, which is specially optimized for PMEM. The results show the maximum bandwidth that can be achieved is about 1600 MB/s with 3 or 4 parallel writer processes. This corresponds to 400k IOPS. We observed a slight degradation (1500 MB/s) for a higher number of writer processes. These numbers serve as an upper limit and a target for DPWC.

Next, we tested DPWC and several other configurations so that we can get a better overview of how DPWC stacks against them. The configurations are the following:

SSD In this configuration, we used the Micron SSD device as the only storage device. This configuration serves as a baseline for the other configurations.

dm-writecache Here we used dm-writecache included with the distribution's kernel build. We used it with its default settings and a 4 KB block size. During our benchmarks, the working set remained below dm-writecache's `low_watermark` parameter. Because of that, dm-writecache does not write back data to the origin device. This ensures a fair comparison with DPWC, as the writeback would slow down dm-writecache and is meant to serve as a throttling mechanism in case the cache device becomes full.

dpwc This configuration uses our own module with 8 writer slots, 4 generations and 16 chunks per generation.

dpwc-4s Same as dpwc, but with only 4 writer slots. According to our previous measurements, this is the optimal number of concurrent writers for Intel Optane memory.

pmem Lastly, we included the performance measurements when the Intel Optane module is used as the primary storage medium. This configuration usually has performance similar to the upper limits we have already established.

For each configuration, we used an EXT4 file system with default settings on top of the corresponding (virtual) block device.

A notable omission in the previous list is ZIL-PMEM. Even though ZIL-PMEM is a closely related project, we found that its performance depends on the speed of the underlying main storage device, even when a PMEM log device is used. This is why the ZIL-PMEM benchmarks in the original paper [40] use a striped configuration with 3 SSDs. However, both dm-writecache's and DPWC's write speed-ups are independent of the underlying origin device. Therefore, we believe that a direct comparison with ZIL-PMEM would not be suitable in our context.

5.2.1 Scalability

The main goal of DPWC is improving the performance in write-dominated multi-threaded workloads. Therefore, our first benchmarks exactly simulate such scenarios. For all of the aforementioned configurations, we ran a fio benchmark with the same options. Each writer process writes a total of 1GB random data in its own file in the filesystem. As the maximum amount of writers in our configuration was 16, this ensured that the total amount of written data (max. 16 GB) was below the writeback limit for dm-writecache and could fit in its entirety in the CDS of DPWC. Therefore, these numbers represent the peak bandwidth of these modules without any writeback in the background.

We use the `sync` fio engine and configure it so that data is `fsync()`-ed after every operation. This effectively makes every write operation synchronous and ensures that the performance of the block device is actually measured. The workload was tested with different number of writer processes and I/O operation sizes (fio's `blocksize` option). A visual representation of the results can be seen in Figure 5.2.

We observe that in these tests, DPWC is usually between 37% and 105% faster than dm-writecache as long as we have multiple writer threads. This holds true regardless of the block size used. These speedups also reflect a corresponding reduction in the average I/O latency, as every request is effectively synchronous due to the `fsync()` call after each operation. For single-threaded workloads, DPWC's performance is within $[-16\%, +20\%]$ of dm-writecache's.

Nevertheless, DPWC fails to achieve the maximum PMEM bandwidth with smaller block sizes, whereas Ext4 on PMEM does. We collected data with perf [31] during a test run with `numjobs=8` and `blocksize=4K`. The data consists of

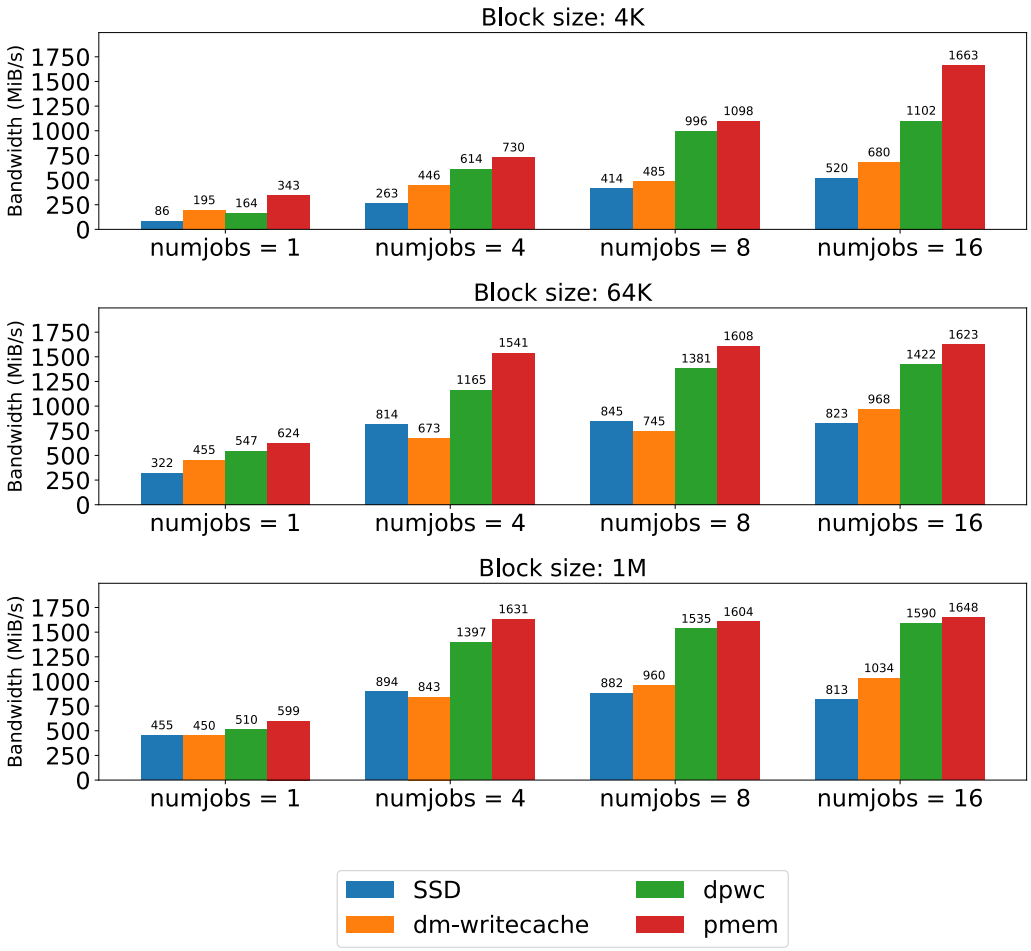

Figure 5.2: Total write bandwidth with N writer threads (`numjobs`=N). Block size indicates the size of each I/O operation.

