Vrije Universiteit Amsterdam

Bachelor Thesis

Evaluating Performance Characteristics of the PMDK Persistent Memory Software Stack

Author: Nick-Andian Tehrany (2618819)

1st supervisor:	Dr. ir. Animesh Trivedi
daily supervisor:	Ir. Sacheendra Talluri
2nd reader:	Prof. dr. ir. Alexandru Iosup

A thesis submitted in fulfillment of the requirements for the VU Bachelor of Science degree in Computer Science

August 19, 2020

Abstract

Society nowadays revolves more and more around data. Data Science, Machine Learning, and Artificial Intelligence depend on large amounts of data to conduct research, and train machine learning models and agents. With the ever increasing amount of data, comes the need for faster storage. The quest for new storage devices has resulted in the development of non-volatile memories, that run alongside conventional memory and are directly accessible by the CPU, with large capacity of persistent storage. This new kind of non-volatile memory is shifting storage technology in a new direction, triggering numerous changes in the software stack on how we store and access data.

One such change is the addition of the direct access (DAX) extension for file systems. It allows using these devices as efficient storage devices with file systems. DAX eliminates obstacles that conventional file systems present to non-volatile memories. Another such change is the development of the Persistent Memory Development Kit (PMDK), a software infrastructure, consisting of a number of libraries and tools for using these devices as memories. The PMDK provides libraries for using non-volatile memory in a persistent way, maintaining data through power loss, and in a volatile way, losing data on power loss.

We present an evaluation of the performance of the DAX extension, and provide one of the first systematic studies analyzing PMDK software overheads. We conduct our study using emulated non-volatile memory on DRAM, and all benchmarks we designed and implemented, as well as function traces collected are open source. Our findings show that DAX bypassing the page cache increases performance for I/O bandwidth by up to 32.85%, showing that this new generation of file systems can outperform conventional file systems. Additionally, we identify that the average cost of persistence with the PMDK is 3x higher than volatile use, and provide guidelines for minimizing persistence software overheads.

Contents

1 Introduction

1.1 Context

Society nowadays revolves more and more around data. Businesses and companies collect large amounts of data on customers, and depend on such data to reach and expand their clientele. Data Science, Machine Learning, and Artificial Intelligence depend on large amounts of data to conduct research, and train machine learning models and agents. Predictions estimate that by 2025 the Global Datasphere will grow to 175 zettabytes of data [32]. With the ever increasing amount of data, comes the need for faster storage for the data. This quest for new storage devices has resulted in the development of non-volatile memories., delivering large persistent storage capacity, with low access latency.

Modern systems, such as servers and databases, can require hundreds of GBs of expensive main memory to deliver fast access to data. Since data is stored on slow storage devices, it needs to be copied into main memory before it can be accessed, introducing considerable overheads. After rebooting a server, it can sometimes require hours to "warm up the cache", by loading frequently used data into main memory. The advent of non-volatile memory technology aims to revolutionize today's and tomorrows' computer systems, by combining the best of memory- and storage- devices into one non-volatile memory device.

Figure 1 illustrates the placement of non-volatile memories within the memory hierarchy, comparing their capacities, access latencies, and bandwidths. Non-volatile memory falls between conventional DRAM and NAND SSD storage in access latency, capacity, bandwidth, and cost. This new kind of memory is shifting storage technology into a new direction, triggering numerous efforts of changing the software stack, and the way we store and access data.

Non-volatile memories, formally known as *non-volatile dual in-line memory modules* (NVDIMMs), and commonly referred to as *persistent memories*, experienced a recent technology breakthrough by Intel and Micron [22]. This technology, brings a new kind of memory to modern systems, combining large capacity persistent storage from conventional storage devices, with low latency byte-addressability from conventional memory, into one memory device. Non-volatile memories are running alongside conventional memory, as part of the main memory, with the CPU being able to directly access them. Unlike volatile memory, which cannot sustain data through power loss, non-volatile memory stores data when the power goes out.

With this new kind of memory, comes the need to change the software stack, to optimize how data is accessed and stored on them. The resulting software stack allows using these devices in two distinctive ways. Firstly, using them as storage devices with a file system. With conventional file systems, the issue arises that these are designed for slow storage devices, and therefore data is copied to main memory before being accessed, illustrated in Figure 2. The direct access (DAX) extension for file systems [10] eliminates this copying of data to main memory, and maps data directly into user space. Secondly, these devices can be used as memories, with the Persistent Memory Development Kit (PMDK) [23]. The PMDK is a software infrastructure consisting of numerous libraries and tools for using non-volatile memories as persistent memory, such as a persistent key-value store, or using it similar to conventional memory, as a volatile heap.

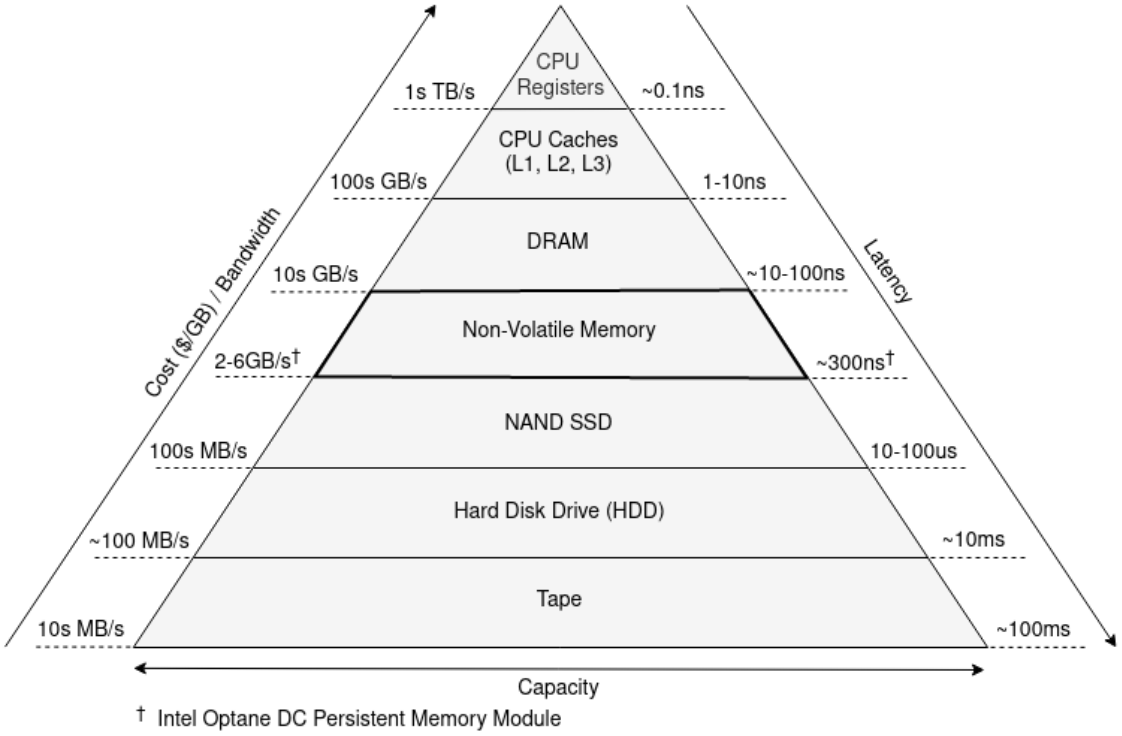

Figure 1: A pyramid of the memory hierarchy, showing the decreasing cost and increasing access latency, as capacity increases. Non-volatile memory falls between conventional DRAM and NAND SSD storage in access latency, capacity, bandwidth, and cost. Based on a figure from Intel [33].

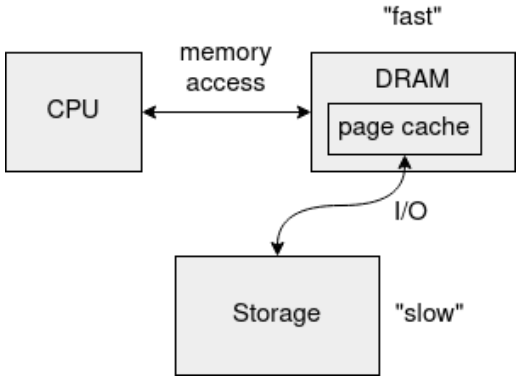

Figure 2: Before data can be accessed on slow storage devices, it has to be copied to the page cache in main memory. The page cache is transparent, and lets data be modified in memory, without modifying it on the storage device.

1.2 Problem Statement

With this newly developed technology slowly starting to become commercially available, there is little known about the performance implications of the PMDK software environment. Despite its libraries being the foundational building block for persistent memory applications, there has been no systematic study into what overheads they introduce. This thesis aims to provide a systematic evaluation of the performance characteristics throughout the persistent memory software stack, by evaluating the performance of the DAX extension for file systems, and identifying overheads of the PMDK libraries. To systematically evaluate the performance characteristics, we break the research objective into the following individual questions:

RQ1. How has the software stack changed with the DAX extension, how does its bypassing of the page cache affect performance, and what performance do these new generation file systems deliver? The aim is to build a foundational understanding of the new software stack, and understand how non-volatile memory interacts with it. This understanding will be important for our second research question.

RQ2. What design decisions were made during the development of the PMDK software environment, how did these decisions affect its performance, and what overheads are coming from the libraries themselves? The PMDK is the foundational building block for persistent memory applications, but so far there has not been a systematic study into its performance and software overheads. Answering these questions will be important for developers, building applications for persistent memory, to optimize performance and understand the software environment they are building with.

1.3 Main Contribution and Research Approach

This thesis presents one of the first systematic studies into the performance characteristics of the persistent memory software stack. We start by understanding the DAX extension for file systems, and identifying its performance. Next, we identify PMDK software overheads by measuring the costs of using persistent memory, compared to volatile memory, and identifying what this cost is made up of. The main contributions of this thesis are:

MC0. We provide an explanation of non-volatile memories, and the PMDK software infrastructure, its complexity and basic ideas. Due to limited documentation, we navigate the challenges of analyzing and understanding the source code and benchmarks, running them, and cross validating achieved results. All benchmarks we created are open source [36], and our function traces are publicly available [35].

MC1. An evaluation into this new generation of file systems, and understanding the DAX extension and its overheads. We show whether these new file systems should be used when employing persistent memories as storage devices.

MC2. We provide one of the early systematic studies of the PMDK and its software overheads, by understanding the software stack of the PMDK, and evaluating how its design decisions affect the performance. Additionally, we provide a set of guidelines for using the PMDK as the development tool for persistent memory applications, to optimize their performance by minimizing software overheads. Our study shows

that the average cost of persistence is 3x higher than volatile use. Additionally, we identified that keeping persistent data structures close together amortizes cache flushing overheads.

The research approach utilized the methodology of experimental research, by designing appropriate micro and workload-level benchmarks, and quantifying a running system prototype [25, 15, 30]. We utilize microbenchmarks for performance measuring of DAX-enabled file systems and the PMDK, as well as designing and implementing microbenchmarks for our evaluation.

Our findings show that DAX bypassing the page cache increases performance for I/O bandwidth by up to 32.85%, showing that this new generation of file systems can outperform conventional file systems. Our evaluation of PMDK software overheads reveals that the average cost of persistence is 3x higher than volatile use. Additionally, we identified that keeping persistent data structures close together amortizes cache flushing overheads.

1.4 Structure

This thesis is organized as follows: First, Section 2 provides background information of the hardware and software concepts used throughout this thesis. Next, the experimental setup and the baseline explained in Section 3. Section 4 then covers all benchmarks for the DAX extension, followed by Section 5 evaluating the PMDK software performance characteristics. The experiments are followed by an analysis of the results in Section 6, discussion in Section 7, and related work in Section 8. Lastly, an evaluation on current limitations and future work is covered in Section 9, followed by the conclusion in Section 10. Figure 3 provides a visual representation of the layout for this thesis, allowing readers the possibility of reading several sections in parallel or focusing only on one topic.

