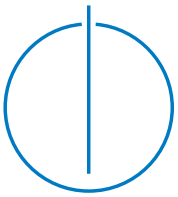
Department of Informatics
Technical University of Munich

Bachelor's Thesis in Informatics

Performance Evaluation of the PMDK Core Components and PMDK-enabled Applications

Robert Steve Jandow

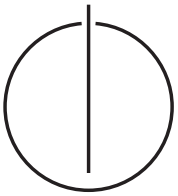


Department of Informatics
Technical University of Munich
Bachelor's Thesis in Informatics

Performance Evaluation of the PMDK Core Components and PMDK-enabled Applications

Performanzauswertung der PMDK-Kernkomponenten und Anwendungen mit PMDK-Unterstützung

Author: Robert Steve Jandow
Submission Date: August 15, 2021
Supervisor: Prof. Dr. Pramod Bhatotia
Advisor: Dimitrios Stavrakakis, M.Eng.



I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Garching, August 15, 2021 Robert Steve Jandow

Acknowledgments

Throughout the writing of this thesis, I have received a great deal of support and assistance. First and foremost, I would like to give a major thanks to my supervisor, Dimitrios Stavrakakis, for having so much patience with me and continuously nudging me in the right direction. His expertise in this rather complex topic of persistent memory and his valuable input have helped me considerably in completing this thesis.

Further, I would like to extend my sincere thanks to Prof. Dr. Pramod Bhatotia for his helpful advice and constructive suggestions. His door was always open whenever I ran into a problem or had questions about my research. Especially on the last mile, his support was invaluable for this thesis.

A special thanks to Lukas Vogel, who provided me access to a server with persistent memory hardware. I am very grateful for the opportunity to conduct my experiments on real persistent memory hardware.

I would also like to thank Nick Tehrany, Takashi Menjo, and Anuj Kalia for their time and help in understanding and interpreting the observed effects.

On a personal level, I want to thank my friends and family, who supported and encouraged me every day. Many thanks to my parents, Ramona and Uwe Jandow, for their endless support. Mom and Dad, thank you for all your love and for always believing in me. Thank you to my brother, Dennis Jandow, for cheering me up and helping to clear my mind.

Thank you to my amazing girlfriend, Julie Freyermuth, for answering a ridiculous number of phone calls, calming me down, and for supporting me anytime, anywhere.

Lastly, a big thanks to my close friends, Matthias Linhuber, Patrik Zander, and Kevin Su, for their unfailing support and for providing me valuable feedback.

Abstract

Nowadays, data is collected, stored, and processed on a large scale, necessitating fast and persistent storage. Persistent Memory (PM) aims to meet this demand by bridging the gap between main memory and storage. It combines the byte-addressability and low latency of main memory with the persistence and density of storage. In order to fully exploit its capabilities, developers have to adapt the novel concepts of persistent memory. The de-facto standard library for developing persistent memory applications is the Persistent Memory Development Kit (PMDK).

This thesis comprises a systematic in-depth performance evaluation of the PMDK, examining its characteristics at the micro and macro level. We develop a versatile open-source benchmarking suite using Docker and collect results representing approximately 570 hours of machine time on a server equipped with Intel Optane DC Persistent Memory Modules.

At the micro level, we assess the performance of the PMDK's core components and its PM-optimized key-value store `pmemkv` across a range of data sizes and thread counts. This approach provides valuable insights into the characteristics and synergies of the PMDK. At the macro level, we employ the YCSB to investigate the real-world performance of `pmemkv` and compare it with state-of-the-art database systems, illustrating the advantages of persistent memory for future applications.

Our results show that the PMDK's optimal data sizes are 64 B and 256 B. In addition, we observe that, especially in write-dominated workloads, a small number of threads is sufficient to achieve the maximum multithreaded throughput. The experiments further demonstrate that the adaption of the PMDK's concepts into the application design increases performance by up to four times. However, the concepts of persistent memory have to be integrated deep into the system architecture, otherwise the performance gain is only marginal.

Contents

1 Introduction

1.1 Motivation

For decades, computer systems have followed the two-level system stack (fast, volatile primary memory and slow, persistent secondary storage). This architecture has influenced the design of operating systems, file systems, software patterns, and their respective APIs. With the ever-increasing importance of data in today's digital society, the need for large and fast storage is also growing. Driven by the advancing digitalization, the global datasphere will exceed 180 zettabytes by 2025 [1]. Under this strain, the current two-level system stack is reaching its limits, thus posing new challenges for hardware manufacturers, software companies, and developers.

Non-volatile main memory (NVMM), commonly known as persistent memory (PM), aims to tackle these challenges by combining the persistence and capacity of traditional storage with the low latency and byte-addressability from conventional memory, bridging the gap between storage and memory. Persistent memory is directly connected to the memory bus alongside DRAM, allowing the CPU to use Load/Store instructions to access it. Unlike conventional memory, the data on non-volatile memory survives a power failure or system crash. The technology experienced a recent technological breakthrough with the first commercially available non-volatile dual in-line memory modules (NVDIMMs), the Intel Optane DC Persistent Memory Module (DCPMM) [2]. With the introduction of non-volatile memory hardware, it is now up to the developers to adapt their software design and architecture to realize the full potential of persistent memory.

There are two distinct approaches for utilizing persistent memory [3]: First, using the persistent memory hardware as an accelerated storage device. For this purpose, the direct access extension for file systems (DAX) allows the persistent memory to be mapped directly into the user space of an application, eliminating the intermediate step of copying the data into main memory. Second, utilizing libraries that can exploit the byte-addressability of persistent memory, empowering developers to take advantage of the fine-grained non-volatile memory access.

This thesis focuses on the latter approach with a performance evaluation of the Persistent Memory Development Kit (PMDK) [4], a platform-independent collection of libraries and tools for developers to fully exploit the capabilities of persistent memory. Among other features, it includes a low-level library for performing atomic memory operations, a higher-level API for transactional programming, and a persistent key-value store.

1.2 Research Questions

Currently used storage technologies have been available for years and are tailored to their areas of application. Persistent memory has the potential to revolutionize most of these areas. However, integrating and leveraging persistent memory poses new challenges to developers. New concepts such as cache flushing and persistent transactions have to be incorporated into the software architectures. For existing software, this may necessitate a complete redesign of the core data structures.

The PMDK seeks to become the foundation for these redesigns as well as future persistent memory applications. Investigating the characteristics and the overhead of the library can assist developers in identifying, understanding, and resolving issues in their applications. Consequently, we address the following research questions within the scope of this thesis:

- What is the amount of overhead associated with the usage of the PMDK? What is the overhead incurred in comparison to existing volatile methods?
- What are the general recommendations when using the PMDK? Are there particular synergies that should be exploited?
- How scalable are PMDK-supported applications? How do they compare to state-of-the-art technologies?