periodic samples of the call stacks from the currently running thread on each CPU core. With enough data points, we can approximate the amount of time spent in each function. The results can be seen in Figure 5.3. We found out that only 30% of the time spent for handling an `fsync()` system call is spent in our module. This includes the time spent copying data to PMEM. We also configured a Device Mapper `dm-zero` device, which acts like `/dev/zero`: it discards write requests and always reads zeroes. This effectively means this configuration is only limited by the software overhead and not by a physical storage device. We observed a peak bandwidth of 1300 MB/s when writing to a `dm-zero` device, which corresponds to 330k IOPS. In comparison, the already established maximum Optane bandwidth is 1600 MB/s, or 400k IOPS. This strongly suggests that the Device Mapper subsystem in the Linux kernel incurs significant overhead and that we cannot utilize the full Intel Optane capabilities with small requests without further optimizations in the Linux kernel.

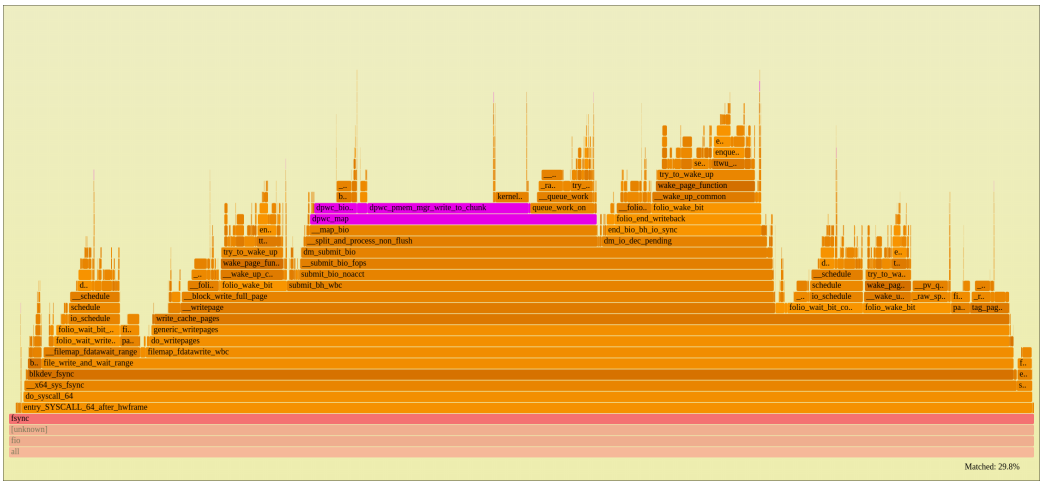

Figure 5.3: A cut-out of the flamegraph [15] generated with data from `perf`. We ran a benchmark with DPWC and 8 parallel writers. The highlighted part is the only occurrence of DPWC in the path handling `fsync()`.

With `numjobs >= 8` and `blocksize=1M` DPWC manages to reach the target maximum bandwidth. In these configurations the CPU overhead is mitigated by a good mixture of CPU and I/O work. We can also see that limiting the number of concurrent writers to only 4 slots does not help performance in most of the highly-threaded benchmarks. This is due to the fact that a 4-slot limitation means that more writer processes block while waiting for a slot, and as we stated previously, the actual bottleneck in may of these benchmarks is the CPU.

dm-writecache appears to achieve a better speedup in single-threaded workloads with `blocksize=4K`. In single-threaded cases, it does not suffer from

contention and its global locking strategy incurs very limited overhead. In contrast, DPWC's slot system and the queue between the main and the writeback modules are designed to work well for multi-threaded access, but incur higher overhead when such access is not necessary. With increasing block size, the management of requests becomes less important and the data transfer becomes the dominant cost. In these cases (`blocksize >= 64K`) we see that our design with 256-byte data alignment and efficient copying with AVX-512 non-temporal store instructions does pay off and DPWC is faster than dm-writecache by 10-20%.

5.2.2 Sustained writes

Another interesting metric is the bandwidth under sustained write load. We ran the benchmark with 16 writer threads and 4 K block size for the DPWC, dm-writecache and SSD configurations for 2 minutes and recorded the bandwidth for each 0.5 second interval. We also limited the PMEM capacity to 10 GB for DPWC and dm-writecache. All in all, this resulted in writing at least 60 GB of data in all of the benchmarks, i.e. the data written exceeds the amount of cache space by a factor of at least 6. The results are plotted on Figure 5.4.