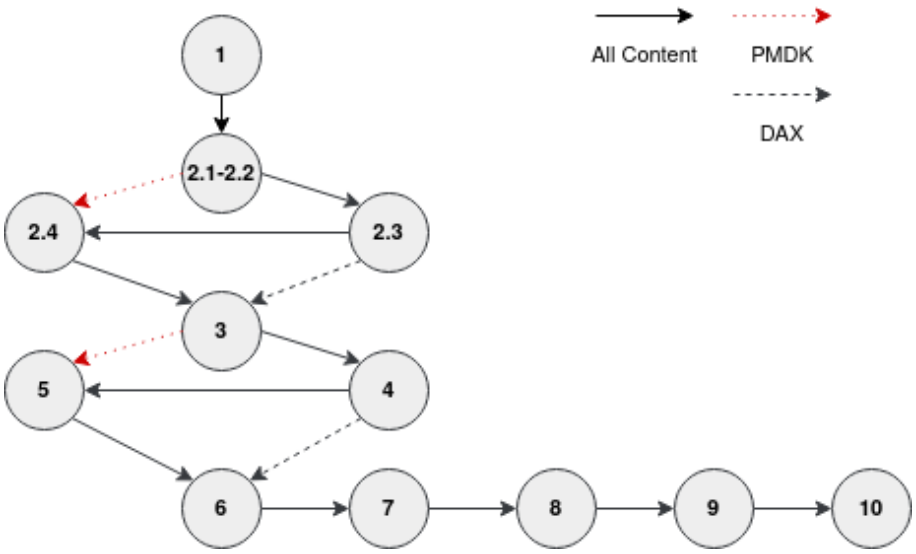

Figure 3: A visual representation of the section layout allowing for reading of experiments in parallel or focusing on only one topic. If a section has no outgoing edge of a color and shape, "All Content" should be followed. For example, only reading "PMDK" content would mean reading Sections 1, 2.1-2.2, 2.4, 3, 5, 6, 7, 8, 9, 10.

2 Background

2.1 Overview

This section provides foundational details for this thesis. First, Section 2.2 provides basic information of non-volatile memory and its integration into systems. Next, Section 2.3 contributes information on the DAX extension and how it internally works. Last, Section 2.4 provides details on the PMDK software infrastructure, and the main libraries and tools it contains.

2.2 Non-Volatile Memory

Modern systems typically employ two types of devices, storage- and volatile memory-devices. Storage devices are slow and cannot be directly accessed by the CPU, but deliver large capacities of persistent storage, and are the cheapest kind of device. Since these devices are slow and not directly accessible by the CPU, data has to be buffered in memory, before it can be accessed. Memories on the other hand, are fast, directly accessible by the CPU, but have low capacity of volatile storage, and are more costly. Non-volatile memories present a unique new addition to the memory hierarchy, combing the larger capacity of persistent storage with the low latencies of memory.

The first commercially available non-volatile memory is the Intel Optane DCPMM [17]. These devices have large capacities, ranging from 128GiB to 512GiB per module, of persistent storage, and access latencies close to conventional memory, as was shown in Figure 1. We illustrate a system layout with these new devices and the two aforementioned memory- and storage- devices in Figure 4. Like conventional memory, non-volatile memories are directly accessible the CPU over the DDR4 channels, and are byte-addressable. Unlike conventional memory, these devices have a lower cost per GB of storage capacity, making them a cheaper alternative, with the added benefit of persistent storage.

The software stack presents three main software ways of using non-volatile memories, presented in Figure 5. Firstly, they can be used a storage device with a DAX-enabled file system. Secondly, they can be used as memory with the PMDK software infrastructure. The PMDK provides libraries to, for example, use the non-volatile memory as a persistent key-value store, or volatile heap memory. Lastly, they can be used as a combination of storage and memory, by mounting a DAX-enabled file system on the device, and using the PMDK libraries alongside the file system.

2.3 Direct Access (DAX)

The first way of using non-volatile memories, is to employ them as storage devices with a file system. Conventional systems have very slow storage devices and therefore use the page cache in main memory to buffer data, illustrated in Figure 2. The page cache buffers pages of data from the storage device, before they ares accessed by the CPU, or are written to the storage device. The page cache has the advantages of being transparent and allowing for modification of data in memory, not on the storage. When using non-volatile memories as storage devices, the issue arises, that they can be directly addressed as main memory, and copying of data to the page cache would be unnecessary. To avoid this, kernel developers have developed the DAX (direct access) extension for file systems [37].

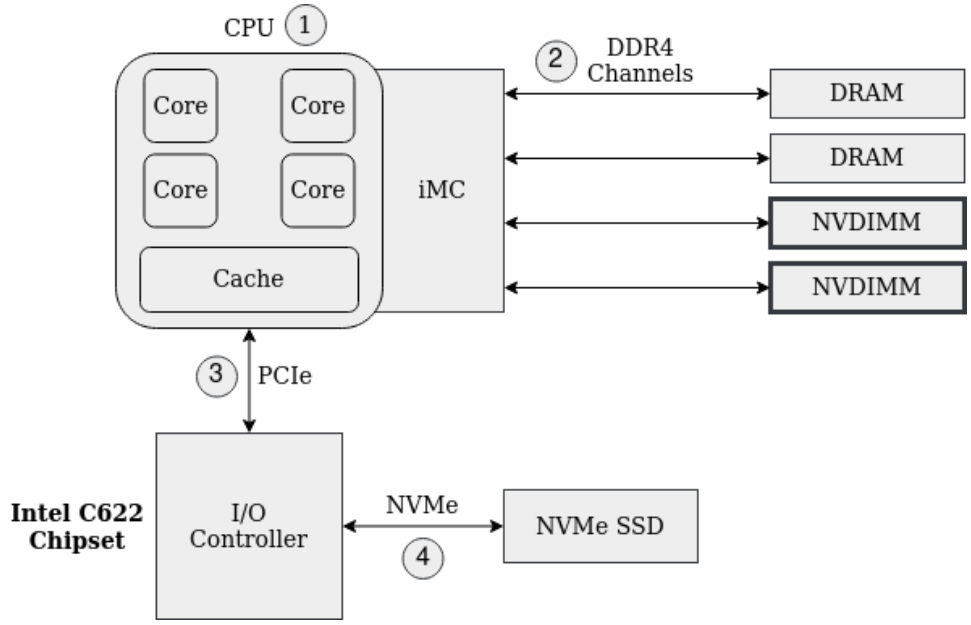

Figure 4: System hardware setup of conventional memory- and storage- devices, with new non-volatile memory. Based on the Intel Server Board S2600BPS Architecture [21], with the Intel C622 Chipset, and the Intel Optane DCPMM Guide [20]. **1.** CPU with with its very fast cache, that has a very small storage capacity. **2.** DDR4 channels that directly connect the CPU to DRAM and Optane DCPMM over the integrated Memory Controller (iMC). **3.** PCIe lane that connects the CPU to the I/O controller, from where secondary devices can be connected over SATA, NVMe, M.2, and more. **4.** Slow NVMe SSD storage that is not directly accessible by the CPU, but has larger storage capacity.

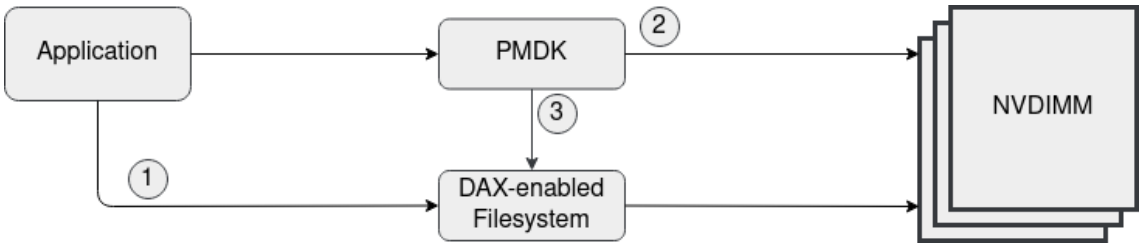

Figure 5: Possible ways of using non-volatile memories. **1.** As a storage device, with a DAX-enabled file system, and using direct load/store. **2.** As a memory device, with the PMDK. For example, using persistent memory as a volatile heap. **3.** Combining storage device with use as a memory, with DAX-enabled file system and the PMDK.

Mode	Description	File System	DAX	Device Type
sector	Block device without DAX but with file system support	Yes	No	Block
raw	Block device without DAX or file system support	No	No	Block
devdax	Character device with fault granularity of 4KiB, 2MiB, or 1GiB	No	Yes	Character
fsdax	Block device for DAX capable file system	Yes	Yes	Block

Table 1: Different access modes to be used for configuring NVDIMMs [18].

DAX eliminates the copying of data into page cache, and maps the file directly into user space. Currently, the only conventional file systems in the linux kernel supporting DAX are ext4, xfs, and ext2.

Prior to mounting a file system on a non-volatile memory, its address range, called the *namespace*, has to be configured to an access mode, to load required drivers, using the *ndctl* tool [3]. Table 1 shows all the access modes that the devices can be configured to. There are two DAX-capable access modes, *fsdax* and *devdax*. Fsdax mode is required for using the non-volatile memory as storage device and mounting a DAX-enabled file system on it. With this mode, the memory is presented as a block device, just like conventional storage, meaning it can be used with any file system, but to bypass page cache, a DAX-enabled file system has to be mounted on it.

Devdax mode on the other hand, presents the device as a character device, without the possibility of mounting a file system on it. The entire device is represented to the operating system as a single file, allowing for the entire address range of the device to be directly mapped into user space at once. This can be beneficial for systems requiring very large memory mappings, or possibly for remote direct memory access (RDMA). During this thesis, we utilize fsdax mode only, for all our evaluations.

2.4 Persistent Memory Development Kit (PMDK)

The second way of using non-volatile memories, is as memory. Since, unlike conventional memory, non-volatile memory maintains data through power loss, a new programming model was required for it. For this, Intel and SNIA have developed the Persistent Memory Development Kit (PMDK) [23]. The PMDK is a software infrastructure consisting of a number of libraries and tools. There are two different kinds of libraries. Firstly, the ones for using the memory as persistent memory, such as a persistent key-value store or for persistent data structures. Secondly, the ones for using it as volatile heap memory, just like conventional memory, but with the added benefit of larger memory capacity.

The persistent libraries are all built on top of one main library, *libpmem*. It provides the low level implementations for all basic functionality required for persistent memory, such as mapping files into memory, writing data to the memory, or persisting data to the non-volatile memory. The libraries built on top of this provide additional features, such as transactions, compile- and run-time- safety checks, and data types.

The volatile libraries are based on two main libraries, *libvmmalloc* and *memkind*. These utilize non-volatile memory as volatile heap memory. Libvmmalloc fully replaces heap memory, meaning that all memory allocations are redirected to the non-volatile memory. Memkind on the other hand only extends heap memory, giving the possibility to run non-volatile memory alongside conventional memory, and deciding to allocate memory from the non-volatile memory or the conventional memory. Figure 6 presents some of the main libraries, running on non-volatile memory with two of the possible access mode configurations.

With the introduction of the new programming model for the PMDK, came several new key concepts for programming on non-volatile memory [7]. Firstly, since data is stored through power loss, pointers need to be relocated when restarting an application. For this, the PMDK uses root objects that store sizes of allocations, and uses these to recalculate offsets of pointers. Secondly, persistent pointers are organized differently than conventional pointers. Persistent pointers require double the size, as they need to store an identifier to which memory mapped file, called pool, they belong to, and what their offset inside that pool is. Lastly, data needs to continuously be flushed from the CPU caches to the device, to maintain consistency.

Flushing of caches is done using the *clflush* instruction, which invalidates the cache line and writes the data back to the device. To improve performance, Intel has recently introduced two new instructions, *clflushopt* and *clwb* [16]. *clflushopt* is an optimized flushing, and *clwb* is a flush without invalidating the cache line. These instructions are only available on certain CPUs, therefore the PMDK checks what instruction is supported on the hardware, and initialize the libraries at load time.