1.3 Contributions

This thesis presents a systematic in-depth performance evaluation of the PMDK software stack. We start by understanding the unique features of persistent memory. With this background knowledge, we evaluate the individual components of the PMDK and explore their performance characteristics as well as their scalability. Lastly, we examine the persistent key-value-store `pmemkv` [5] to get a sense of the PMDK's real-world performance. Our contributions can be summarized as follows:

- We provide an introductory overview of the available persistent memory hardware and the fundamentals of persistent memory programming. In order to use PMDK optimally, it is critical to understand the peculiarities of its underlying layers and concepts.
- We design and develop an open-source benchmarking suite for performance evaluation of the PMDK, consisting of a containerized benchmarking environment, configurations for various frameworks, and scripts for automatic execution.
- We provide a systematic in-depth evaluation of the PMDK's core components, resulting in a set of guidelines for developers.
- We present, to the best of our knowledge, the first detailed evaluation of the `pmemkv` using the *Yahoo! Cloud Serving Benchmark* (YCSB) [6], including a comparison with state-of-the-art database systems.

1.4 Structure

The remainder of this thesis is organized as follows: Chapter 2 starts with an introduction to the topic of non-volatile memory by shedding light on the functionality and characteristics of available hardware, namely Intel Optane DCPMM. Then, the structure of the PMDK and its available components is detailed. The design and implementation of our containerized benchmark suite are covered in Chapter 3. Chapter 4 opens with Section 4.1 describing the utilized experimental setup. Next, Section 4.2 evaluates the performance of fundamental functionalities such as persisting data and allocating persistent memory. In this context, the concepts of atomicity and transactions in the PMDK are explained as well. Using `pmemkv` as an example, Section 4.3 demonstrates the capabilities of persistent memory and its implications for future technologies. For this purpose, `pmemkv` is compared with current database technologies to better assess and classify its real-world performance. Afterwards, Chapter 5 portrays previous work done in the field of persistent memory, including hardware and library evaluations. In addition to the evaluations, the chapter further covers selected areas of application. Chapter 6 thoroughly examines our findings, discusses current limitations and presents potential future work. Finally, Chapter 7 concludes the thesis by summarizing our contributions and results.

2 Background

Before engaging in the topic of non-volatile memory, we want to define several essential terms and concepts. This chapter provides an introductory overview of hardware and software technologies. We start by presenting the characteristics of non-volatile hardware, with the focus on the Intel Optane Persistent Memory Module. Then, we explain the key concepts of persistent memory and how they differ from volatile ones. Finally, we introduce the PMDK and outline its structure and components in order to better understand its performance behavior later on.

2.1 Non-Volatile Memory

Typically, modern computer systems are designed for a strict bifurcation of devices into memory and storage devices.

Storage devices offer the highest capacity and lowest cost-per-bit for persistent storage. They are frequently implemented as block devices and thus cannot be accessed by the CPU using Load/Store instructions. As a consequence, storage devices are too slow for direct access and data has to be buffered into the main memory to accelerate application execution. A common technique in this context is *page caching* [7, Chapter 16]. The goal of page caching is to minimize data access to slower secondary storage devices by storing recently used pages in unused main memory. Whenever data from secondary storage is requested, the operating system first checks whether the requisite data is in the page cache. If that is not the case, the page containing the requested data has to be read from the slower storage device and is added to the page cache afterwards. When modifying data residing in the page cache, the whole memory page is marked as *dirty*. Periodically, all dirty marked pages are written back to disk. Therefore, small changes create a large I/O overhead as the whole page has to be written back to secondary storage. For reference, the typical Linux page size is 4 KiB [7].

Memory, on the other hand, is directly accessible by the CPU, but has a smaller capacity and is more expensive than storage. Moreover, it only provides volatile storage, which means that data stored in memory is lost in the event of a power outage or a system crash.

Non-volatile main memory (NVMM) aims to bridge the gap between memory and storage by offering fast, persistent, byte-addressable memory. Various technologies can be considered as persistent memory hardware, such as Phase Change Memory (PCM) [8], Spin-Transfer Torque RAM (STT-RAM) [9], and 3D XPoint [10]. All of them have in common that they offer a high density and low cost-per-bit while also being byte-addressable and achieving latency close to DRAM. The updated storage hierarchy, which now includes persistent memory, is depicted in Figure 2.1.

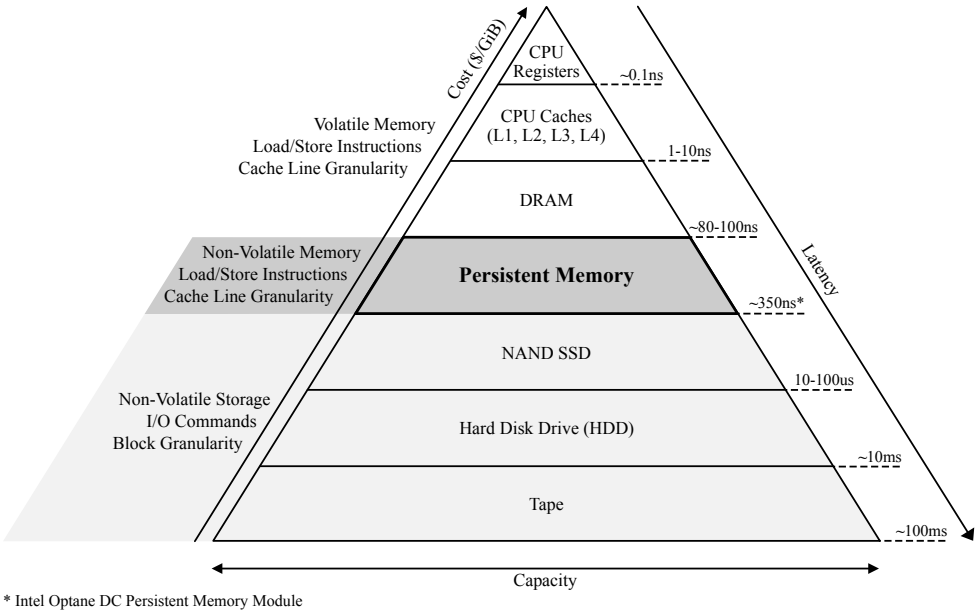


Figure 2.1: *Pyramid of the storage hierarchy with focus on latency, capacity, and cost. Persistent memory closes the gap between non-volatile storage and volatile memory. Figure based on [11]*

The first scalable, commercially available non-volatile memory hardware is the Intel Optane DC Persistent Memory Module [2], which is based on the aforementioned 3D XPoint technology. Throughout the thesis, we refer to it as DCPMM. The modules are available in three different capacities: 128 GB, 256 GB, and 512 GB per module.

Like conventional memory, DCPMMs are directly connected to the CPU's integrated Memory Controller (iMC) via the memory bus. A single iMC can support up to three DCPMMs. Hence, one processor can employ up to six DCPMMs across its two iMCs. The iMC is located inside the asynchronous DRAM Refresh domain (ADR), which guarantees that data reaching this domain will survive a power failure. Internally, the iMC maintains read and write pending queues for each DCPMM, ensuring that data is flushed to media on power failure. It should be emphasized that the ADR does not include the processor's caches. Stores are consequently only persistent once they reach the iMC [12]. However, there is ongoing research in the area of enhanced ADR (eADR), which also includes the CPU caches [13].