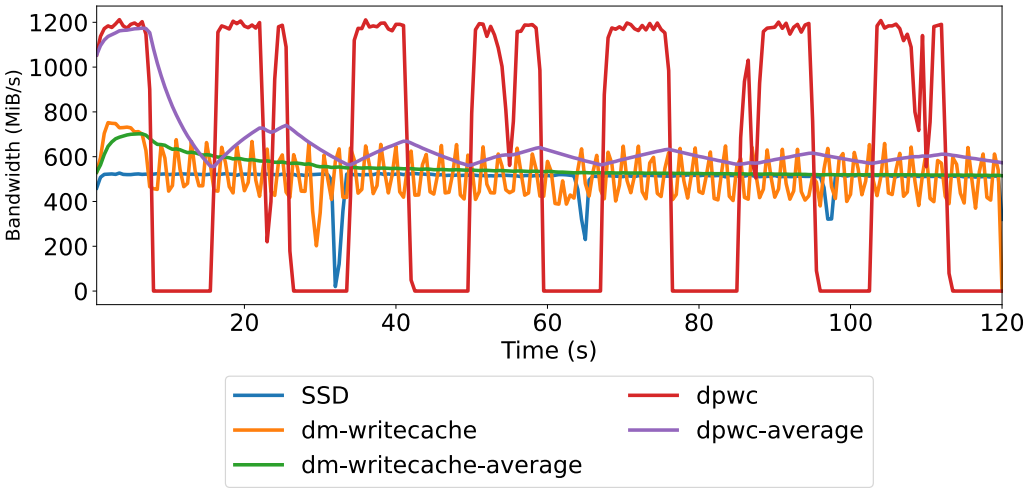

Figure 5.4: Total write bandwidth timeline for all writer threads in different configurations. dpwc-average and dm-writecache-average represent the average bandwidth up to the given point in time.

The behavior of dm-writecache is rather predictable. In the beginning it is caching data without writeback until the `low_watermark` threshold is reached. After that we observe fluctuations around the average bandwidth, representing

short periods during which writeback was happening, and periods without writeback.

DPWC's behavior shows extreme fluctuations due to the lack of any throttling mechanisms. Similarly to dm-writecache, we observe a period of peak bandwidth until the cache becomes full. After that, the bandwidth drops to zero, as any further requests are directly enqueued in the WBM. As the WBM processes requests sequentially, this means all writers block until the WBM has processed all queued requests. At that point, some of the generations are usually completely written back and therefore available for use in new requests. The result is a new period of peak bandwidth until the cache becomes full again and then the sequence repeats.

The length of the initial peak bandwidth period for dm-writecache and DPWC is almost the same. This is due to the fact that DPWC writes until the cache becomes full. In contrast, dm-writecache with the default settings starts throttling when the cache usage reaches 45%. However, DPWC's peak bandwidth is almost double that of dm-writecache (1200MB/s vs 700MB/s), which balances out the fact that DPWC uses the full PMEM space available.

Interestingly, as there were no conflicting write requests during this benchmark, DPWC's strategy of submitting multiple bios in parallel seems to have an effect on the average bandwidth. It remained slightly above (569 MB/s) the average bandwidth of the SSD (508 MB/s) and of dm-writecache (517 MB/s).

5.2.3 Impact of read requests

In practice, even write-dominated workloads often need a certain amount of read operations. Therefore, we measured the performance of DPWC with different mixes of read and write operations of size 4 KB. The results are displayed on Figure 5.5.

DPWC appears to be between 12% and 37% slower than dm-writecache for most workloads, and up to 50% slower in highly multi-threaded (16 writers) mixed workloads. The only outliers are the cases with 99% write requests and 4 or 8 writers, where DPWC is slightly faster. The SSD configuration also delivered better performance than DPWC and even dm-writecache in multiple test cases.

These numbers shows the main weakness of DPWC. The main module and the CDS are optimized for multi-threaded write requests. However, read requests suffer from higher latencies as they spend time in the queue between the main and the writeback modules and in the writeback module itself, especially since it processes every request sequentially. This incurs a high performance penalty. The SSD-only configuration, in contrast, is able to handle parallel read and write requests in any order. There are also few conflicts between different requests in this benchmark. Thus, very little time is spent waiting for a request to complete before the next one is submitted.

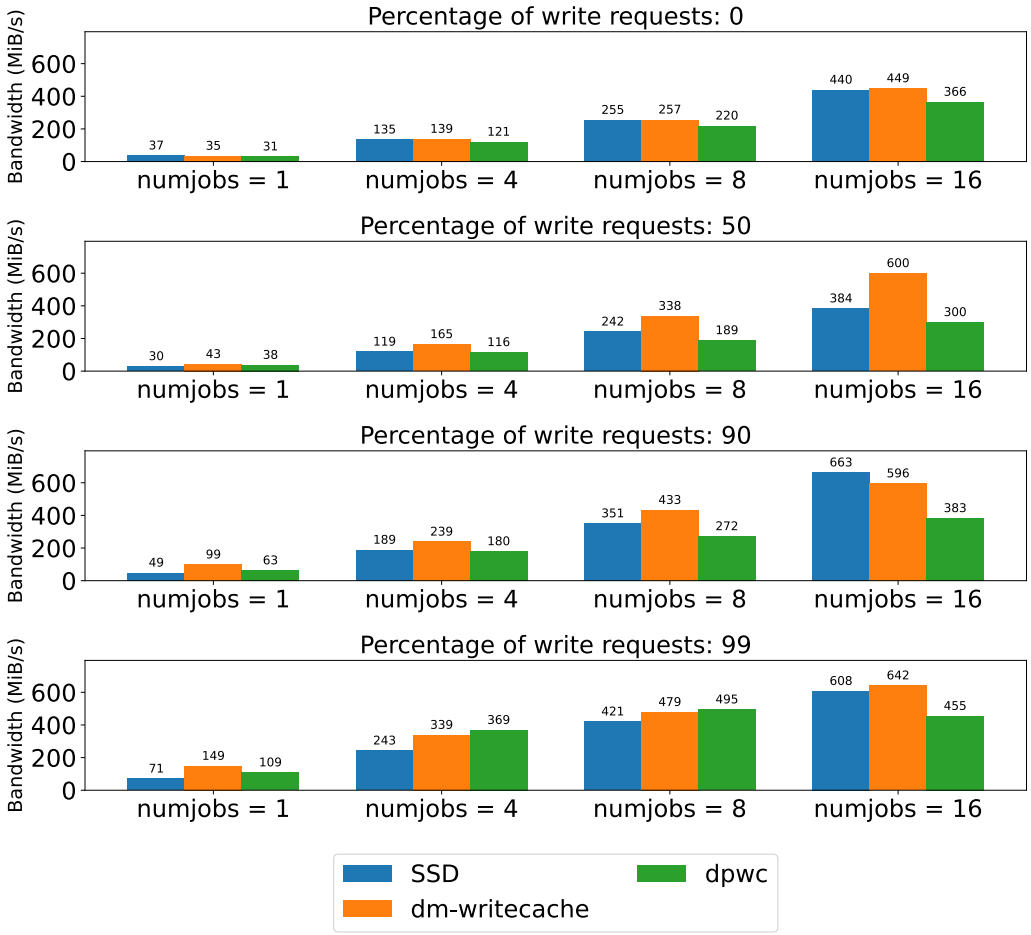

Figure 5.5: Total read+write bandwidth with N workers (`numjobs`=N). The PMEM configuration was left out as it outperforms all other configurations by a large margin.