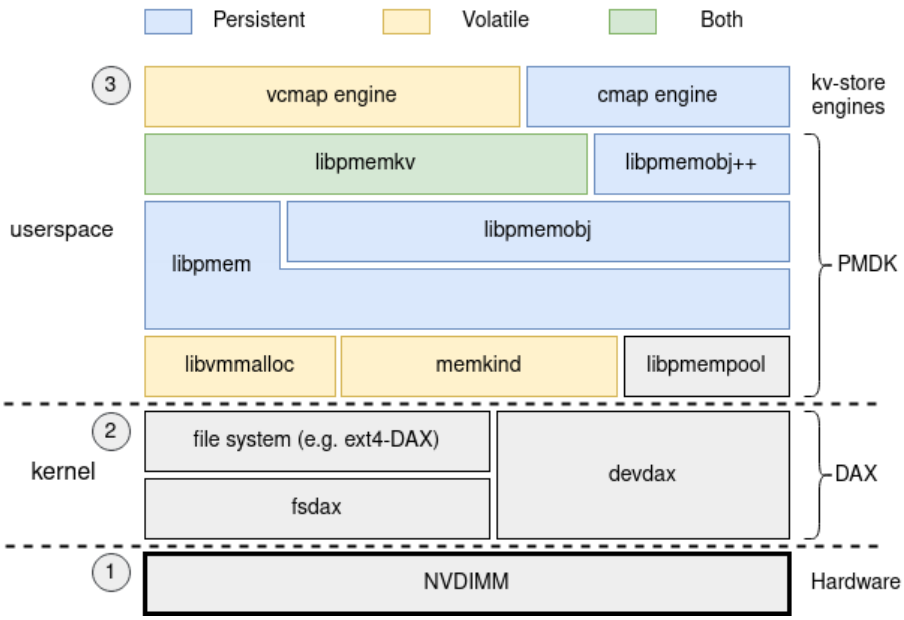

Figure 6: Layout of non-volatile memory and parts of the persistent memory software stack. **1.** The non-volatile memory hardware, which is evaluated first in Section 3.3. **2.** The DAX-enabled file system and different access modes, evaluated in Section 4.1. **3.** Some of the PMDK libraries, evaluated in Section 5.1.

3 Experimental Setup and Baseline

3.1 Overview

Having established all required prior background knowledge, we evaluate the performance of the previously shown possible ways of utilizing non-volatile memories in systems. First, Section 3.2 covers the system setup, followed by Section 3.3 establishing our baseline memory bandwidth and latency. For reproducibility, all commands for setup and execution of benchmarks are shown in Appendix B. All benchmarks we designed and implemented are publicly available [36].

3.2 System Setup: NVDIMM Emulation on DRAM

To evaluate the performance of the persistent memory software stack, we emulate persistent memory in DRAM [26]. As this is emulated memory, the results of our findings do not present absolute values, but are meant to understand the software stack, and identify patterns and behavior. The system uses a 6 core Intel i7-8850h CPU with 2x16GB of DDR4 memory, where 2x7GB are used for emulation of persistent memory. The system is running Linux kernel 5.4.0.

For the experiments, we use tmpfs, a temporary file system living in the page cache, ext4 and xfs, which are both running on an emulated block device on DRAM [14], their DAX-enabled versions, ext4-DAX and xfs-DAX, and lastly NOVA, a file systems designed and optimized for persistent memories. NOVA delivers stronger consistency guarantees than ext4-DAX and xfs-DAX [42]. Experiments with the NOVA file system run Linux kernel 5.1.0, since it has not been added to the mainline kernel.

3.3 Baseline

To establish our baseline performance, we measure memory bandwidth and average memory access latency. We measure bandwidth by calling an anonymous *mmap*, meaning the memory mapping is not backed by an actual file, with a 1GiB size. We divide the mapping into "chunks" of a certain size, ranging from 4B to 4MiB, and continuously access different chunks in sequential and random order, for 60 seconds. Accesses consist of reading from the memory and writing to the memory, by calling *memcpy*. Reading is achieved by copying a chunk of data into a destination buffer in memory. Similarly, writing is done by copying a chunk of data from a source buffer in memory to the mapping. To eliminate overheads of page faults, page table entries are pre-populated by passing the MAP_POPULATE flag in the *mmap* call.

Figure 7 shows the bandwidth for the different access types, with the varying chunk sizes. The resulting peak memory bandwidth is measured at **18.9 GiB/s**. To identify the differences in sequential and random bandwidth, we measured the number of hardware perfetches per CPU cycle, and identified that sequential access has increased prefetching for chunk sizes up to 16KiB. These data points are presented in Appendix 16.

To verify the accuracy of the result, we cross check it with two commonly used tools for performance analysis, LMbench [2] and STREAM [28]. LMbench is a widely used benchmark for analyzing systems performance, whereas STREAM is a benchmark solely

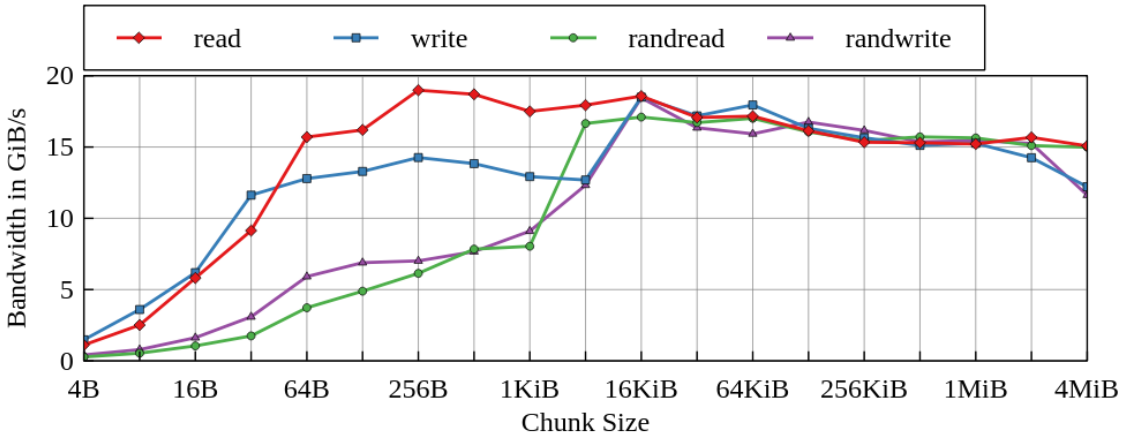

Figure 7: Baseline DRAM bandwidth measured using a memory mapped file, writing/reading to/from the mapping with varying chunk sizes, in random and sequential order. Shows a peak bandwidth of **18.9 GiB/s** and converging peak bandwidth for all access types at 18 GiB/s. We verified that higher sequential access bandwidth is due to increased hardware prefetching.

meant for measuring sustainable memory bandwidth. Both benchmarks show an average bandwidth within a similar range, confirming the results we measure.

We measure average memory access latency by allocating a 300MiB buffer in memory, and filling it with random pointers, pointing to another random pointer inside the buffer. Starting at the base of the buffer, we move from one pointer to the next, for 10,000,000 times. The resulting memory latency is 30.71 nanoseconds. To verify the results, we also measure the latency using the Intel Memory Latency Checker tool [38]. The tool is configured to measure the load latency for idle memory accesses. The resulting memory access latency is 30.5 nanoseconds. In conclusion, the average memory access latency on our system is close to **30.5 nanoseconds**.

Summary. We measured our baseline peak memory bandwidth at 18.9GiB/s and the memory access latency at 30.5 nanoseconds, and have cross checked the accuracy of our results with several different tools.

4 Benchmark: DAX

4.1 Overview

Having established the baseline, we proceed by examining the performance of using non-volatile memories as storage devices, with DAX-enabled file systems. Our main results show:

1. DAX-enabled file systems bypassing the page cache have an increase in asynchronous asynchronous I/O bandwidth of up to 32.85%, compared to file systems without DAX. These findings are presented in Section 4.2.

2. The performance of file system operations involving I/O is improved by up to 67.96%, and operations without I/O are not affected by DAX. We present these findings in Section 4.3.

3. Ext4-DAX has a performance improvement of 50.28% on page faults, showing that avoiding the page cache affects performance. These findings are shown in Section 4.4

The evaluations are followed by a summary of our findings in Section 4.5, and we present guidelines for using non-volatile memory as storage devices in Section 7.

4.2 Measuring I/O Bandwidth of DAX-enabled File Systems

To evaluate whether DAX bypassing the page cache improves the performance for I/O operations, we measure the I/O bandwidth for file systems with DAX enabled, and compare it to file systems without DAX. We select fio [5] as our benchmarking tool. For our evaluation, we configure fio to create a 1GiB file on each file system and use its libaio engine to submit asynchronous read and write requests for the file. Libaio is the Linux Asynchronous I/O library. We choose asynchronous I/O, as servers utilize this and we believe servers will be one of the main adopters of non-volatile memories. With servers, certain threads handle only incoming client requests, while other threads handle the actual workload.

Fio's libaio engine submits I/O requests, in a single system call, without waiting for the call to complete, while a separate process waits for completed calls. We use fio with a single thread submitting I/O requests for reading, writing, random reading, and random writing data, 4KiB in size, which corresponds our system's page size. Requests are continuously issued for 60 seconds, for each I/O type, and fio measures the average bandwidth of each.

The resulting bandwidth for each of the file systems and the respective I/O operation is depicted in Figure 8. DAX-enabled file systems achieve higher bandwidth across all workloads. The achieved bandwidth by ext4-DAX gets close to that of tmpfs. Similarly, xfs-DAX achieves comparable results to tmpfs. Overall, ext4-DAX achieves an average 32.85% higher bandwidth than ext4, and xfs-DAX outperforms xfs by an average of 31.60%.

Summary. We evaluated the performance of DAX-enabled file systems compared to their counterpart file system without DAX, and have shown that DAX bypassing the page cache improves performance for single threaded asynchronous I/O by up to 32.85%.

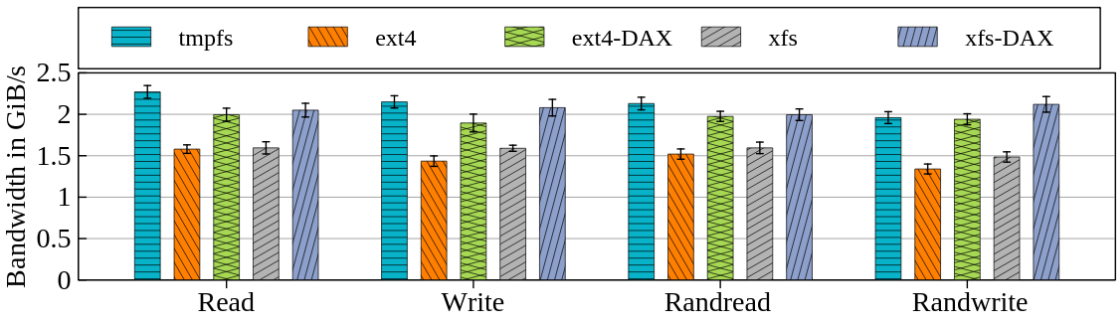

Figure 8: Fio benchmark using asynchronous I/O operations on the different file systems, to evaluate performance of DAX bypassing the page cache. Y-axis shows the achieved bandwidth in GiB/s, where higher is better. DAX-enabled file systems achieve higher bandwidth than file systems without DAX, on all workloads. Error bars represent standard deviation.

4.3 Measuring File Operation Latencies of DAX-enabled File Systems

Next, we measure the latency of different file system operations on the different file systems, to evaluate how DAX-enabled file systems perform on such workloads. We choose Filebench [34] as our tool for generating workloads. Filebench is a workload generator, used for benchmarking storage and file systems. We choose three different workloads. Inspired by related work [42], we configure Filebench to a similar workload of large amounts of small files and issue small I/O requests, which is comparable to server workloads. Firstly, Filebench creates 100,000 files, each 64KiB, and fills the entire file with random data. Next, it writes 16 times to already created files, in 4KiB I/Os. Lastly, it deletes all created files. For each of these operations, Filebench measures the number of operations completed per second, from which we calculate the average latency of a single operation. Each workload is repeated 10 times for every file system, and we report the average latency.