To communicate with the DCPMM, the iMC uses a proprietary DDR-T protocol [14], which has a lot in common with the DDR4 standard but has been adapted to the peculiarities of non-volatile applications. Just like DDR4 (with ECC), the interface for DDR-T uses a 72-bit data bus and transfers data in cache line (64 B) granularity between iMC and DCPMM [15]. Starting with the Cascade Lake processor family, Intel added CPU support for the DDR-T protocol and consequently for DCPMM. Therefore, DCPMM support is not available on prior Intel CPU generations [12, 16].

The DCPMM itself contains an onboard controller that coordinates the accesses to the 3D XPoint media by performing wear-leveling and bad-block management. As the physical media access granularity of 3D XPoint is 256 B (XPLine) [12], the controller includes a small write-combining buffer in the size of 256 B, coalescing adjacent 64 B DDR-T writes into larger 256 B media writes. As a result, the optimal access size for DCPMM is 256 B [12, 15, 17]. The communication between iMC and DCPMM is illustrated in Figure 2.2.

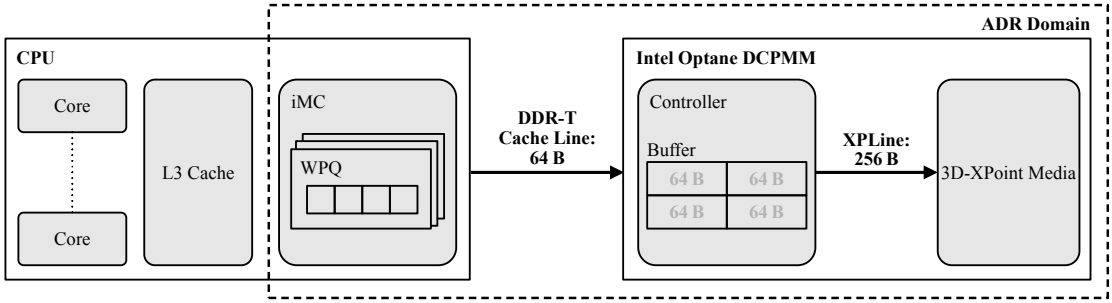


Figure 2.2: *Communication structure between CPU and Intel Optane DCPMM with focus on the components of the ADR domain*

Latency and bandwidth are key memory technology parameters. Yang *et al.* [12] demonstrate in their evaluation of the DCPMM that the average read latency is two to three times higher than DRAM. Since both DRAM and DCPMM use the iMC to commit data to media, they perform similarly in terms of write latency [12]. Regarding the performance characteristics of a single Intel DCPMM DIMM, Intel specifies the sequential bandwidth for reads with 7.6 GB/s and for writes with 2.3 GB/s. As for the random bandwidth, Intel quantifies the bandwidth with 2.4 GB/s for reads and 0.5 GB/s for writes [18, p.350]. Several publications verify these numbers [12, 15, 17].

When looking at the performance numbers, two things stand out: First, the random bandwidth is significantly lower compared to sequential bandwidth. Second, the performance of read and write operations is asymmetrical, with writes being the slower of the two. From this, it can be deduced that data structures with primarily random writes and a high write amplification should be avoided when working with DCPMMs. Before DCPMM became available, researchers used emulation to validate and test their non-volatile memory applications [19–22]. These emulations often add artificial latency to data accesses and limit the overall bandwidth. However, the previously mentioned empirical analysis by Yang *et al.* [12] indicates that these emulations have failed to reflect the distinctive properties of DCPMM. Characteristics like the internal 256 B granularity and the asymmetrical performance of read/write operations were not incorporated into the prior emulations, resulting in less meaningful insights.

The DCPMM has two operation modes: *Memory Mode* and *App Direct Mode* [18, p.347]. In *Memory Mode*, the hardware acts as a larger volatile main memory. In this mode, DCPMM is transparent to the operating system and applications. To hide the longer latency and lower bandwidth, DRAM is used as a *L4 cache* [18]. Especially applications

requiring larger DRAM capacity can benefit from this mode (e.g. scientific simulations [14]).

In *App Direct Mode*, the DCPMM is directly exposed as a non-volatile memory device separated from DRAM. For the operating system, the DCPMM and the DRAM appear as individual entities. Applications can now use the non-volatile memory either as an accelerated block device (*Storage over App Direct Mode*) or access it directly using CPU instructions on memory-mapped files (*App Direct Mode*). To utilize the full potential of persistent memory, the DCPMM should be used in *App Direct Mode* in conjunction with a PM-aware file system and the DAX (direct access) extension for file systems [23]. File systems currently supporting DAX are *xfs*, *ext2*, and *ext4*. DAX enables direct access to the files by bypassing the page cache. Consequently, neither an intermediate copy to main memory nor synchronization to storage is required [3].

In this thesis, we solely use the DCPMM in the latter mode in combination with DAX and memory-mapped files to fully exploit its persistent capabilities.

2.2 Concepts of Persistent Memory Programming

Most programmers are aware of the distinct properties of memory and storage. Consequently, they think in terms of memory-based and storage-based data structures. Storage-based data structures are usually intended to store data for extended periods of time and across application and system restarts. However, a consistent state has also be maintained in the event of a system crash. A common solution for ensuring consistency in storage-based data structure is logging, which persists all operations before executing them. This technique has proven itself in transactional systems [24, 25].

In contrast to storage, memory contents are typically cleared between application runs. As a result, memory-based data structures only require developers to maintain consistency at runtime. Techniques such as *locking* are frequently used to ensure this runtime consistency [3].

In the matter of persistent memory, both storage and memory considerations apply. The application is responsible for keeping states consistently not only during runtime but also between runs and reboots. This poses new challenges to developers working with persistent memory.

Each platform provides a *power-fail protected domain*, also called *persistence domain* [3]. Depending on the system's hardware configuration, the *power-fail protected domain* may include the persistent memory, the integrated memory controller (iMC), and the CPU caches. Data entering this domain is presumed to be persistent and recoverable in case the system is restarted. On Intel platforms, as seen in Section 2.1, this domain is also known as asynchronous DRAM Refresh domain (ADR) and does not include the CPU's caches [3]. As a result, data from the volatile cache may not have reached the *power-fail protected domain* and therefore cannot be recovered with certainty in the event of a system crash. To ensure persistence, the data in the processor's cache has to be flushed continuously to memory using the `CLFLUSH`, `CLFLUSHOPT`, and `CLWB` instructions of the Intel ISA [18]. All those instructions serve the same purpose of flushing the volatile cache but differ in their details.