Lastly, our writeback strategy involves sending write requests to the origin device as soon as possible. This means that even in write-dominated workloads (for example the 99% write workload), the origin device is constantly under pressure. Thus, it is unable to process the read requests in a timely manner, resulting in even higher read latencies for the DPWC configuration.

5.2.4 Database performance

We also evaluated the performance of our storage configurations with the following database benchmarks:

Redis is an "open source, in-memory data store used by millions of developers as a database, cache, streaming engine, and message broker." [37].

We used Redis server v6.2.6 from the distribution's package manager. By default, Redis is an in-memory data store but can be configured to persist data in two ways. The Redis Database (RDB) mechanism stores point-in-time periodic snapshots of the dataset. The Append Only File (AOF) logs every write operation and can be used to replay operations at startup. The combination of RDB and AOF makes Redis a persistent database. We enabled both of these features and configured Redis to call `fsync()` after every write. The `io-threads` option was also set to 6. This is the recommended value for 8-core systems. With this option, the Redis server is able to process multiple incoming requests from different clients in parallel.

For the benchmarks, we used the standard redis-benchmark utility [38]. We ran 2 M `SET` commands 5 times for all configurations with varying number of client threads, and recorded the average operations per second as reported by the redis-benchmark program. The results are displayed on Figure 5.6. We observed moderate improvements in the range 12-34% compared to dm-writecache. PMEM remains the fastest storage configuration except for the single-threaded case, where DPWC comes out the fastest.

Even though Redis handles client requests in multiple threads, we observed that write operations in the CDS always used the first slot. We also observed only a minimal amount of read requests. In other words, this workload can be characterized as single-threaded and write-only from the perspective of the CDS, and therefore DPWC does not provide the significant boost seen previously with multi-threaded write workloads. The improvements in this case can be mostly be attributed to the offloading of submission of bio requests to a different thread, which is the main difference with all of the other configurations.

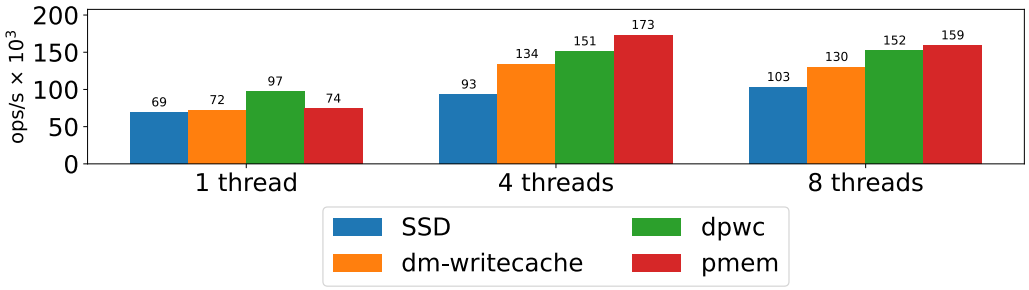

Figure 5.6: Operations per second in the redis-set benchmark for different number of client threads. Note that the values are plotted in thousand operations per second.

RocksDB "RocksDB is a persistent key-value store. It is optimized for fast, low latency storage such as flash drivers and high-speed disk drivers. RocksDB exploits the full potential of high read/write rates offered by flash or RAM" [39]. These qualities make RocksDB an interesting candidate for testing with DPWC, as our goal is to emulate the performance of such a device.

We used RocksDB 7.0.3-33f8a08a built from source. For benchmarking, the built-in benchmarking tool `db_bench` was used. We chose the `fillsync` benchmark, which "writes values in random key order in sync mode" [5]. All parameters were left to their default values, except the database path, which was set to a directory in the corresponding storage stack for each configuration. We also varied the amount of writer threads.

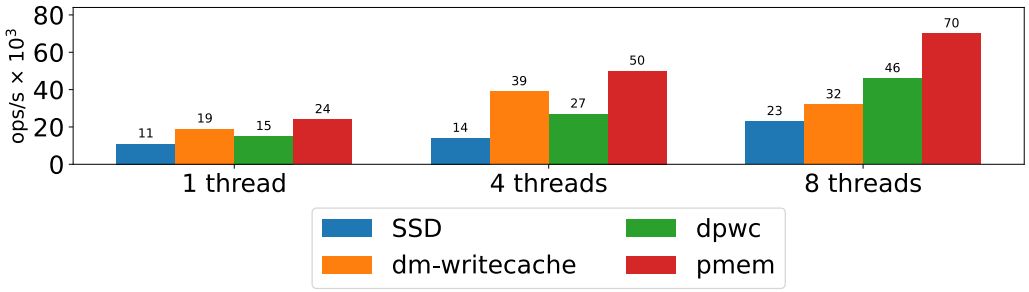

Figure 5.7: Operations per second in the RocksDB fillsync benchmark for different number of inserter threads. Note that the values are plotted in thousand operations per second.

Similarly to the Redis benchmark, we found that this scenario can also be described as write-only and single-threaded I/O workload. As can be seen in Figure 5.7, DPWC is only able to provide a 43% speedup compared to dm-

writecache when the degree of multi-threading is sufficiently high (8 writer threads configuration). We believe that this is again due to the fact that with our design, bio requests can be submitted asynchronously and even on other CPU cores, which seems beneficial for performance when the CPU utilization is higher. In the cases with fewer threads, DPWC was between 22% and 31% slower than dm-writecache in this benchmark. This matches our observations from the synthetic FIO workloads for similar single-threaded write workloads.