Figure 9 shows the resulting latency for each operation on the different file systems. Since file systems have different implementations, they can show very different performance for individual operations, which is why we do not compare the file systems against each other, but only focus on how the DAX-enabled version of each file system compare to their version without DAX. File creation on ext4-DAX only takes 10.00 microseconds, compared to 16.66 microseconds on ext4, requiring only 60.00% of the time it takes on ext4. Similarly, writing to the file takes only 4.00 microseconds on ext4-DAX, compared to 9.48 microseconds on ext4, showing an increase in performance by 57.71%. This is due to ext4-DAX bypassing the page cache when writing to the file, whereas ext4 buffers the write in page cache and then proceeds with writing the data back to the file system on the storage device.

Similar results appear when comparing xfs-DAX to xfs. On file creation, xfs-DAX requires 10.00 microseconds, compared to 13.34 microseconds on xfs, requiring 74.99% the latency of xfs. The writing workload only requires 3.85 microseconds with DAX enabled, and 12.03 without DAX, showing an increase in performance by 67.96%. Again, this is

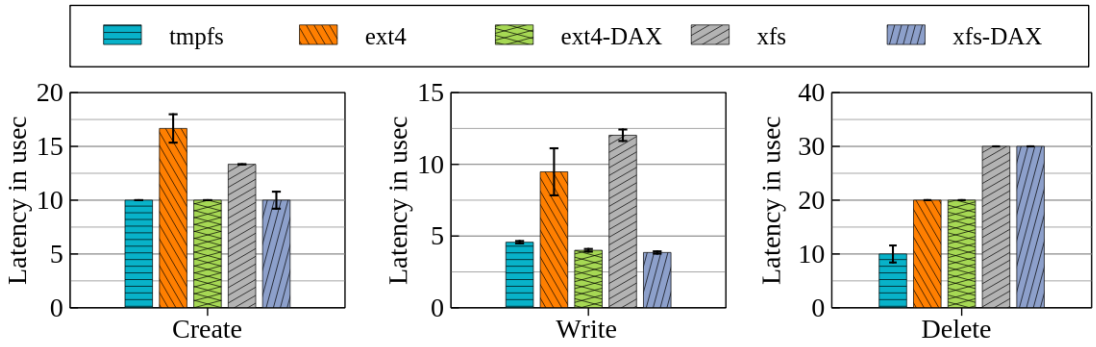

Figure 9: Filebench workload for file system operations on the different file systems. *Create* creates 100,000 64KiB files and fill the entire file with random data. *Write* writes to already created files 16 times, 4KiB at a time. *Delete* deletes all already created files. Y-axis depicts the achieved latency for each operation, where lower is better. DAX-enabled file systems achieve lower latencies for all workloads that involve writing, as it bypasses the page cache, but that does not affect other operations. Error bars represent standard deviation.

due to DAX bypassing the page cache and writing directly to the storage. Performance of tmpfs is typically similar to that of DAX-enabled file systems, due to it being in the page cache, also avoiding copying of data. Deleting of files achieves the same latency for DAX-enabled file systems, as it does for their counterparts without DAX. DAX has no involvement in the deletion of files, and therefore performance does not change for the file systems on such operation.

Summary. We measured the latencies of several different file operations and have shown that DAX-enabled file systems bypassing the page cache can improve performance for I/O operations by up to 67.96%, but operations not involving I/O are not affected by DAX.

4.4 Measuring the Cost of Page Faults for DAX-enabled File Systems

To specify the overheads introduced by copying data to page cache, we measure the cost of page faults across the different file systems. On page faults, page table entries are setup, storing the mapping of the virtual- to the physical- memory address. Additionally, if necessary, pages of data from the storage device are copied to the page cache in memory. We identify if DAX avoiding the copying of data to page cache on page faults improve performance, compared to file systems without DAX having to bring data into page cache.

We map a 4GiB file, from each file system, into memory using *mmap*, and divide the mapping into page sized chunks. We then access the first byte of these chunks, triggering one page fault on each access. The page fault then sets up the page table entry, and if necessary, brings the data into memory. We measure the time it takes to return from the one byte access. Only the first byte is accessed, to have the smallest possible overhead for the access, and the latency we measure being solely the latency of the page fault.

Typically, there are several hardware and software techniques, such as prefetching and

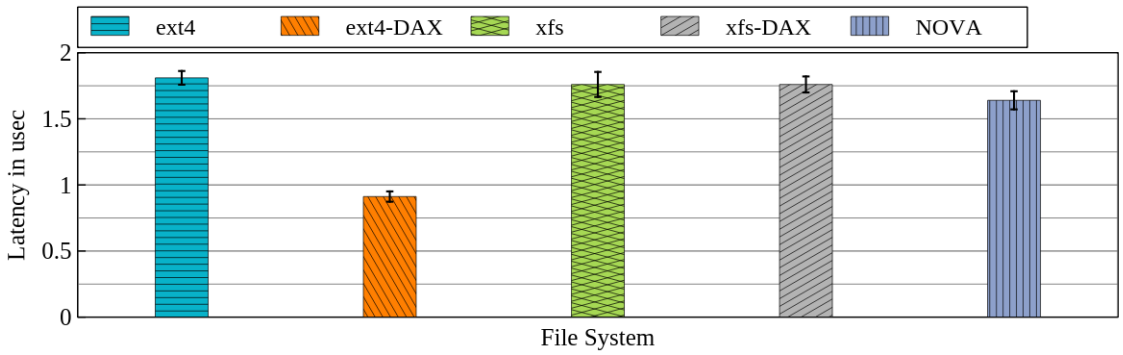

Figure 10: Measuring the latency of page faults for each file system to identify the cost of bringing a page into page cache. Ext4-DAX bypassing the page cache improves performance by 44%. Error bars represent standard deviation.

pre-allocation of page table entries, to avoid the penalty of page faults. These techniques are based on access patterns, and aim to predict what will be accessed next, before it is actually being accessed. To account for these techniques, we access the first byte of pages within the mapping in random order. The file is removed from memory and remapped, once we start causing less page faults. We repeat this for a total of 1,000,000 page faults.

Figure 10 shows the resulting average latency for a page fault on the respective file system. Ext4-DAX, requires 0.91 microseconds, compared to ext4 taking 1.81 microseconds, showing a performance increase of 50.28%. This shows how DAX improves performance, by avoiding the page cache, and providing direct access. From this, we can also calculate that bringing a page into the page cache adds an additional overhead of 0.90 microseconds with ext4. Surprisingly, xfs-DAX performs the same as xfs. To analyze why, we trace the function calls using ftrace [1], the Linux kernel internal tracer. We find that xfs-DAX calls the fault handler of DAX twice, where the second call sets up the page table entry. Ext4-DAX only calls the DAX fault handler once. The traces are publicly available at [35].

NOVA achieves similar latencies to xfs-DAX. This is due to NOVA using *atomic-mmap* to map files into user space [42]. With *atomic-mmap*, data is copied into "replica pages", which are then mapped into user space. This provides stronger consistency guarantees, as data will be modified in the replica pages, and pointers to the pages are swapped atomically, when calling *msync*. It introduces an overhead similar to the page cache, but provides stronger consistency guarantees than ext4-DAX and xfs-DAX provide [42].

Prior to measuring page faults, we measure the cost of mapping files into memory for the different file systems, using *mmap*. We do this to identify how many page table entries are set up on the *mmap* call, or whether they are set up only on page faults. We find that the cost of the function call is the 1.11 microseconds across all the file systems. When tracing the function call, we find that it does not set up any page table entries, but only does required work to acquire the memory region and mark it as accessible.

Summary. We measured the cost of page faults, in order to identify if DAX avoiding the copying of data to page cache improves performance. Our findings show that on ext4-DAX performance improved by 50.28%, compared to ext4.

4.5 Results

These benchmarks show that, when using non-volatile memories as storage devices, DAX-enabled file systems provide better performance for I/O operations. DAX bypassing the page cache improves I/O bandwidth by up to 32.85% on ext4, and operations not involving I/O are not affected by DAX. Additionally, we identified that bringing data into page cache adds an overhead of 0.90 microseconds on ext4, and ext4-DAX avoiding this improves performance by 50.28%, compared to ext4.

5 Benchmark: PMDK

5.1 Overview

After having established the performance of using non-volatile memories as storage devices, we measure the performance of using them as memories with the PMDK software infrastructure. Our main findings show:

1. The average cost of persistence is 3x higher than volatile use. These findings are presented in Section 5.2.

2. Costs of achieving persistence, by flushing data from the caches can be amortized, if data is kept close together, and is flushed together. We show these findings in Section 5.3.

The evaluations are followed by a summary of our findings in Section 5.4, and we present guidelines for avoiding software overheads in Section 7.

5.2 Measuring PMDK Key-Value Store Performance

As we explained in Section 2.2, non-volatile memories can be used as storage and as memories. We now evaluate performance for using non-volatile memories as memories, by measuring the performance of PMDK libraries and comparing differences between volatile and persistent usage to our baseline, to identify the cost of persistence and the library's additional software overheads. We use the provided key-value store within the PMDK, and its persistent and volatile engines. The key-value store is optimized for persistent memory, featuring a large variety of different engines, providing different features, such as persistence, concurrency, and key ordering, and supporting multiple language bindings, such C++, Java, JavaScript, Ruby, and more.

Benchmarking the key-value store with the individual engines is done using pmemkv-tools [19], a benchmarking utility for the key-value store, ported from LevelDB's db_bench performance benchmark for databases [11, 13]. The benchmark fills the key-value store with 45,000 keys, 16B each, and a value of a specified size, ranging from 64B up to 256KiB. Additionally, we perform 1,000,000 reads of keys in random order to retrieve read performance. For simplicity, we have renamed the engines to c-eng, p-eng, pc-eng, and s-eng which respond to the official engine names of vcmap, tree3, cmap, and vsmap, respectively. Naming letters represent; 'p' for persistent, 'v' for volatile, 'c' for concurrent, and 's' for sorted.

Bandwidth Comparison. Figures 11 and 12 show the bandwidths for reading and writing, respectively, with varying value sizes. On reading, the volatile vc-eng achieves the highest bandwidth for sizes up to 4KiB, since it is a volatile engine, but drops for larger sizes. To investigate why, we profile the execution using perf [4], a tool for performance profiling, and find that the implementation of the engine does unaligned memory accesses, causing a significant performance penalty. Unaligned memory accesses are a common issue we see appear across all engines tested. The volatile vs-eng performs the worst, because of its implementation using a red-black tree, requiring a minimum of three lookups for locating a key, and a maximum of $\mathcal{O}(\log n)$. The read performance of the engines peaks

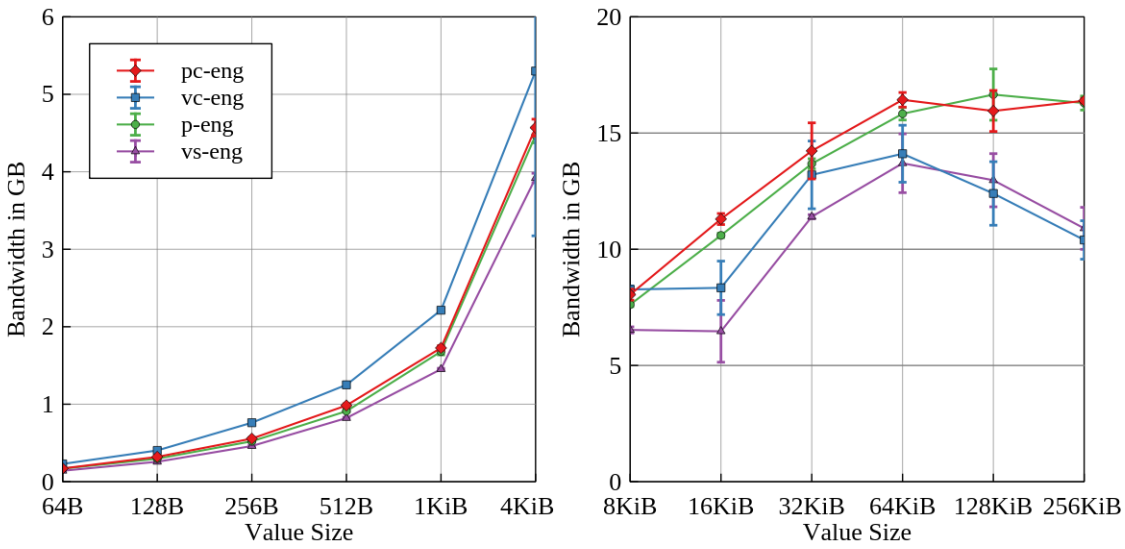

Figure 11: Reading bandwidth for pmemkv engines, with 45,000 keys and varying data sizes. Volatile engines perform better on smaller sizes, but worse for larger sizes due to unaligned memory accesses. Error bars represent standard deviation.