CLFLUSH, for example, triggers a cache line flush but at the same time invalidates the cache line, resulting in performance degradation due to more cache misses [26]. To improve performance, Intel introduced two additional instructions. CLFLUSHOPT is conceived as optimized CLFLUSH by allowing concurrency when executing multiple CLFLUSHOPT instructions back-to-back. CLWB behaves similarly to CLFLUSHOPT but does not invalidate the cache line, increasing the likelihood of a cache hit. However, Kalia *et al.* [26] recently discovered that on current processor generations, CLWB and CLFLUSHOPT behave identically. As they point out, the documentation of CLWB states that "the line *may* be retained in the cache hierarchy in a non-modified state" [26, p.12].

Another essential operation related to flushing the CPU cache is SFENCE [3]. The SFENCE instruction creates a memory barrier by serializing all pending stores, preventing unwanted reordering of flushes. Thus, when the aforementioned flushing operations are combined with a subsequent SFENCE, the correct order of stores reaching the *power-fail protected domain* can be guaranteed before resuming. Importantly, all these instructions can be invoked from user space, enabling programs to control when and where data is flushed and fenced.

2.3 Persistent Memory Development Kit (PMDK)

Introduced in 2014, the PMDK was originally named NVML [27]. It is a vendor and platform independent collection of open source libraries and tools for persistent memory developed by Intel. The PMDK libraries build on the SNIA NVM programming model [28]. They were developed alongside the advancement of operating system support for persistent memory, allowing the libraries to take full advantage of the most recent features exposed by the operating system [3].

The PMDK provides two categories of libraries: volatile and persistent ones. This thesis focuses on the persistent components of the PMDK. The persistent libraries assist developers in building applications that are consistent and fail-safe. To further facilitate programming, the PMDK's libraries can automatically detect platform optimizations and select the appropriate durability semantics and memory transfer mechanisms for persistent memory [3].

All of the PMDK's persistent functionality builds on top of the two low-level C libraries, libpmem [29] and its successor libpmem2 [30]. Both provide developers raw access to the persistent memory devices and an easy-to-use interface for CPU instructions like CLWB and SFENCE. Due to the direct access to persistent memory primitives, these libraries can be utilized to implement low-level persistent data structures with complete control over the memory management and the recovery logic. Furthermore, they can be used to migrate existing applications that already employ memory-mapped files to persistent memory.

Besides the low-level functionality of libpmem, the PMDK also provides the higher-level library libpmemobj [31]. libpmemobj builds upon libpmem to realize more advanced data structures and programming concepts such as memory management and locking. With the help of this library, the memory-mapped files of libpmem can be transformed into a versatile object-store. libpmemobj further provides APIs for atomic operations, transactions, and

reserve/publish with the guarantee of data integrity.

`libpmemobj-cpp` [32] is another library extending `libpmemobj` with the meta-programming features of C++ (e.g. static type-system). It primarily addresses developers who want to make their volatile projects compatible with persistent memory.

Lastly, `pmemkv` [5] should also be mentioned. This library implements a PM-optimized embedded key-value store with put/get-interface. In addition to the C and C++ interfaces, there are numerous bindings for other programming languages (e.g. Java, JavaScript) [33–36].

There are, of course, more libraries and tools in the PMDK to choose from [4]:

- `libpmemlog`: PM-resident log file for frequently logging applications
- `libpmemblk`: PM-resident array of same-sized blocks that update atomically
- `libvmmalloc`: conversion of memory allocations into persistent memory allocations
- `libpmempool`: support for off-line pool management and diagnostics
- `librpma`: access to persistent memory over Remote Direct Memory Access (RDMA)
- `libvmemcache`: embeddable and lightweight in-memory caching solution

For the purpose of this thesis, we are going to focus on the major persistent libraries, whose hierarchy is depicted in Figure 2.3.

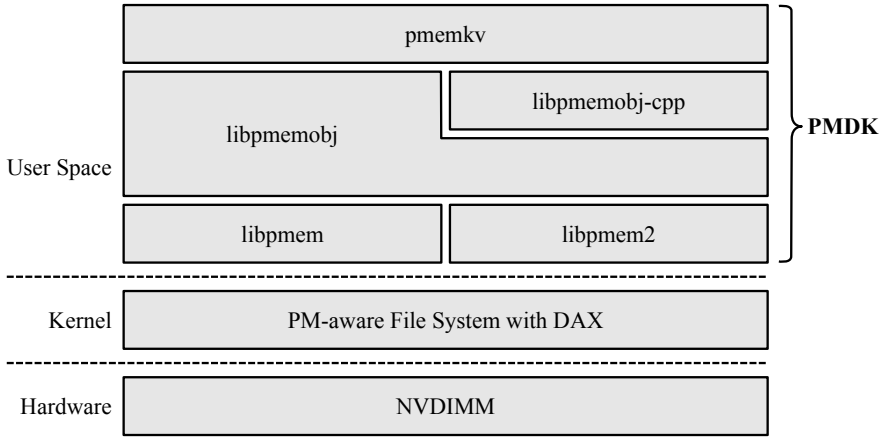


Figure 2.3: *Hierarchy of PMDK libraries evaluated in this thesis*

3 Design and Implementation

Having established the required background knowledge, this chapter explains the design and implementation decisions involved in the development of our benchmarking suite.

3.1 Design

In literature, there are numerous ways to evaluate performance [37–40]. For our performance analysis of the PMDK's core components, we decide to use microbenchmarks. Microbenchmarks are designed to investigate a specific aspect of a system [40]. We complement these benchmarks with selected parameters to capture the performance characteristics of each component. This approach is subsequently also used to determine the performance properties of the different operations in `pmemkv`. In addition, we employ macrobenchmarks in the form of the *Yahoo! Cloud Serving Benchmark* (YCSB) [6] to quantify the overall performance of `pmemkv` and compare its performance to other modern database systems.

pmembench The PMDK includes its own internal benchmark framework `pmembench` [41], to allow developers and maintainers to measure, analyze, and optimize the performance of each component individually. The framework consists of the main library, a set of support modules handling the measuring and the output format, and the benchmark modules that implement the actual functions to be examined. Each benchmark module adopts the same structure consisting of an initialization, the actual execution, and the subsequent cleanup. In addition to the predefined options, each benchmark can declare its own custom options and flags. As a result, developers can implement a comparison of multiple operations and alter operations depending on the selected option. Furthermore, they have the opportunity to declare the properties of their benchmark scenarios individually. In particular, they can specify whether their benchmark is designed to support multithreading and multi-operation execution.

During execution, the `pmembench` measures a variety of execution parameters such as latency, throughput, and the total duration of the workload. Additionally, the framework features the calculation of important mathematical quantities such as standard deviation or various percentiles. In order to simplify the creation of workloads, the framework provides the ability to specify the benchmark options either via command-line arguments or as a configuration file. The configuration file has the advantage of supporting the declaration of value ranges. This feature is particularly useful when observing the progression across a variety of values for one parameter, e.g. multithreaded scaling from 1 to 16 threads.

pmemkv-bench Benchmarking the individual operations of the `pmemkv` is done using `pmemkv-bench` [42]. This framework is ported from LevelDB's `db_bench`, a well-known performance benchmark for databases [43]. The tool provides microbenchmarks for *read*, *write*, and *delete* operations. Each operation can be executed with either random or sequential key order. Aside from the fundamental benchmarks, `pmemkv-bench` also includes

benchmarks addressing the multithreaded behavior of concurrent *read* and *write* operations. This functionality enables developers to investigate the behavior of `pmemkv` for different access patterns and operations.