MariaDB "MariaDB Server is one of the most popular open source relational databases." [24] We used the MariaDB 10.5.13 version which comes with the package manager on Fedora 35. All settings were left to default. The `/usr/lib/mysql` directory was bind-mounted to a directory in the corresponding storage configuration for each test.

For driving the benchmark we used sysbench, "a scriptable multi-threaded benchmark tool based on LuaJIT" [20]. We ran it in the `oltp_insert` configuration, which simulates database insertions with multiple parallel threads. As usual, we used the default settings, prewarmed the database before each test run and varied the number of inserter threads. We ran each test 5 times and recorded the average operations per second as reported by sysbench. The results are displayed on Figure 5.8.

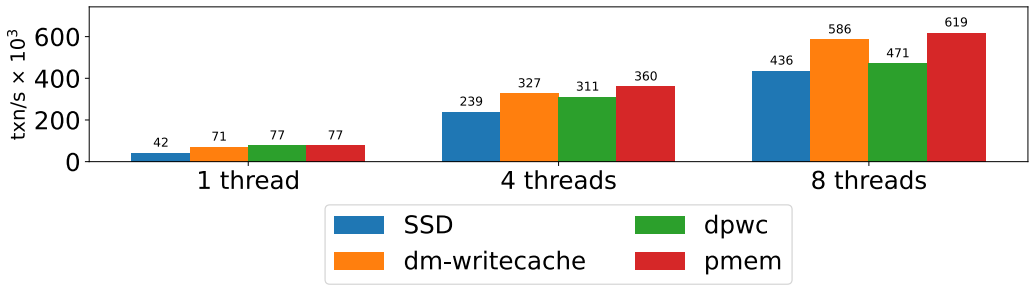

Figure 5.8: Operations per second in the MariaDB/sysbench `oltp_insert` benchmark for different number of inserter threads. Note that the values are plotted in thousand operations per second.

MariaDB/sysbench is the only benchmark where we observed actual parallel writes, even though the first slot in the CDS was used more than 99% of the time. We also observed high variations in the results in the single thread configuration, despite averaging results. Nevertheless, DPWC was always 5-20% slower than dm-writecache in the multi-threaded cases, but was still faster than the raw SSD configuration by 10-30%. There results look similar to the fio benchmarks with one writer thread from the beginning of this

chapter. We conclude that MariaDB does not benefit from our strategy of delayed bio submission, as its write patterns are essentially single-threaded despite appearances.

5.2.5 Analysis and Experiments

Motivated by the rather poor performance of DPWC in some of the benchmarks above, we experimented with slight variations of our module and writeback strategy. Our goal was to explore possible performance trade-offs between peak multi-threaded write performance and all other cases.

We collected perf data from the workload with 1 writer and block size 4 KB. We saw that up to 42% of the time spent in `dpwc_map`, the function in our module which handles all incoming requests, is taken up by `queue_item()`. This function is called when adding a new request to the synchronization queue between the main and the writeback modules. This seems to indicate that using workqueues may not be the most efficient solution for this problem.

However, our benchmarks indicate that read performance is even more of an issue than single-threaded write performance. The key insight for solving this problem is that read requests are often stuck waiting for write requests to be processed in the WBM. However, read requests very rarely need this, as they usually do not induce conflicts. Conflicting read requests would mean that the data to be read was recently written. That data is, however, typically available in the page cache already.

Keeping this in mind, we wrote a prototype reimplementation of the WBM with the following changes:

- Instead of using a workqueue between the main and the writeback modules, the main module calls directly into the WBM. This change reduces the peak write bandwidth, as the write request handler needs to do more work instead of just enqueueing the request for the WBM. However, this also removes the need for the ordered workqueue between the two modules and reduces latencies for read requests.

- If we detect a conflict, we place the request on a dedicated ordered workqueue for conflicting requests. This operation is rare since most requests should never block. By moving such requests to their own workqueue, we enable the processing of further requests instead of blocking as in our default implementation. Non-conflicting reads and writes are added to the interval tree and then placed on *separate* workqueues to be submitted to the origin device.

- By using a simple `usleep_idle_range()` call in the write submission workqueue, we throttle write requests sent to the SSD. Read requests, on the other hand, are sent to the SSD as fast as possible. This measure avoids excessive requests to the origin device and ensures that read requests are serviced sooner.

  Note that the throttling of write requests may mean that a conflicting read request will be stalled for a long time. As stated previously, conflicting reads are a very rare event. For example, they did not occur at all during our benchmarks, as the working set could fit into the main memory.

We also tried varying the multi-core policy for the blocking and conflicting workqueues. As mentioned previously in Section 4.4, workqueues can be either bound or unbound [9]. We tried both configurations in our fio benchmarks and plotted the results, which can be seen on Figure 5.9. The modified writeback strategy does indeed lower the maximum write bandwidth, as predicted. On the upside, we saw a positive change in mixed and read-only configurations. Unfortunately, none of the configurations could improve performance across the board.

The main takeaway from this experiment is that our writeback module can be significantly improved. The tests seem to indicate that the usage of workqueues has a non-trivial overhead. We believe that a strategy similar do dm-writecache's could solve this issue. Instead of writing data immediately, the WBM should write back whole generations at once, when the cache usage reaches a particular threshold, like dm-writecache's `low_watermark`. Read requests can be serviced from the SSD or by reading from the entries in the CDS, if the data is already cached.

This strategy can potentially combine some of the benefits of DPWC and dm-writecache. By avoiding writes when the cache is not full, we avoid the workqueue overhead and reads can be submitted without delays. On the other hand, parallel writes to the CDS remain possible. Unfortunately, we could not test this version due to limited scope of this bachelor thesis.