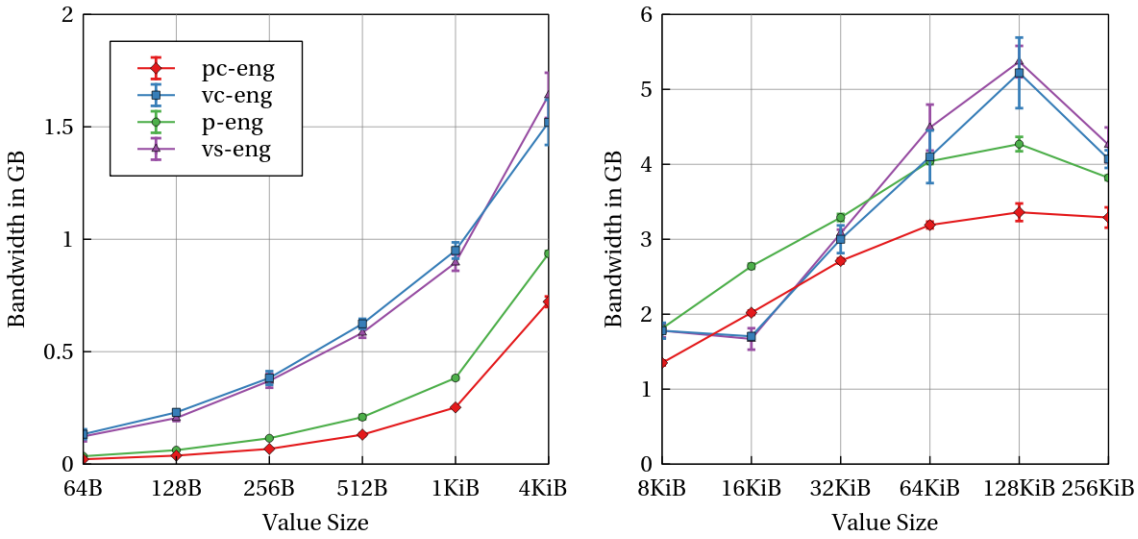

Figure 12: Writing bandwidth for pmemkv engines, with 45,000 keys and varying data sizes. Volatile engines perform better, but drop suddenly for sizes 16KiB value size. We verified that this is due to a decrease in hardware prefetching, shown in Appendix 17. Error bars represent standard deviation.

at 17GiB/s, close to our baseline DRAM bandwidth, followed by a sudden drop for data size of 256KiB, assumed to be caused by L2 cache being the same size.

On writing, the volatile engines achieve a higher bandwidth than the persistent ones, but suddenly drop for value size of 16KiB. We verified that this is due to a sudden decrease in hardware prefetching, depicted in Appendix 17. The peak bandwidth for writing is at 5GiB, significantly lower than our baseline DRAM bandwidth and our previously mentioned peak reading bandwidth. This significantly lower bandwidth illustrates the software overheads of writing with the PMDK libraries and the key-value store, which we now further analyze, by breaking down the different latencies for reading and writing.

Latency Comparison. Figures 13 and 14 show the latency for reading and writing, respectively. These plots correspond to the previous results of bandwidth, and therefore show similar patterns. On reading, the volatile vc-eng achieves the lowest latency, for sizes up to 4KiB. As previously explained for the reading bandwidth, the higher latency for larger sizes is due to the unaligned memory accesses. The persistent engines follow the vc-eng with slightly higher latency, followed by the vs-eng with the highest latency. Again, this is due to the internal implementation of the vs-eng.

Comparing the latency of the volatile vc-eng to our baseline DRAM access latency, we calculate the software overhead of reading with the PMDK libraries. Reading 64B, the size of a cache line, takes 0.38 microseconds with the vc-eng. This is 12.46x times higher than our baseline latency, showing the additional software overhead of volatile use. Reading 64B with the persistent pc-eng takes 0.46 microseconds, 15.08x our baseline latency, showing an even higher software overhead. Comparing our volatile to persistent use, we calculate the additional software overhead for reading with persistent engines as being 1.21x higher than volatile engines.

On writing, the persistent pc-eng has the highest latency, due to it being a persistent engine, and supporting concurrent accesses, followed by the persistent p-eng. Pc-eng utilizes transactions to provide concurrent accesses, which introduce a significant overhead for smaller data sizes. The penalty of transactions becomes less significant as the data size increases, since it is a constant overhead that does not depend on data size. Volatile engines achieve lower latencies, as they do not need to ensure data has reached the device, to keep it in a consistent state.

Again, comparing the latency of the volatile vc-eng, we calculate the software overhead of writing with the PMDK libraries. For a 64B write, the vc-eng takes 0.57 microseconds, 18.69x our baseline DRAM access latency. The persistent pc-eng takes 3.52 microseconds for a 64B write. Comparing this to baseline DRAM access latency, we identify that its software overhead is 115.41x higher. Persistent engines, compared to volatile ones, have a 6.18x higher overhead for 64B writes. Over all value sizes, the average cost of persistence decreases to 3x higher than volatile use.

Summary. We measured the performance of the PMDK key-value store, and identify its software overheads, by comparing persistent and volatile engines to our baseline performance, from Section 3.3. We identified, that reading 64B with persistent engines has an overhead 1.21x higher than volatile engines, and 15.08x higher than our baseline. For writing of 64B, the overhead of persistent engines is 6.18x higher than volatile engines, and 115.41x higher than the baseline. Over all value sizes, the average cost of persistence decreases to 3x higher than volatile use. These findings show that it is important for developers to carefully organize persistent and volatile data structures.

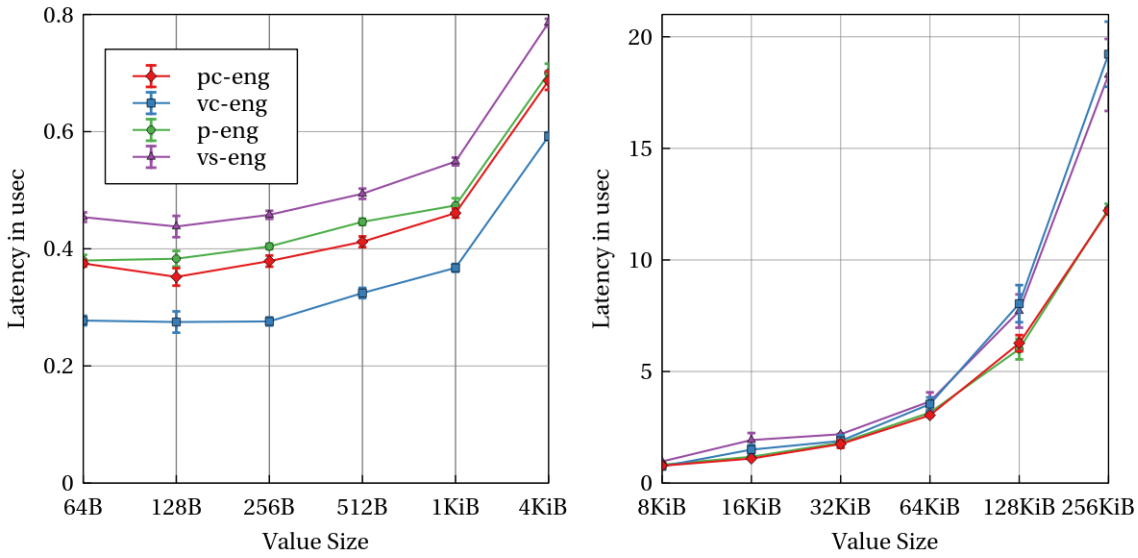

Figure 13: Reading latency for pmemkv engines, corresponding to Figure 11, with 45,000 keys and varying data sizes. Higher latencies for volatile engines, for sizes above 64KiB, are due to unaligned memory accesses. Error bars represent standard deviation.

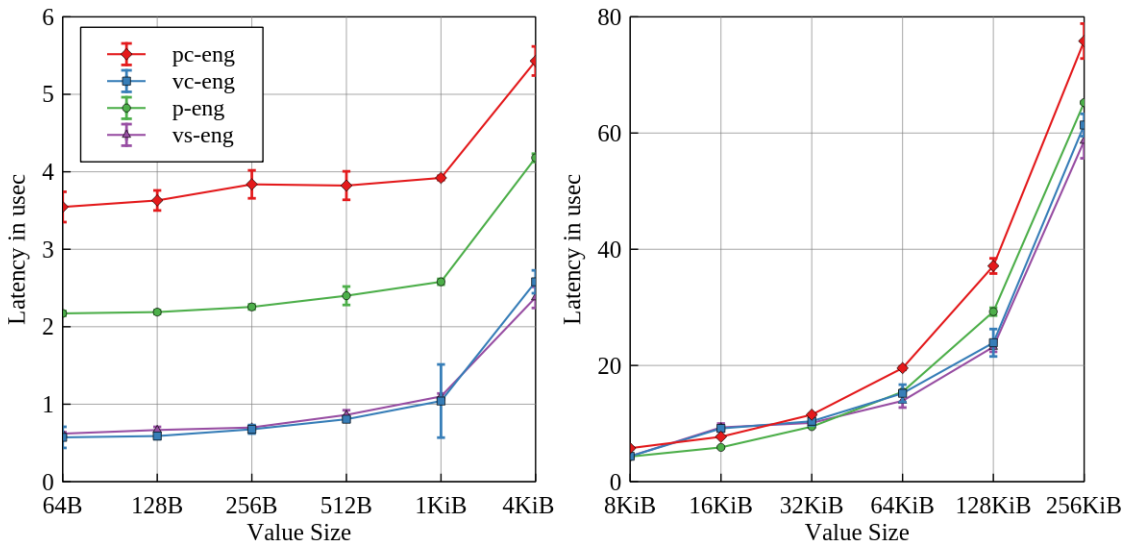

Figure 14: Writing latency for pmemkv engines, corresponding to Figure 12, with 45,000 keys and varying data sizes. Shows the added cost of persisting data on the non-volatile engines is higher than volatile use. Error bars represent standard deviation.

5.3 Cost of Achieving Persistence

As we observed in Section 5.2, achieving persistence introduces additional overheads. Part of the overhead comes from flushing caches, which ensures that data is written back to the non-volatile memory, and will not be lost in case of power failure. We use the pmembench benchmark, provided with the PMDK, to evaluate the additional overhead of issuing a function call to flush data from the caches.

The benchmark maps a file into memory and persists data by calling the library function *pmem_persist*. The function call causes the data to be flushed from the caches, and be written back to the device. The benchmark will be run 10 times, each time calling *pmem_persist* 1,000 times. The latency for each of the calls to *pmem_persist* is measured, and averaged across all the calls. This means we measure average latency, not individual latencies. As mentioned before, depending on the CPU architecture, different instructions are available for flushing. Our system uses the *clflush* instruction.

Figure 15 shows the resulting latency per cache line flushed using the *pmem_persist* function with varying value sizes. The latency is calculated by dividing the latency for flushing the entire value by the number of cache lines in it. Initially, as the data size increases, the latency decreases significantly, but it starts leveling off for data sizes above 4KiB. This shows that flushing together is most efficient for smaller data sizes, between 1KiB and 4KiB. For larger data sizes, the performance gains diminish and become less significant.