Unlike the internal PMDK benchmarking framework, `pmemkv-bench` does not support configuration files. Thus, all workloads have to be specified using command-line arguments only. Mandatory arguments are the location of the database file and its size. Additionally, one can specify the `pmemkv` engine, the number of entries and operations, the key and value size, as well as the number of threads.

YCSB We use the *Yahoo! Cloud Serving Benchmark* (YCSB) [6] to evaluate the real-world performance of `pmemkv`. The YCSB is a popular key-value store benchmarking framework originally developed by Cooper *et al.* [44]. Its primary objective is to facilitate the "performance comparisons of the new generation of cloud data serving systems" [44, p.1].

The framework includes two components, the YCSB client and a set of workloads, called the *YCSB Core Package*. The YCSB Client itself is a Java program containing two parts: the extensible workload executer and the database interface layer. The workload executor uses multiple threads to generate a series of operations and execute them by calling the database interface layer. In general, a workload consists of two phases: the load phase to fill the database and the transaction phase to execute the actual workload. In addition to the YCSB client, Cooper *et al.* developed the *YCSB Core Package*, a set of workloads to evaluate the different aspects of database performance. Each workload comprises a unique mix of *read*/*write* operations, data size, and request distribution. A detailed list of the core workloads is presented in Table 3.1. During the design of the YCSB, special attention was paid to extensibility. Consequently, developers can easily implement custom workloads and add database interfaces to the existing framework. Instead of developing new workloads, one can also modify existing core workloads using command-line arguments or parameter files. Important adjustable workload properties include the record count, the number of operations, and the selected database interface.

Workload	**Operations**	**Distribution**
A (Update Heavy)	Read: 50% Update: 50%	Zipfian
B (Read Heavy)	Read: 95% Update: 5%	Zipfian
C (Read Only)	Read: 100%	Zipfian
D (Read Latest)	Read: 95% Insert: 5%	Latest
E (Short Ranges)	Scan: 95% Insert: 5%	Zipfian/Uniform

Table 3.1: *The workloads of the YCSB Core Package based on [45]*

3.2 Implementation

As recommended by Ousterhout [39], we automate our experiments by building a complete benchmark suite based on Docker containers. Docker provides a consistent benchmarking environment while being versatile and easy to use. In an early analysis, Giles [46] examines the current approaches for utilizing byte-addressable non-volatile memory in container-based virtualization. The author details three methods to expose persistent memory to containerized applications: Docker Storage, Docker Volume, and Docker Direct Device Access. For our use case, Docker Storage and Volume are the least favorable options, as they expose persistent memory to the container via a slower image layer and driver [46]. Due to the intermediate layers, applications inside the container cannot leverage the byte-addressability of persistent memory. This fact makes them unsuitable for our performance evaluation. With this knowledge in mind, we choose Docker Direct Device Access to ensure that applications can still take advantage of the byte-addressability of non-volatile memory while tolerating reduced application isolation and portability. Nevertheless, the behavior of containerized persistent memory should be further investigated in future studies.

For better isolation and durability, each framework is located in its own Docker container. An advantage of this structure is that the individual Docker containers are smaller and can be rebuilt quickly in case of changes to experiments. Inside the container, we use one main script as entry point for our benchmarks. Thus, a single command-line argument can be used to execute individual experiments or all of them in succession. In the main script, the general parameters for all experiments are defined, such as the path to the benchmarking framework, the mounting point of the persistent memory, and the destination directory for the results.

All benchmarks are invoked with the `PMEM_IS_PMEM_FORCE=1` flag of the PMDK [29]. This option, as the name implies, instructs `libpmem` and `libpmem2` to treat the given memory-mapped file as persistent. Generally, this flag is not required, but it is unclear how the Docker Direct Device Access would affect the persistent memory primitives exposed by the host operating system. For this reason, we manually set the flag to ensure a consistent behavior of the PMDK. Another benefit of using this flag in combination with Docker is that it allows us to mount volatile devices into the Docker container and simulate their persistent behavior. For example, one can easily mount DRAM via `/dev/shm` into the container and mimic its persistent performance by replacing the host mounting point in the Docker command.

To avoid fluctuations in our measurements, we repeat all experiments several times. Each data point averages 100 consecutive runs for `pmembench` and 10 consecutive runs for `pmemkv-bench` and the YCSB. With this approach, we minimize the impact of temporarily occurring load peaks on the measurements. As an additional precaution, we manually monitor the experiments to ensure that the results were not distorted by any background processes.

All of our scripts, configurations, and Dockerfiles are open source to encourage other researchers and developers to evaluate and verify our results [47].

As part of our implementation, we use multiple libraries and tools to evaluate the performance of the PMDK. Table 3.2 lists all of them with their corresponding version to help comprehend and classify the findings of this work.

Name	**Version/Commit**
PMDK [4]	1.11.0
libpmemobj-cpp [32]	1.12
pmemkv [5]	1.14
pmemkv-java [33]	1.0.1
pmemkv-bench [42]	d26c3a1
memkind [48]	1.11
Docker [49]	20.10.2
PostgreSQL [50]	9e7dbe3
PMEM-Redis [51]	06c077a
MongoDB [52]	3.5.13
PMSE [53]	6ff2abc

Table 3.2: *Overview of all employed libraries/tools with their corresponding version*

pmembench The PMDK, as stated above, has its own extensible benchmarking framework. The pre-existing benchmarks already cover a large part of the PMDK's core components. For these benchmarks, we create our own configuration files to highlight and demonstrate the PMDK's characteristics.

However, since not all of our areas of interest are covered, we create our own benchmarks and complement the existing ones [54]. Our changes include:

- Implementing the additional operations `pmem_flush()` and `pmem_deep_persist()` for the low-level benchmark `pmem_flush`
- Extending the `pmem_memcpy` and `pmem_memset` benchmark with a plain `memcpy()` without persistence and `memset()` without persistence, respectively
- Adding a completely new benchmark for `libpmem2`, since it was not yet represented in `pmembench`

pmemkv-bench Unlike the PMDK's internal benchmarking tool, `pmemkv-bench` does not support parameter declaration via configuration files. As a consequence, the scripts for this section are more detailed since they have to serve additionally as configurations. Furthermore, the convenient post-processing features of PMDK's internal benchmarking framework, such as generating value ranges and automatic average calculation, are not present in `pmemkv-bench`. Hence, we rebuild those functionalities in our scripts. For example, we use loops to execute each benchmark multiple times and automatically report the mean.

While preparing the workloads for `pmemkv-bench`, we noticed that the figures for random *read* operations from other papers were significantly higher than our measurements [12, 22]. The discrepancy is shown in Figure 3.1.