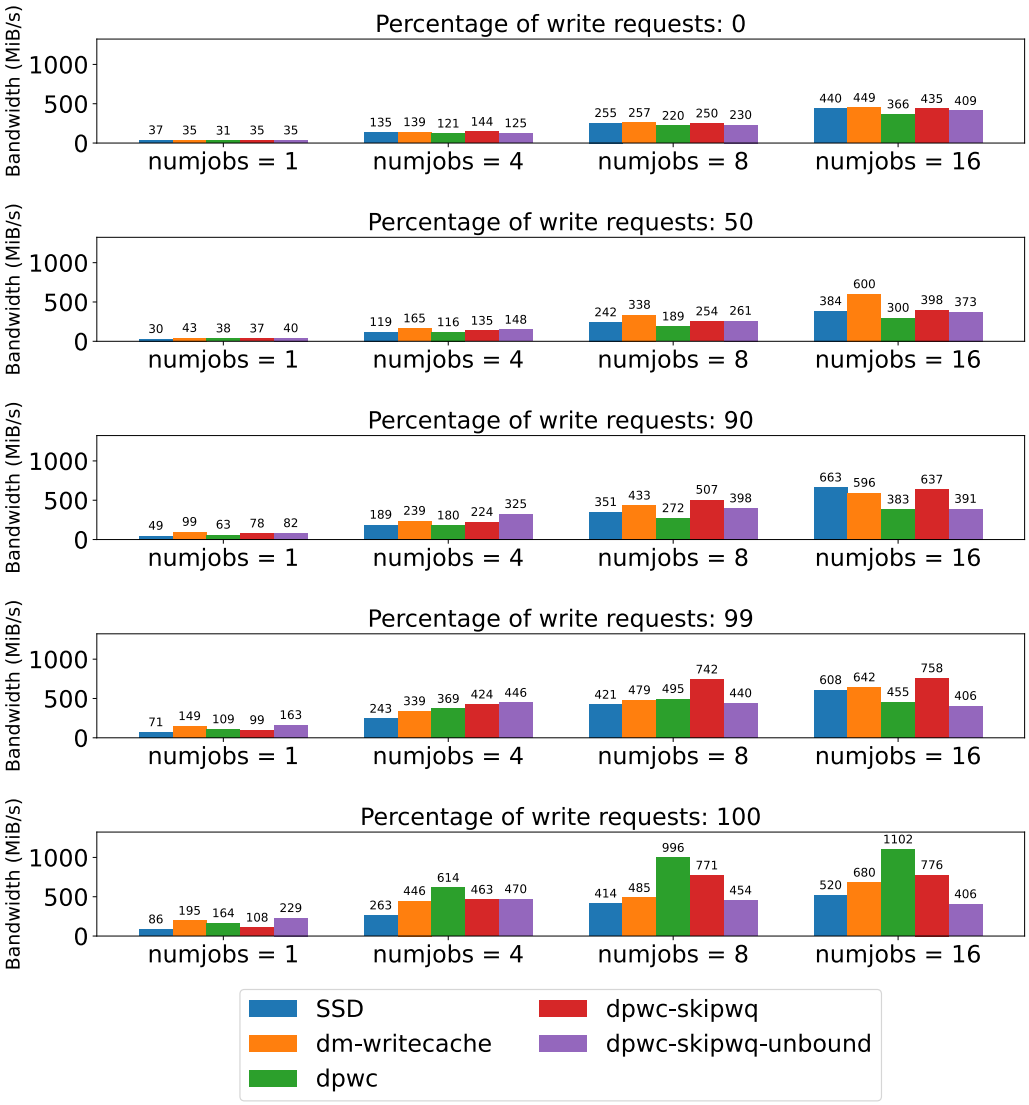

Figure 5.9: Total read+write bandwidth with N workers (`numjobs`=N) and different read/write mixes. dpwc is our base implementation. dpwc-skipwq incorporates the changes for our experiment in Section 5.2.5 with bound workqueues. dpwc-skipwq-unbound is the same, but with unbound workqueues.

Chapter 6

Conclusion

In this bachelor thesis we showed that the existing Linux kernel infrastructure, and more specifically dm-writecache, is often unable to take full advantage of the high-performance Intel Optane devices. This is partly due to using less efficient data transfer routines and also due to the usage of global locks, which limits performance in multi-threaded workloads.

We designed and implemented our own alternative, DPWC. Despite certain inefficiencies in the software stack, DPWC is a definite improvement over the status quo in pure write workloads, providing a significant (up to 2.05x) speedup over dm-writecache. The key to achieving this result is the caching data structure inspired by ZIL-PMEM [40], which allows efficient parallel writes in the cache, and the asynchronous writeback module.

Unfortunately, in optimizing for a specific case, our module fails to hold up to the competition in many other benchmarks. Analysis and experiments show that in practice, the performance in a particular workload depends on many factors and the specific storage access patterns of the application. This makes it very difficult to find a one-size-fits-it-all solution. Nevertheless, we believe that the approach used in DPWC is a viable option for further developments and can be used in real-life scenarios.

6.1 Future Work

During the course of this thesis, we have also identified multiple areas where DPWC can be improved. There are also many areas of the design which can be tweaked and potentially boost performance or improve failure recovery in many cases. Here are some of the directions which, in our opinion, are worth pursuing:

Writeback Strategy As suggested in Section 5.2.5, an alternative writeback strategy for DPWC may bring significant advantages in mixed workloads. We

would like to test out a variant of DPWC where data is written back only when the cache is (nearly) full, and conflicting read requests are serviced from the cache.

Tighter integration with the origin device In Section 5.2.5, we saw that throttling write requests to the origin device helps with read performance in mixed workloads. As an alternative to changing the writeback strategy, we might be able to integrate better with multi-queue NVMe SSDs and use dedicated queues for read requests. We would also like to explore the possibility of cooperating with the lower block I/O layers to prioritize those requests.

Throttling DPWC's performance under sustained writes has extreme fluctuations. It would be interesting to implement a throttling mechanism, potentially similar to dm-writecache's, and measure its influence on the average bandwidth.