Summary. As we previously identified, persistence comes with extra overheads. We break down the cost of this overhead by measuring the latency of flushing data from the caches. Our findings show that keeping data close together minimizes flushing overheads. Additionally, we show that keeping data together, in sizes between 1KiB and 4KiB, significantly decreases cache flushing overheads, and performance gains diminish for larger sizes.

5.4 Results

Our findings show that using the PMDK software infrastructure introduces overheads for achieving persistence, but certain practices can minimize these overheads. Firstly, we saw that the average cost of persistence is 3x higher than volatile use, making it crucial for performance to carefully organize persistent and volatile data structures. Secondly, we saw that the cost of flushing caches, to achieve persistence, can be amortized by maintaining persistent data structures close together on the non-volatile memory, and flushing them together to amortize flushing costs. Flushing together is most efficient for data sizes between 1KiB and 4KiB, as for larger sizes performance gains become less significant.

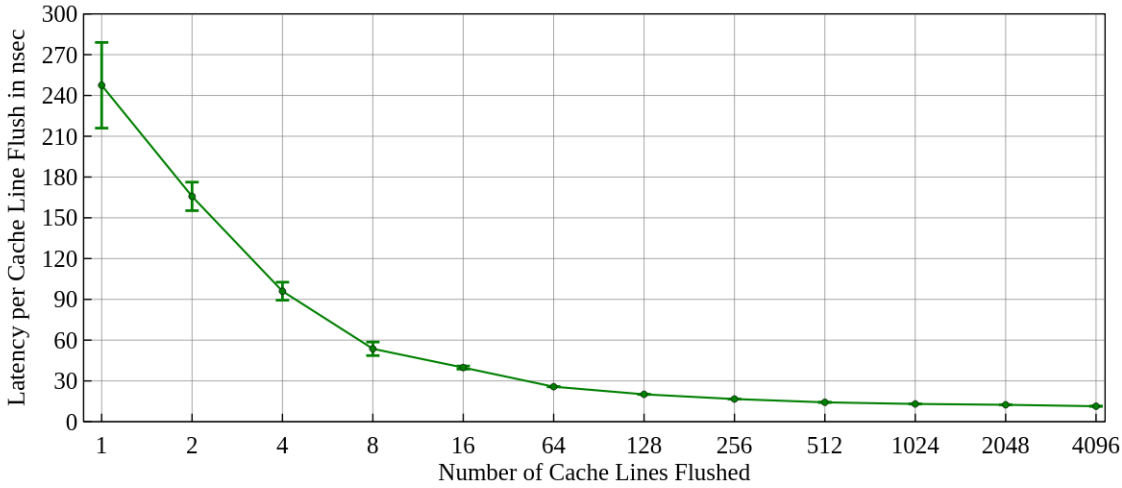

Figure 15: Latency per cache line, for cost of flushing data from the caches, with different sized data. Performance overhead of flushing can be amortized when data is kept close together, and is flushed together. Error bars represent standard deviation.

6 Results and Analysis

Non-volatile memory as storage devices. Our evaluation of using non-volatile memories as storage devices shows that DAX-enabled file systems provide better performance than conventional file systems for I/O operations. We evaluated the performance of this new generation file systems with three different benchmarks. Firstly, we measured I/O bandwidth of these file systems, and saw that DAX bypassing the page cache improves performance by up to 32.85%, on single threaded workloads.

Secondly, we evaluated performance of different file system operations on DAX-enabled file systems, showing that DAX can significantly improve performance for operations involving I/O, but does not affect performance of other operations. Lastly, we measured the cost of page faults across the different file systems. Results showed that DAX avoiding copying of data to page cache improves performance by 50.28% with ext4-DAX.

Non-volatile memory as memory devices. The evaluation of using non-volatile memories as memory devices, in volatile and persistent mode, showed that the PMDK software infrastructure introduces certain overheads. We first identified software overheads by measuring performance of the key-value store from the PMDK, and comparing its volatile- to persistent- use, to identify the cost of persistence. Results showed that the average cost of persistence is 3x higher.

Following these findings, we broke down the cost of persistence, to identify what persistence depends upon. Persistence depends on the flushing of data from caches. Flushing of caches will write data back to the medium to ensure the storage stays in a consistent state in case of power failure. We measured the cost of issuing flush instructions, and identified that overheads of flushing can be amortized, when data structures are kept close together, and are flushed together. Flushing together is most efficient for data sizes between 1KiB and 4KiB, as for larger sizes performance gains become less significant.

7 Discussion

Lessons learned during this research. From the mistakes made and pitfalls encountered during this thesis, we provide some helpful advice for researchers employing microbenchmarking to measure system performance. Firstly, **assume produced numbers are wrong.** This gives incentive to think of how else things can be measured or through what additional testing the produced numbers can be proven to be correct. Secondly, **measure more.** At the micro- and nanosecond levels, it is important to measure millions, if not billions of times, to achieve reliable results. Systems are not stable enough to produce consistent results with only a few iterations.

Whenever possible, include errors such as standard deviations or standard errors with the results. Lastly, **ask questions about results.** If you produce some results, ask yourself why the results are what they are, and not half of it or double of it. Being able to explain why a result is what it is brings more validity to the results, as well as more credibility to the research.

Guidelines for employing non-volatile memories. Based on our findings and experiences during this evaluation, we propose several guidelines for incorporating non-volatile memories into systems.

1. When using non-volatile memories as storage devices, DAX-enabled file systems provide better performance than conventional file systems for I/O operations, due to the bypassing of page cache.

2. Using these devices as memories introduces software overheads developers should be aware of. Since the cost of persistence is 3x higher, it is important to carefully plan and organize persistent and volatile data structures.

3. The overhead of persistence can be amortized when persistent data is kept close together, and is flushed together. Flushing together is most efficient for data sizes between 1KiB and 4KiB, as for larger sizes performance gains become less significant.

4. We saw that the PMDK software infrastructure uses unaligned memory accesses, adding significant overheads.

Future implications. This new technology brings major implications for future systems. Servers and other large storage systems can be running 10s of TBs of non-volatile memories. With estimates being that by 2025 the Global Datasphere will grow to 175 zettabytes of data [32], non-volatile memories present a strong alternative to conventional storage. Running large capacities of non-volatile memories will diminish the need for conventional storage- and memory- devices. Conventional storage will only be necessary for data that is infrequently accessed. It is cheaper to keep such data on slow storage, and load it when needed. Conventional memories will only be necessary when low latency accesses are required, and while non-volatile memories have higher accesses latency. Once non-volatile memories incorporate these factors, they will revolutionize modern systems, making conventional storage- and memory- devices obsolete.

8 Related Work

Even before the hardware was available, there have been several efforts for researching how to integrate non-volatile memories into systems, and evaluating what performance they deliver. We break down the related research into separate fields.

File Systems. Utilizing persistent memories as storage devices with file systems has been a common implementation, as these devices provide large capacities of persistent storage. Xu et al. [42] present NOVA, a file system implementation, designed and optimized for the use of volatile and non-volatile memories. Similar to our evaluation, they evaluate performance of DAX-enabled file systems and compare those to the performance of NOVA. Zheng et al. [45] present a high performance file system, called Ziggurat, that combines non-volatile memories with DRAM and disks. To improve performance, it directs writes to certain devices, based on access patterns. Similarly, their evaluation also compares the performance of DAX-enabled file systems and the NOVA file system to their file system.

Other file system implementations have been proposed for non-volatile memories [12, 9]. Worth mentioning are the ones that evaluate performance of DAX-enabled file systems, similar to our evaluation, and compare performance to their developed file system. Ou et al. [29] present such an effort, with a high performance file system, that combines DRAM with non-volatile memories. It provides write buffering of data in DRAM and lazily writes it back to the non-volatile memory, to improve performance. Gangadharaiah et al. [43] provide another such evaluation of DAX-enabled file systems, and propose a new file system that provides higher performance, while being able to efficiently handle corruption errors. Our evaluations show similar results to others, showing the increased performance of DAX-enabled file system for I/O operations.

NVDIMM use cases and implications. Besides using non-volatile memories as storage devices with file systems, there have been different suggestions for integration of these new devices into systems. Similar to our research, Wang et al. [39] present an evaluation, quantifying overheads for the PMDK. Marathe et al. [27] present the effort of porting memcached to persistent memories. Memcached is a key-value store, that is kept in memory, and serves as a cache for larger databases. Coburn et al. [8] present one of the first implementation and evaluation of using non-volatile memory as a heap. This effort provides tools, such as hash tables and search trees, for building persistent data structures.

Important to note is that this effort came before the development of the PMDK libraries, and the PMDK provides a broader range of tools. With the introduction of new technology, arise limitations of how modern systems can incorporate these devices. Bailey et al. [6] present an evaluation of such limitations and obstacles that current operating system design poses for these new devices. They discuss issues such as reliability and how virtual memory can be changed. Changes include the elimination of paging, as non-volatile memories do not require pages, or using larger page granularity.

Optane DIMM hardware. With Intel Optane being one of the first commercially available non-volatile memory, there have been several efforts into evaluating performance of real hardware [44, 40, 31], where Izraelevitz et al. [24] present one of the first, in-depth evaluations of Intel Optane DIMMs. Results verify that real hardware has lower bandwidth and higher latency than DRAM. This evaluation also provides several benchmarks

for measuring performance of applications on non-volatile memory, as well as evaluating different file system performances. Wu et al. [41] present an evaluation of the performance characteristics for 3D XPoint, the same technology used in the Intel Optane non-volatile memory.

9 Current Limitations and Future Work

Limitations. Our research utilizes emulated persistent memory, which introduces several limitations. Firstly, missing characteristics of real hardware. Studies show that real hardware behaves differently for particular access patterns and has different characteristics than DRAM [40, 24]. Emulated environments fail to account for such characteristics.

Secondly, our environment uses smaller capacity than available from real hardware. Real hardware, such as the Intel Optane DIMM, has a minimum capacity of 128GB, whereas our experiments were limited to 14GB. This restricted our benchmarks to smaller workloads, that might not represent real world applications. Thirdly, missing optimized CPU instructions. Optimized instructions for flushing of caches are only available on certain architectures. Integrations of non-volatile memories into systems would likely include CPUs supporting the optimized instructions, *clflushopt* and *clwb*.

Future work. With these limitations and the findings in our research, comes a broad spectrum of possible future research. This includes evaluating the experiments with real hardware and CPUs that support the optimized instructions, as well as using larger systems and larger workloads. As we saw in Section 5.2, a common issue within the PMDK, is the unaligned memory accesses. We suggest future work to investigate how applications can avoid these issue, by manually aligning memory or by using byte arrays, and evaluate how performance develops with the mitigation of such overheads.

Additional future work we propose consists of expanding our proposed guidelines for developers employing persistent memories. Such guidelines can include an evaluation of the different fault granularities provided by *devdax* mode, what performance they deliver to certain applications, and when developers should integrate such features into their systems. Lastly, we propose investigating the scalability of large systems with 10s of TBs of non-volatile memory, and how, or if current software- and hardware- infrastructure could support such systems.

10 Conclusion

The advent of new non-volatile memory technology will revolutionize today's and tomorrows' storage systems, offering large capacities of low latency persistent memory. In the future, databases and servers can be running 10s of TBs of NVDIMMs, bringing benefits such as instantly warm caches after reboots, and direct access to data without the page cache. These new memory devices have shifted storage technology in a new direction, triggering changes in the software stack on how we store and access data. They present two main ways of using them, as a storage device, or as a memory. We provide an evaluation into the performance characteristics of this new software stack and evaluate the performance of the different ways of using these devices.