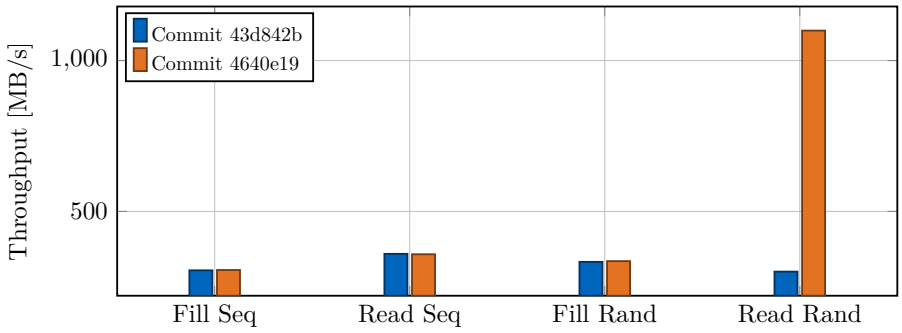


Figure 3.1: *pmemkv-bench: Comparison between pmemkv-bench commit 43d842b and commit 4640e19 (1 M entries and 10 M read operations).*

After investigating the problem, we were able to identify a change in the key calculation, resulting in a higher miss rate and consequently in an inferior performance. The error occurs as soon as the number of *read* operations exceeds the number of entries, as depicted in Figure 3.2.

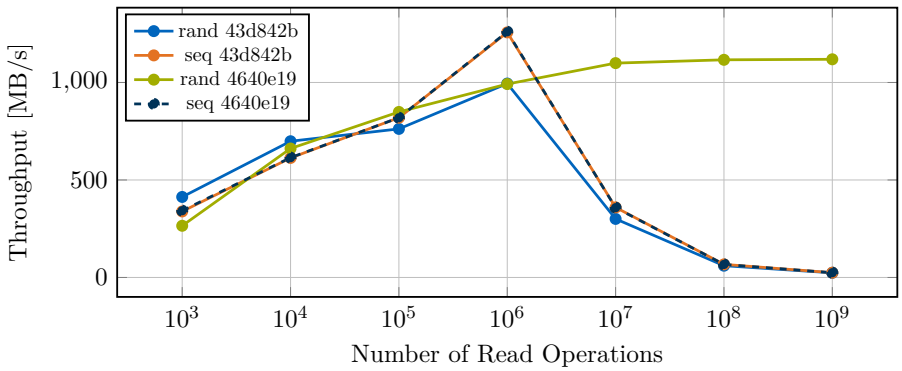


Figure 3.2: *pmemkv-bench: Comparison between pmemkv-bench commit 43d842b and commit 4640e19 with respect to different ratios between entries (1 M) and read operations.*

In earlier implementations, the key calculation is performed using the following function

$$rand() \mod N \equiv key \implies key \in \{0, \ldots, N-1\}$$

where N is the number of entries and R is the number of reads. The formula indicates that the miss rate is always 0% because the calculated key is definitely contained in the database, which comprises N entries with sequential key values from 0 to $N-1$. Starting with commit 43d842b, the calculation changes to

$$rand() \mod \mathbf{R} \equiv key \implies key \in \{0, \ldots, N-1, \ldots, \mathbf{R\text{-}1}\}$$

The miss rate now depends on the difference between the number of entries and the number of operations. We express the miss rate as

$$missrate = \begin{cases} \frac{R-N}{R}, & \text{if } R \geq N \\ 0, & \text{otherwise} \end{cases}$$

To obtain a more realistic analysis of `pmemkv`'s reading performance, we modify the implementation of random reads in `pmemkv-bench` [55]. With our changes, it is now possible to set the desired miss rate for read operations and thus tailor it to different areas of application. Our implementation is based on the earlier function, which provided a 0% miss rate. We additionally add two counters, one for misses and one for hits, and alternate randomly between miss and hit. Each counter is decremented when its associated method is executed. When one counter becomes zero, the remaining opposing operations are executed. With this approach, we ensure that the miss rate is achieved and the operations are not executed in batches, e.g. first all hits and afterwards all misses.

YCSB The YCSB is a popular framework for evaluating the performance of key-value stores. At the time of creating our benchmark suite, there was no database interface for `pmemkv` available. As previously seen, the YCSB is easily extendable. We therefore decided to implement our own corresponding database interface [56] by using the `pmemkv` Java binding [33]. With our implementation, we were able to get a first overview of `pmemkv`'s real-world performance and refine the YCSB core workloads to best demonstrate its benefits and weaknesses. With the release of Intel's own `pmemkv` YCSB implementation and its slightly higher performance, we decided to migrate our experiments to this version.

To better classify the YCSB performance of `pmemkv`, our benchmark suite also includes three PMDK-enabled databases. The first database is MongoDB. We employ it in combination with Intel's persistent memory storage engine (PMSE) [53]. PMSE uses the PMDK to manage MongoDB's data transactionally and to eliminate the need for snapshot and/or journal creation. Our second PMDK-enabled database is a modified version of Redis [51]. Typically, Redis operates as in-memory key-value store. To ensure persistence, Redis logs all write operations to an *append-only file* (AOF). At regular intervals, Redis flushes the AOF to persistent storage. Our employed version of Redis, however, uses PMDK's `libpmemobj` to ensure its persistent state and thus no longer requires an AOF [51]. The last database we include is PostgreSQL. We use PostgreSQL in conjunction with patches proposed by Menjo [57]. These patches accelerate PostgreSQL by using the PMDK's `libpmem` to optimize the Write-Ahead-Logging (WAL). More specifically, the patches replace the POSIX system calls with `libpmem` functions. Instead of opening, reading, and writing files, the modified version of PostgreSQL uses memory mapping, memory copying, and flushing.

Throughout all YCSB experiments, we use our scripts for post-processing of the measurements and pre-processing of the database. Unlike `pmemkv`, most modern database systems run independently as a standalone application in the background. Therefore, the scripts have to initialize the databases before the YCSB workloads can be executed. The initialization stage includes spawning of the database process, user management and, where applicable, the creation of the necessary database tables.

4 Evaluation

This chapter presents the results of the PMDK's performance evaluation. First, we give an overview of the used system specifications to conduct our experiments. We then evaluate the performance of the PMDK's core components and highlight important characteristics. Lastly, the performance of `pmemkv` is examined and compared with other state-of-the-art database systems using the YCSB.

4.1 Experimental Setup

We conduct our benchmarks on a server powered by a single 24-core Intel® Xeon® Gold 6212U processor. The CPU has two iMCs and a total of six memory channels (three channels per iMC). Each memory channel is populated with a 32 GB DDR4 DIMM and a 128 GB Intel Optane DCPMM. Thus, the processor is backed by 192 GB of DRAM and 768 GB of persistent memory. The machine is running Ubuntu 20.04.2 with Linux kernel version 5.4.0. A detailed server specification can be found in Table 4.1.