Better configurability and error handling To make DPWC an actually viable project in real-life scenarios, it should be extended with more options to control the various aspects of the CDS (number of generations, chunks and slots). In addition, DPWC should be able to better handle software and medium errors as described in Section 4.5.

Aside from work on DPWC, we would have found the following items interesting in the context of this bachelor thesis:

Comparison with ZIL-PMEM and striped configurations ZIL-PMEM uses a very similar cache data structure. Therefore, a comparison with it may be able to show the differences in the overhead between ZFS and Device Mapper. ZIL-PMEM also achieves its maximal performance in a striped origin device configuration. Evaluating DPWC on that setup will also show how the performance of the origin device affects the overall performance of our module.

Investigate the overhead of Linux kernel interfaces In Sections 5.2.1 and 5.2.5, we have mentioned that our initial measurements show that the Device Mapper subsystem incurs a significant overhead. The same appears to be true for workqueues. It would be interesting to explore what exactly causes these issues and how these interfaces might be optimized for our particular scenario.

Bibliography

[1] URL: https://www.kernel.org/doc/Documentation/vfio.txt (visited on 03/21/2022).

[2] *15.1. The Page Cache - Understanding the Linux Kernel, 3rd Edition [Book]*. URL: https://www.oreilly.com/library/view/understanding-the-linux/0596005652/ch15s01.html (visited on 03/13/2022).

[3] *A Block Layer Introduction Part 1: The Bio Layer [LWN.Net]*. URL: https://lwn.net/Articles/736534/ (visited on 03/31/2022).

[4] *Appendix A. The Device Mapper Red Hat Enterprise Linux 6*. Red Hat Customer Portal. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/logical_volume_manager_administration/device_mapper (visited on 03/31/2022).

[5] *Benchmarking Tools · Facebook/Rocksdb Wiki*. GitHub. URL: https://github.com/facebook/rocksdb (visited on 03/28/2022).

[6] Jeff Bonwick et al. *The Zettabyte File System*. 2003. URL: https://www.semanticscholar.org/paper/The-Zettabyte-File-System-Bonwick-Ahrens/27f81148ecbcd04dd97cebd717c8921e5f2a4373 (visited on 03/31/2022).

[7] Hadi Brais. *The Significance of the X86 SFENCE Instruction*. Micromysteries. Feb. 25, 2019. URL: https://hadibrais.wordpress.com/2019/02/26/the-significance-of-the-x86-sfence-instruction/ (visited on 11/03/2022).

[8] *Cache (Computing)*. In: *Wikipedia*. Feb. 13, 2022. URL: https://en.wikipedia.org/w/index.php?title=Cache_(computing)&oldid=1071635269 (visited on 03/19/2022).

[9] *Concurrency Managed Workqueue (Cmwq) - The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/v4.14/core-api/workqueue.html (visited on 03/26/2022).

[10] *DebugFS — The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/latest/filesystems/debugfs.html (visited on 03/21/2022).

[11] *Dm-Raid — The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/dm-raid.html (visited on 03/31/2022).

[12] *Dm-Zero — The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/zero.html (visited on 03/31/2022).

[13] *Explicit Volatile Write Back Cache Control*. URL: https://www.kernel.org/doc/Documentation/block/writeback_cache_control.txt (visited on 03/29/2022).

[14] N. V. M. Express. *NVMe Command Set Specifications*. NVM Express. URL: https://nvmexpress.org/developers/nvme-command-set-specifications/ (visited on 03/31/2022).

[15] *Flame Graphs*. URL: https://www.brendangregg.com/flamegraphs.html (visited on 04/02/2022).

[16] *Head-of-Line Blocking*. In: *Wikipedia*. Mar. 2, 2022. URL: https://en.wikipedia.org/w/index.php?title=Head-of-line_blocking&oldid=1074778472 (visited on 03/19/2022).

[17] Satoshi Imamura and Eiji Yoshida. "The Analysis of Inter-Process Interference on a Hybrid Memory System." In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region Workshops*. HPCAsia2020: Workshops of HPCAsia. Fukuoka Japan: ACM, Jan. 15, 2020, pp. 1–4. ISBN: 978-1-4503-7650-1. DOI: 10.1145/3373271.3373272. URL: https://dl.acm.org/doi/10.1145/3373271.3373272 (visited on 03/31/2022).

[18] "Intel® Optane™ Technology: Memory or Storage?" In: (), p. 5.

[19] *Kernel/Reference/IOSchedulers - Ubuntu Wiki*. URL: https://wiki.ubuntu.com/Kernel/Reference/IOSchedulers (visited on 03/31/2022).

[20] Alexey Kopytov. *Sysbench*. Mar. 29, 2022. URL: https://github.com/akopytov/sysbench (visited on 03/29/2022).

[21] Youngjin Kwon et al. “Strata: A Cross Media File System.” In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP ’17. New York, NY, USA: Association for Computing Machinery, Oct. 14, 2017, pp. 460–477. ISBN: 978-1-4503-5085-3. DOI: 10.1145/3132747.3132770. URL: https://doi.org/10.1145/3132747.3132770 (visited on 03/31/2022).

[22] *Linux Storage Stack Diagram - Thomas-Krenn-Wiki*. URL: https://www.thomas-krenn.com/en/wiki/Linux_Storage_Stack_Diagram (visited on 03/31/2022).

[23] Ziyi Lu and Qiang Cao. “A Case Study of Migrating RocksDB on Intel Optane Persistent Memory.” In: *2021 IEEE International Conference on Networking, Architecture and Storage (NAS)*. 2021 IEEE International Conference on Networking, Architecture and Storage (NAS). Oct. 2021, pp. 1–4. DOI: 10.1109/NAS51552.2021.9605438.

[24] *MariaDB Foundation*. MariaDB.org. URL: https://mariadb.org/ (visited on 03/28/2022).

[25] Ian Neal et al. “AGAMOTTO: How Persistent Is Your Persistent Memory Application?” In: (), p. 19.