RQ1. We ask the research question of how the software stack has changed with the DAX extension, how its bypassing of page cache affects performance, and what performance these new generation file systems deliver. We evaluate the performance of using non-volatile memories as storage devices with DAX-enabled file systems, by benchmarking the file systems. First, we benchmarked the performance of asynchronous I/O bandwidth for DAX-enabled file systems, and compared it to conventional file systems. From this we identified that DAX bypassing page cache improves performance by up to 32.85%. Next, we measured performance of different file system operations to identify how DAX affects such operations. Our findings show that DAX-enabled file systems bypassing the page cache can improve performance for I/O operations by up to 67.96%, but operations not involving I/O are not affected by DAX. Last, we measured the cost of page faults across the file systems, and identified that ext4-DAX outperforms ext4 by 50.28%, due to avoiding the page cache.

RQ2. We ask the research question of what design decisions were made during the development of the PMDK software environment, how these decisions affect its performance, and what overheads are coming from the libraries themselves. We evaluated the performance of using non-volatile memories as memory, with the PMDK software infrastructure, by measuring the performance of its key-value store, and comparing its volatile- to persistent- use. Our findings show that the average cost of persistence is 3x higher than volatile use, making it important to carefully organize persistent and volatile data structures. Additionally, we identified what the cost of persistence depends on, by measuring the cost of issuing instructions to persist data. We show that the cost of persistence can be amortized when persistent data structures are kept close together, and are flushed together. Flushing together is most efficient for data sizes between 1KiB and 4KiB, as for larger sizes performance gains become less significant. These results present two practices to minimize software overheads and maximize performance.

References

[1] Ftrace, Function Tracer. https://elinux.org/Ftrace.

[2] LMbench - Tools for performance analysis (Version 3.0-a1). http://www.bitmover.com/lmbench/.

[3] NDCTL - Utility for managing the Linux LIBNVDIMM Kernel subsystem. https://github.com/pmem/ndctl.

[4] perf: Linux profiling with performance counters (Version 5.4.41). https://perf.wiki.kernel.org/index.php/Main_Page.

[5] J. Axboe. Flexible I/O Tester (Version 3.19). https://github.com/axboe/fio.

[6] K. Bailey, L. Ceze, S. D. Gribble, and H. M. Levy. Operating System Implications of Fast, Cheap, Non-Volatile Memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*. HotOS'13:, 2011.

[7] P. Balcer. Persistent Memory Development Kit - The libpmemobj library. https://pmem.io/pmdk/libpmemobj/, 2015.

[8] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. *SIGARCH Comput. Archit. News*, 39(1):105–118, Mar. 2011.

[9] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09),*, 2009.

[10] J. Corbet. Supporting filesystems in persistent memory, 2014.

[11] R. Dickinson. Benchmarking with different storage engines using pmemkv. https://pmem.io/2017/12/27/pmemkv-benchmarking-engines.html, 2017.

[12] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, New York, NY, USA, 2014. Association for Computing Machinery.

[13] Google. LevelDB. https://github.com/google/leveldb.

[14] P. Gortmaker. The Linux Kernel Documentation - Using the RAM disk block device with Linux. https://www.kernel.org/doc/html/latest/admin-guide/blockdev/ramdisk.html, 1995.

[15] G. Heiser. Systems Benchmarking. http://www.cse.unsw.edu.au/~Gernot/benchmarking-crimes.html, 2020.

[16] C. Hui. Enabling Persistent Memory in the Storage Performance Development Kit. https://software.intel.com/content/www/us/en/develop/articles/enabling-persistent-memory-in-the-storage-performance-development-kit-spdk.html, 2019.

[17] Intel. Intel Optane Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[18] Intel. NDCTL User Guide - Managing Namespaces. https://docs.pmem.io/ndctl-user-guide/managing-namespaces.

[19] Intel. pmemkv-tools. https://github.com/pmem/pmemkv-tools.

[20] Intel. *Intel Optane DC Persistent Memory Qick Start Guide*, 2020.

[21] Intel. *Intel Server Board S2600BP Intel Compute Module HNS2600BP Product Family Technical Product Specification*, 2020.

[22] Intel and Micron. Intel and Micron Produce Breakthrough Memory Technology. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology, 2018.

[23] Intel and SNIA. PMDK: Persistent Memory Development Kit, 2016.

[24] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. https://arxiv.org/abs/1903.05714, 2019.

[25] R. Jain. *The Art of Computer Systems Performance Analysis*. John Wiley and Sons Inc., 1991.

[26] M. Maciejewski. How to emulate Persistent Memory, 2016.

[27] V. J. Marathe, M. Seltzer, S. Byan, and T. Harris. Persistent Memcached: Bringing Legacy Code to Byte-Addressable Persistent Memory. In *9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 17)*, Santa Clara, CA, July 2017. USENIX Association.

[28] J. D. McCalpin. STREAM: Sustainable Memory Bandwidth in High Performance Computers (Version 5.10). https://www.cs.virginia.edu/stream/.

[29] J. Ou, J. Shu, and Y. Lu. A High Performance File System for Non-Volatile Main Memory. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, New York, NY, USA, 2016. Association for Computing Machinery.

[30] J. Ousterhout. Always measure one level deeper. *Communications of the ACM 61*, pages 74–83, 2018.

[31] I. B. Peng, M. B. Gokhale, and E. W. Green. System Evaluation of the Intel Optane Byte-addressable NVM. *Proceedings of the International Symposium on Memory Systems*, Sep 2019.

[32] D. Reinsel, J. Gantz, and J. Rydning. The Digitization of the World From Edge to Core. *An IDC White Paper*, (Doc# US44413318), 2018.

[33] A. Rudoff. Developers Embrace Intel Optane DC Persistent Memory for Web-Scale Data-Centric Solutions. `https://software.intel.com/content/www/us/en/develop/articles/developers-embrace-intel-optane-dc-persistent-memory-for-web-scale-data-centric-solutions.html`, 2019.

[34] V. Tarasov, E. Zadok, and S. Shepler. Filebench - A Flexible Framework for File System Benchmarking. *;login*, 41, 2016.

[35] N. Tehrany. Function Traces - "Evaluating Performance Characteristics of the PMDK Persistent Memory Software Stack". `https://github.com/nicktehrany/pmem_evaluation_traces`, 2020.

[36] N. Tehrany. membench - Benchmarking Memory and File System Performance. `https://github.com/nicktehrany/membench`, 2020.

[37] L. Torvalds. Linux kernel. `https://github.com/torvalds/linux/blob/master/Documentation/filesystems/dax.txt`.

[38] K. Viswanathan. Intel Memory Latency Checker (Version 3.9). `https://software.intel.com/content/www/us/en/develop/articles/intelr-memory-latency-checker.html`, 2013.

[39] W. Wang and S. Diestelhorst. Quantify the performance overheads of pmdk. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '18, page 50–52, New York, NY, USA, 2018. Association for Computing Machinery.

[40] M. Weiland, H. Brunst, T. Quintino, N. Johnson, O. Iffrig, S. Smart, C. Herold, A. Bonanni, A. Jackson, and M. Parsons. An Early Evaluation of Intel's Optane DC Persistent Memory Module and its Impact on High-Performance Scientific Applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. Association for Computing Machinery.

[41] K. Wu, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Towards an Unwritten Contract of Intel Optane SSD. In *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, Renton, WA, July 2019. USENIX Association.

[42] J. Xu and S. 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)*, pages 323–338, Santa Clara, CA, Feb. 2016. USENIX Association.

[43] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. Da Silva, S. Swanson, and A. Rudoff. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 478–496, New York, NY, USA, 2017. Association for Computing Machinery.

[44] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, pages 169–182, Santa Clara, CA, Feb. 2020. USENIX Association.

[45] S. Zheng, M. Hoseinzadeh, and S. Swanson. Ziggurat: A Tiered File System for Non-Volatile Main Memories and Disks. In *17th USENIX Conference on File and Storage Technologies (FAST 19)*, pages 207–219, Boston, MA, Feb. 2019. USENIX Association.

A Self-Reflection

Personal experience. Starting with this project, there was not much I knew about non-volatile memories, and I had never done any real systems research before. My expectations were, that I would do research for 2-3 months, write a thesis, present the results, and be finished with the research. Once I actually started the research, it sparked real interest for me. Being part of the early stages of new technology is very exciting, and with new technology comes the possibility for a great deal of future research, which makes it even more interesting. After this thesis, I will continue research in this field, alongside my future studies.

Besides sparking interest in this field of technology, this project gave me important lessons on research, and on larger projects. I personally, had little experience with projects of this scale, spanning several months. I have learned the importance of planning projects from the beginning, presenting and discussing findings with peers and supervisors, and most importantly, managing time properly. Managing time properly is very difficult, but setting a daily work schedule with goals and milestones, helps the progress and quality of the research significantly.

Time spent on research. Starting a new project is always difficult. This is why, the first 4-5 weeks of this research were spent on understanding this new technology and the new software stack, setting up the environment for our experiments, and generating our baseline. After this, experiments were typically done within a 1-2 week time span, one after each other, in the same order as they are presented in this thesis. This includes preparing experiments, either learning to use tools or coding the benchmark, running the actual experiment, and evaluating the results. After the evaluation of results, sometimes there was the need to further analyze certain behavior of performance, or to change the benchmark. This usually added an extra week to the experiment. A large part of the research was to bring all the collected data into this thesis, which was started alongside the running of experiments.

B Setup and Execution Commands

Commands for reproducing of benchmarks/results with included comments for important flags and/or required adjustments. **Most commands require root privileges.**

Listing 1: **Emulating PMEM.** If Kernel already has required modules (LIBNVDIMM, PMEM, DAX), skip to checking for available memory. Entire process is thouroughly explained in [26].

```
# Preferably download latest Linux Kernel (or any Kernel above 4.2)
# Create config and add required modules (LIBNVDIMM, PMEM, DAX)
$ make nconfig
# Compile Kernel (modify flag to number of threads)
$ make -j 12
# By default modules are installed with debug symbols, unless symbols are
    needed, add flag to minimize size of initrd
$ make INSTALL_MOD_STRIP=1 modules_install install

# Check available memory regions using one of two options
$ dmesg | grep BIOS-e820
# or
$ journalctl -k | grep BIOS-e820
# Reserve memory region by modifying the Kernel's command line arguments
# This will create 14GB of PMEM, starting at address 9G
  memmap=14G!9G
```

Listing 2: **Configuring PMEM Namespace.** Emulated PMEM is by default already configured to fsdax, but verify it or adjust to desired mode. Configuring using *ndctl* [3].

```
# Show available namespaces
$ ndctl list --human
# If mode is not 'fsdax', reconfigure
$ ndctl create-namespace -fe namespace0.0 --mode=fsdax
```

Listing 3: **Creating/mounting filesystems.** Creating file system on emulated PMEM and mounting file system. NOVA requires the needed modules, refer to Listing 5 for compiling kernel for NOVA.

```
# Creating ext4 or xfs file system on PMEM
$ mkfs.ext4 /dev/pmem0
$ mkfs.xfs -m reflink=0 /dev/pmem0 -f
# Mounting file system to mount directory
$ mount -o dax /dev/pmem0 /mnt/mem
# NOVA file system does initializing on mounting (NOVA module required)
$ mount -t NOVA -o init /dev/pmem0 /mnt/mem
# Verify if mounted correctly
$ mount | grep /dev/pmem0
# Unmounting file system when finished benchmarking
$ umount /dev/pmem0
```

Listing 4: **Emulate block device.** Emulating block device in RAM, for file system without DAX, and mounting file system.

```
# Creating a 14GiB block device in RAM (adjust to size needed)
$ modprobe brd rd_size=14680064 max_part=1 rd_nr=1
# Creating ext4 or xfs file system on the block device
```

```
$ mkfs.ext4 /dev/ram0
$ mkfs.xfs /dev/ram0 -f
# Mounting file system to mount directory
$ mount /dev/ram /mnt/mem
# Verify if mounted correctly
$ mount | grep /dev/ram0
# Unmounting file system when finished benchmarking
$ umount /dev/ram0
```