CPU	Intel® Xeon® Gold 6212U
Frequency	2.4 GHz (Turbo 3.9 GHz)
# Cores	24 physical (48 virtual)
L1 Cache (per Core)	64 KiB (L1I + L1D)
L2 Cache (per Core)	1 MiB
L3 Cache (Shared)	35.75 MiB
DRAM	192 GB (6× 32 GB)
Persistent Memory	768 GB (6× 128 GB DCPMM)
Operating System	Ubuntu 20.04.2 (Kernel v5.4.0)

Table 4.1: *Specifications of the benchmarking server*

As previously stated, we mainly focus on the persistent aspects of the PMDK. Therefore, the Intel Optane DCPMM operates in the prior mentioned *App Direct mode*. In our system, the persistent memory hardware is formatted with the *ext4* file system and mounted with enabled DAX (direct access) support [23].

4.2 PMDK Component Benchmarks

The PMDK consists of several libraries and tools to help developers manage and utilize persistent memory in their applications. In this chapter, we evaluate and assess the performance of the PMDK's core components. As mentioned previously, we primarily focus on the persistent aspects of the PMDK, thus omitting libraries like `libvmem` and `libvmmalloc`. For performance evaluation, we use the internal benchmarking framework of the PMDK, called `pmembench` [41]. This framework lets us evaluate each aspect of the PMDK individually. We complement the framework with our own scripts, configuration files, and benchmarks, as detailed in Chapter 3.

4.2.1 Persisting Data (`libpmem`)

`pmem_persist()` and `msync()`

Background `libpmem` [29] provides a non-transactional interface for the low-level persistent primitives such as cache flushing, file mapping, and raw access to persistent memory.

As discussed earlier in Section 2.3, the PMDK uses memory-mapped files to access the persistent memory. In `libpmem`, mapping files is done using the `pmem_map_file()` function. Depending on the capabilities of the system, `libpmem` automatically determines the optimal method for memory-mapping files and the optimal set of CPU instructions for cache flushing. After mapping a file into the application's address space, the developer is responsible for selecting the appropriate persisting mechanism. `libpmem` offers two basic functions for persisting: `pmem_persist()` for persistent memory and `msync()` for both persistent memory and traditional storage devices. In general, `pmem_persist()` should always be used when dealing with persistent memory. Unlike `msync()`, `pmem_persist()` does not call into kernel space resulting in less overhead and better performance.

Internally, `pmem_persist()` consists of two stages: First, `pmem_flush()` to flush the processor's caches. Second, `pmem_drain()` to ensure that the hardware buffers have drained and the data has reached the actual media. On Intel platforms with persistent memory support, data is considered persistent once it reaches the iMC, so draining the hardware buffers is not required. Instead, `pmem_drain()` acts as an `SFENCE` operation to prevent reordering and to ensure that all previous store instructions have been completed [58].

In addition to the aforementioned persisting methods, there is a third method called `pmem_deep_persist()`. In order to increase the reliability, `pmem_deep_persist()` evicts the data to "the most reliable domain available to software" [59] instead of only flushing it to the write-pending queue of the CPU's iMC, like `pmem_persist()`.

Our objective in this section is to analyze and quantify the performance difference between the three offered methods as well as the overhead introduced by more reliable persisting mechanisms. To demonstrate the performance characteristics, we compute each method on a single thread in sequential and random access patterns for varying data sizes reaching from 8 B to 8 KiB. Furthermore, we examine the scalability of all operations with different numbers of threads and a random access pattern. For this purpose, we choose three distinct data sizes: 64 B (cache line size), 256 B (size of the DCPMM's write-combining buffer), and 4 KiB (default Linux page size). For all experiments, we set the number of

operations per thread to 100,000 and perform a warmup beforehand to eliminate the effect of page allocation on our measurements.

Evaluation Overall, we can observe two major tendencies: First, the additional step of draining the hardware buffers results in reasonable overhead. Second, `msync()` and `pmem_deep_persist()` are significantly slower than `pmem_persist()` in every scenario. One reason for the slow behavior of `msync()` is the *huge page* support [60]. As Iwata [61] points out, using `msync()` for persisting sets the dirty flag of an entire 2 MiB *huge page*, which leads to its invalidation and consequently flushing of the whole page.

Looking at the figures in detail, we see two distinct behaviors for random and sequential data access. The left plot in Figure 4.1 depicts the behavior for persisting data in random order. The throughput remains at a constant level up to 256 B before it slowly starts to decline. Based on this observation, we deduce that the data size is of secondary importance for evicting data in random order as long as it remains smaller than 256 B.

For sequential flushing, shown in the right plot of Figure 4.1, the throughput progression is not as even. We observe a sudden peak in throughput for both `pmem_flush()` and `pmem_persist()` at 64 B. To verify this finding and exclude any side effects of the containerized application, we reran these experiments outside of the Docker container and obtained similar performance numbers, with 64 B being the optimal data size. This behavior can be explained by the cache line size of 64 B. Hence, we infer that flushing a single cache line is the most efficient approach for persisting data.

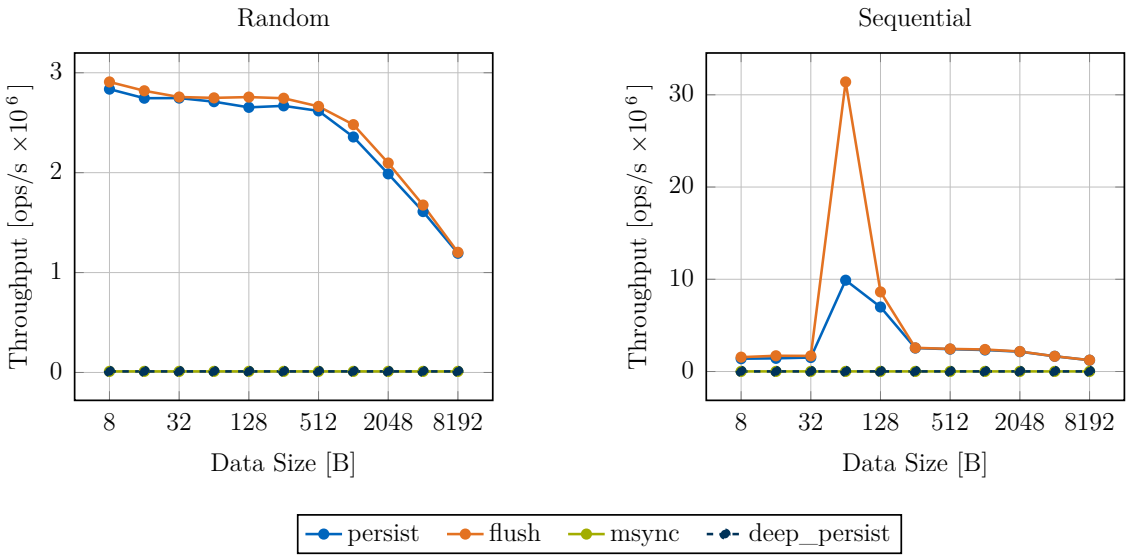


Figure 4.1: *PMDK benchmark: Performance of persist operations for varying data sizes*

Comparing `pmem_flush()` and `pmem_persist()` at the data size of 64 B, it can be seen that `pmem_flush()` is approximately 3× faster than `pmem_persist()`. These measurements illustrate the performance impact of `SFENCE`. On Intel platforms, like our experimental server, the additional `pmem_drain()` in `pmem_persist()` only serves as an ordering barrier

using SFENCE. As visualized by our plots, this additional instruction has a significant impact on the throughput, because the CPU cannot reorder stores to optimize performance. However, it should be emphasized that these barriers are critical to guarantee the persistence order of consecutive operations.