[26] *NOVA: NOn-Volatile Memory Accelerated Log-Structured File System*. The Non-Volatile Systems Lab, Mar. 27, 2022. URL: https://github.com/NVSL/linux-nova (visited on 03/31/2022).

[27] “NVM Express Explained.” In: (), p. 5.

[28] *Open(2) - Linux Manual Page*. URL: https://man7.org/linux/man-pages/man2/open.2.html (visited on 03/31/2022).

[29] *Operating Systems: Three Easy Pieces*. URL: https://pages.cs.wisc.edu/~remzi/OSTEP/ (visited on 03/29/2022).

[30] *Overview of the Linux Virtual File System — The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/latest/filesystems/vfs.html (visited on 03/31/2022).

[31] *Perf Wiki*. URL: https://perf.wiki.kernel.org/index.php/Main_Page (visited on 03/25/2022).

[32] PMem.io. *PMem.Io*. URL: https://pmem.io/ (visited on 03/29/2022).

[33] *Predefined Personalities · Filebench/Filebench Wiki*. GitHub. URL: https://github.com/filebench/filebench (visited on 03/27/2022).

[34] *Problame/Zfs at Zil-Pmem/Upstreaming*. GitHub. URL: https://github.com/problame/zfs (visited on 03/23/2022).

[35] Paul Alcorn published. *Intel Optane DIMM Pricing: $695 for 128GB, $2595 for 256GB, $7816 for 512GB (Update)*. Tom's Hardware. Apr. 7, 2019. URL: https://www.tomshardware.com/news/intel-optane-dimm-pricing-performance,39007.html (visited on 03/30/2022).

[36] Sean Webster published. *Intel Optane SSD DC P5800X Review: The Fastest SSD Ever Made*. Tom's Hardware. July 2, 2021. URL: https://www.tomshardware.com/reviews/intel-optane-ssd-dc-p5800x-review (visited on 03/30/2022).

[37] *Redis*. Redis. URL: https://redis.io/ (visited on 03/28/2022).

[38] *Redis Benchmark*. Redis. URL: https://redis.io/docs/reference/optimization/benchmarks/ (visited on 03/28/2022).

[39] *RocksDB | A Persistent Key-Value Store*. RocksDB. URL: http://rocksdb.org/ (visited on 03/28/2022).

[40] Christian Schwarz. *Low-Latency Synchronous IO for OpenZFS Using Persistent Memory*. Master thesis. 2021–607.

[41] servethehome. *What Is the ZFS ZIL SLOG and What Makes a Good One*. ServeTheHome. Nov. 12, 2017. URL: https://www.servethehome.com/what-is-the-zfs-zil-slog-and-what-makes-a-good-one/ (visited on 03/31/2022).

[42] Steven Swanson and Adrian M. Caulfield. "Refactor, Reduce, Recycle: Restructuring the I/O Stack for the Future of Storage." In: *Computer* 46.8 (Aug. 2013), pp. 52–59. ISSN: 1558-0814. DOI: 10.1109/MC.2013.222.

[43] *The End of Block Barriers [LWN.Net]*. URL: https://lwn.net/Articles/400541/ (visited on 03/29/2022).

[44] *The Open Group Base Specifications Issue 7, 2018 Edition*. URL: https://pubs.opengroup.org/onlinepubs/9699919799/ (visited on 03/29/2022).

[45] *Understanding Caching | Linux Journal*. URL: https://www.linuxjournal.com/article/7105 (visited on 03/19/2022).

[46] *Using QEMU Virtualization*. URL: https://docs.pmem.io/persistent-memory/getting-started-guide/creating-development-environments/virtualization/qemu (visited on 03/21/2022).

[47] Alexander van Renen et al. "Building Blocks for Persistent Memory." In: *The VLDB Journal* 29.6 (Nov. 1, 2020), pp. 1223–1241. ISSN: 0949-877X. DOI: 10.1007/s00778-020-00622-9. URL: https://doi.org/10.1007/s00778-020-00622-9 (visited on 03/31/2022).

[48] *Welcome to FIO's Documentation! - Fio 3.27-169-G1953e1-Dirty Documentation*. URL: https://fio.readthedocs.io/en/latest/ (visited on 03/23/2022).

[49] *What Is Xfstests?* URL: https://kernel.googlesource.com/pub/scm/fs/ext2/xfstests-bld/+/HEAD/Documentation/what-is-xfstests.md (visited on 03/23/2022).

[50] *Writecache Target - The Linux Kernel Documentation*. URL: https://www.kernel.org/doc/html/latest/admin-guide/device-mapper/writecache.html (visited on 03/19/2022).

[51] Yinjun Wu et al. "Lessons Learned from the Early Performance Evaluation of Intel Optane DC Persistent Memory in DBMS." May 15, 2020. arXiv: 2005.07658 [cs]. URL: http://arxiv.org/abs/2005.07658 (visited on 03/31/2022).

[52] *Xfs/Xfstests-Dev.Git - XFSQA Testsuite*. URL: https://git.kernel.org/pub/scm/fs/xfs/xfstests-dev.git/ (visited on 03/23/2022).

[53] Jian Xu and Steven Swanson. "{NOVA}: A Log-structured File System for Hybrid {Volatile/Non-volatile} Main Memories." In: 14th USENIX Conference on File and Storage Technologies (FAST 16). 2016, pp. 323–338. ISBN: 978-1-931971-28-7. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu (visited on 03/29/2022).

[54] Jian Yang et al. "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory." Aug. 9, 2019. arXiv: 1908.03583 [cs]. URL: http://arxiv.org/abs/1908.03583 (visited on 03/14/2022).

[55] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. "Ziggurat: A Tiered File System for {Non-Volatile} Main Memories and Disks." In: 17th USENIX Conference on File and Storage Technologies (FAST 19). 2019, pp. 207–219. ISBN: 978-1-939133-09-0. URL: https://www.usenix.org/conference/fast19/presentation/zheng (visited on 03/31/2022).