Listing 5: **NOVA.** Compiling Linux Kernel 5.1.0 with NOVA modules, which are required to use the NOVA file system.

```
# Clone Kernel 5.1.0 with NOVA modules
$ git clone https://github.com/NVSL/linux-nova
# Create config file from current Kernel's config file and select all
    required modules (LIBNVDIMM, PMEM, NOVA, DAX)
$ make nconfig
# Compile Kernel (modify flag to number of threads)
$ make -j 12
    # If error of wrong gcc version appears
    # install gcc-8
    $ update-alternatives --install /usr/bin/gcc gcc /usr/bin/gcc-8 1
# By default modules are installed with debug symbols, unless symbols are
    needed, add flag to minimize size of initrd
$ make INSTALL_MOD_STRIP=1 modules_install install
```

Listing 6: **Baseline memory bandwidth.** Measuring memory bandwidth using a memory mapped file, and reading/writing from/to the memory. Available at https://github.com/nicktehrany/membench. Since results have shown differenent performance for random and sequential access, we profiled hardware performance using perf [4] to identify the performance differences.

```
# adjust mode (read,write,randread,randwrite), and copysize in workload
    file
$ ./membench -file=workloads/baseline.txt

# Profiling multiple hardware counters to identify differences in random vs
    sequential access performance
$ perf stat -e cpu-cycles,l2_rqsts.all_pf,inst_retired.any ./membench -file
    =workloads/baseline.txt
```

Listing 7: **Baseline memory latency.** Baseline Benchmark to measure memory latency. Available at https://github.com/nicktehrany/membench

```
# Uses default 300MiB memory buffer (specify e.g. -size=400M for larger
    buffer) with 10 iterations
$ ./membench -engine=mem_lat -iter=10
```

Listing 8: **Filesystem I/O bandwidth.** FIO Benchmark to measure bandwidth for asynchronous I/O with a 1GiB file on mounted device.

```
# modify --rw (read,write,randread,randwrite), --directory to mount dir
# set buffered=1 for tmpfs
$ fio --directory=/mnt/mem --name=fio_bench --ioengine=libaio --iodepth=8
    --bs=4K --direct=0 --buffered=0 --size=1G --numjobs=1 --group_reporting
     --time_based --runtime=60 --rw=read --norandommap=1
```

Listing 9: **File operation latency.** For measuring latencies of file system operations we are using filebench [34]. We use three different workloads. One to create 100,000 64KiB files and write to the whole file. A second workload to overwrite data on the already created files. A third workload for deleting all the files. Filebench presents results as operations/second, from which we calculate the latency of one operation.

```
# Create a file called micro_create.f containing
set $dir=/mnt/mem # for tmpfs set to /dev/shm
set $nfiles=100000
set $filesize=0
set $nthreads=1
set $meandirwidth=10
set $appendsize=64k
set mode quit firstdone
define fileset name=bigfileset,path=$dir,size=$filesize,dirwidth=
    $meandirwidth,entries=$nfiles,prealloc=0
define process name=benchmark,instances=1
{
  thread name=benchmarkthread,instances=$nthreads
  {
    flowop createfile name=createfile1,filesetname=bigfileset,fd=1
    flowop appendfile name=append,filesetname=bigfileset,fd=1,iosize=
    $appendsize
    flowop closefile name=closefile1,fd=1
   }
}
run 30

# Create a file called micro_write.f containing
set $dir=/mnt/mem # for tmpfs set to /dev/shm
set $nfiles=100000
set $filesize=64k
set $nthreads=1
set $iosize=64k
set $meandirwidth=10
set mode quit firstdone
define fileset name=bigfileset,path=$dir,size=$filesize,dirwidth=
    $meandirwidth,entries=$nfiles,prealloc=100
define process name=benchmark,instances=1
{
  thread name=benchmarkthread,instances=$nthreads
  {
    flowop openfile name=open,filesetname=bigfileset,fd=1
    flowop writewholefile name=write,fd=1,filesetname=bigfileset,iosize=
    $iosize
    flowop closefile name=closefile1,fd=1
   }
}
run 10

# Create a file called micro_delete.f containing
set $dir=/mnt/mem # for tmpfs set to /dev/shm
set $nfiles=100000
set $filesize=64k
set $nthreads=1
set $meandirwidth=10
```

```
define fileset name=bigfileset,path=$dir,size=$filesize,entries=$nfiles,
    dirwidth=$meandirwidth,prealloc=100
define process name=deleter,instances=1
{
  thread name=deleter,instances=$nthreads
  {
    flowop deletefile name=deleting,filesetname=bigfileset,iters=$nfiles
    flowop finishoncount name=finish,value=$nfiles
  }
}
run 30

# Run each benchmark with respective file name
$ filebench -f micro_create.f
    # If error code 3 shows up run
    $ echo 0 > /proc/sys/kernel/randomize_va_space # as root
```

Listing 10: **Filesystem mmap latency.** Measuring the cost of mapping a file into memory for he different filesystems, and tracing the call stack of calling *mmap* on each filesystem.

```
# Store file with random data somewhere (adjust sizes however needed)
# /dev/urandom is very slow, quicker to get from it once, and copy that
    file
$ dd if=/dev/urandom of=file bs=1G count=1 iflag=fullblock
# Copy file to mount location
$ dd if=file of=/mnt/mem/file bs=1G count=1 iflag=fullblock,direct
# Run Benchmark on file (adjust flags if needed)
$ ./membench -engine=mmap_lat -dir=/mnt/mem/file -map_pop=0 -map_shared=0
    -iter=1000000

# Trace call stack of a single mmap call
$ trace-cmd record -p function_graph -l __x64_sys_mmap -F ./membench
    -engine=mmap_lat -dir=/mnt/mem/file -map_pop=0 -map_shared=0 -iter=1
# Redirect trace output to a file for reading or use tools such as
    kernelshark
$ trace-cmd report &> temp.txt
```

Listing 11: **Page fault latency.** Benchmark for measuring latencies of page faults, used for comparing cost of non DAX-enabled file systems to file systems without DAX. Results will show added latency of bringing page into page cache vs. providing direct access. Available at https://github.com/nicktehrany/membench. We additionally trace page faults to examine the DAX call stack.

```
# Store file with random data somewhere (adjust sizes however needed)
# /dev/urandom is very slow, quicker to get from it once, and copy that
    file
$ dd if=/dev/urandom of=file bs=1G count=4 iflag=fullblock
# Copy file to mount location
$ dd if=file of=/mnt/mem/file bs=1G count=4 iflag=fullblock,direct
# Run Benchmark on file
$ ./membench -engine=page_fault -dir=/mnt/mem/file

# Trace call stack of a page fault
$ trace-cmd record -p function_graph -l do_page_fault -F ./membench -engine
    =page_fault -dir=/mnt/mem/file
```

```
# Redirect trace output to a file for reading or use tools such as kernelshark
$ trace-cmd report &> temp.txt
```

Listing 12: **Key-value store bandwidth.** Measuring the bandwidth of PMDK's key-value store using pmemkv_bench [19]. We are measuing bandiwdth for different engines running the key-value store (volatile and persistent), and different data sizes.

```
# Add flag to cmake to build release version with debug symbols for profiling
$ cmake .. -DCMAKE_BUILD_TYPE=RELWITHDEBINFO
# Running benchmark (adjust engine,size, and all other flags as needed, some engines require a file name in --db flag)
$ ./pmemkv_bench --db=/mnt/mem --db_size_in_gb=9 --engine=vcmap --benchmarks=fillrandom,readrandom,deleterandom --num=45000 --reads=100000 --value_size=4096

# Profile pmemkv debug symbols
# Add libc debug version to perf buildid-cache, for libc debug symbols (path/version might differ)
$ perf buildid-cache -u /usr/lib/debug/lib/x86_64-linux-gnu/libc-2.31.so
# Run perf profiling with a 5 second delay
$ perf record -D5000 ./pmemkv_bench --db=/mnt/mem --db_size_in_gb=9 --engine=vcmap --benchmarks=fillrandom,readrandom,deleterandom --num=45000 --reads=100000 --value_size=4096
# Report the results in several ways
$ perf report --header
# or annotate the assembly of a function (adjust the function name)
$ perf annotate --symbol=pmem::kv::vcmap::get
```

Listing 13: **Cost of achieving persistence.** Benchmarking the latency of flushing a cache line with the PMDK, using the benchmark provided by it. We are measuing flushing of differently sized data to compare the cost.

```
# Create a file called pmembench_flush.cfg and safe the following in it (adjust numbers for flags as needed)
[global]
group = pmem
file = /mnt/mem/testfile.flush
ops-per-thread = 10000
repeats = 100
threads = 1
data-size = 4096
mode = rand
no-warmup = true
[flush_persist]
bench = pmem_flush
operation = persist

# Run the benchmark
$ ./pmembench pmembench_flush.cfg flush_persist
```

C Plot: Baseline Bandwidth Increased Hardware Prefetching on Sequential Access

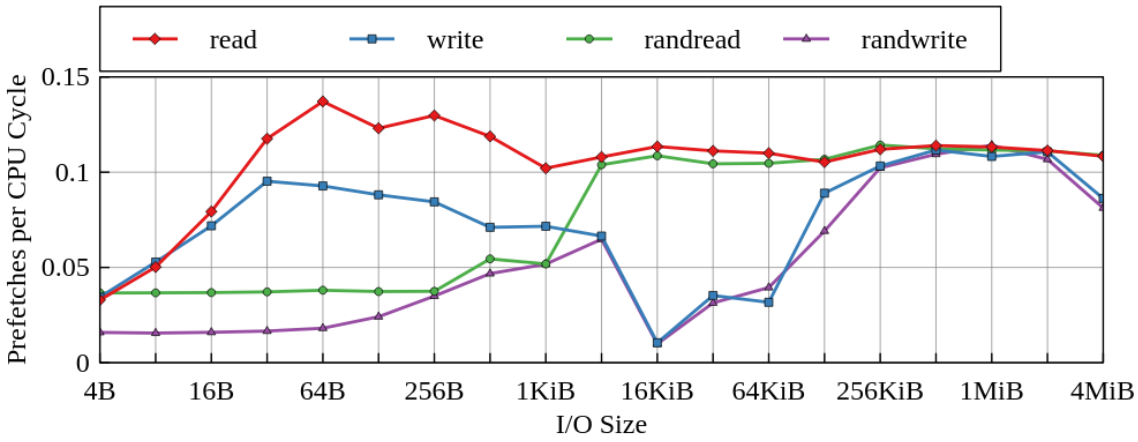

Figure 16: Baseline bandwidth benchmark, hardware prefetching per CPU cycle for the different access types. Shows increased prefetching for sequential access, corresponding to the increased bandwidth in Figure 7.

D Plot: PMDK Key-Value Store Hardware Prefetching Drop on Writing

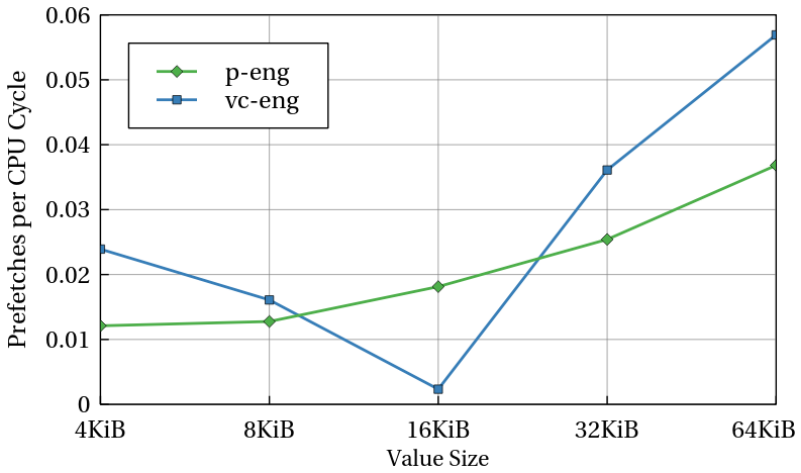

Figure 17: Prefetching of two of the pmemkv engines for sizes from 4KiB to 64KiB. Drop in prefetching explains the sudden drop at 16KiB data size, of writing bandwidth for volatile engines in Figure 12.