Returning to the remaining sequential data sizes in Figure 4.1, we observe that those larger than 64 B have a higher throughput than those below 64 B. Based on this observation, we deduce that data should be at least in the size of a cache line (64 B) to ensure efficient sequential flushing. To support and verify our finding, we further examine the latency per cache line for the `pmem_persist()` operation. As seen in Figure 4.2, the latency per cache line for sequential flushing remains steady for data sizes up to 256 B (size of the DCPMM's write-combining buffer). This finding corresponds with previous analyses of DCPMM hardware, which show that data sizes smaller than 256 B do not imply faster accesses [12, 15, 17].

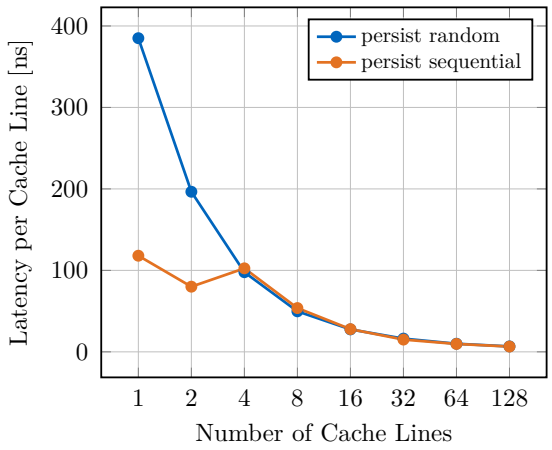


Figure 4.2: *PMDK benchmark: Latency per cache line for random and sequential persist operations on data sizes between 64 B and 8 KiB*

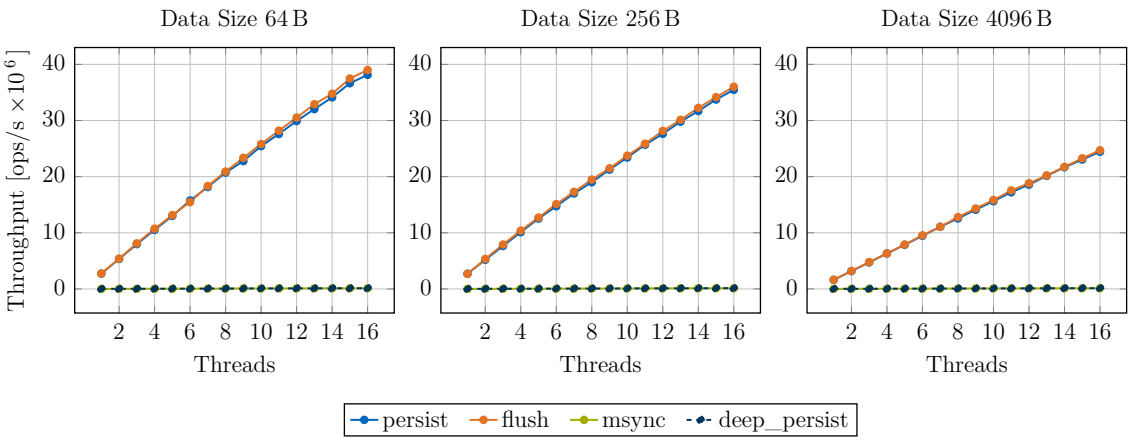


Figure 4.3: *PMDK benchmark: Multithreaded performance of persist operations*

Concerning the multithreaded scaling, Figure 4.3 shows that all data sizes scale linearly with an increasing number of threads. As with single-threaded performance, `msync()` and `pmem_deep_persist()` provide significantly worse performance. `pmem_persist()` and `pmem_flush()` offer a comparable performance overall, with differences becoming more visible as the number of threads increases. Comparing the throughput of the individual data size across the plots, we see that smaller data sizes tend to perform better.

Summary Based on our plots, we conclude that `pmem_persist()` and `pmem_flush()` scale well across all data sizes and thread counts. Among all evaluated methods, the best performance is achieved by sequential flushing in data sizes of 64 B. Consequently, developers should make use of this knowledge when designing and developing data structures and algorithms with `libpmem`. In addition to the data size, developers should also distinguish between the different persisting methods of the PMDK. As seen by our results, the wrong method can have a significant impact on the overall performance of the applications. Therefore, we endorse the recommendation of the PMDK to primarily use `pmem_persist()` on persistent memory. Alternatively, `pmem_deep_persist()` can be used for critical values that require the highest reliability. However, this function should be handled with caution since it involves a much larger overhead than `pmem_persist()`.

`libpmem2`

Background `libpmem2` is the successor of `libpmem`. According to the developers, it offers "a more universal and platform-agnostic interface" [4]. Like `libpmem`, `libpmem2` provides the same low-level primitives to access the persistent memory directly. However, it enhances the existing mechanism with additional concepts.

One of the new concepts is granularity. With the concept of granularity, developers can now distinguish between different levels of storage performance in terms of the *power-fail protected domain*. Traditionally, block devices (e.g. SSD and HDD) preserve data by flushing entire pages to the medium [7]. In `libpmem2`, this behavior can be achieved using `PMEM2_GRANULARITY_PAGE`. For persistent memory, `libpmem2` offers two granularity types: If only the memory controller is covered by the *power-fail protected domain*, cache lines have to be flushed to persistence by using CPU instructions such as `CLWB`, `CLFLUSH`, and `CLFLUSHOPT`. In `libpmem2`, this granularity type is called `PMEM2_GRANULARITY_CACHE_LINE`. In systems where both the memory controller and the CPU caches are covered by the *power-fail protected domain*, developers can use the granularity `PMEM2_GRANULARITY_BYTE`. On such systems, flushing of the CPU caches is no longer required as the CPU caches are already considered persistent.

The concept of granularity enables developers to be more specific about required storage patterns by declaring their minimum supported granularity [30]. For example, a database storage engine may require the granularity `PMEM2_GRANULARITY_CACHE_LINE`. Depending on the platform's capabilities, `libpmem2` creates the file mappings with granularity lower or equal to the requested one. For our database engine, this means that files are mapped with either `PMEM2_GRANULARITY_BYTE` or `PMEM2_GRANULARITY_CACHE_LINE` but not with `PMEM2_GRANULARITY_PAGE`.

In our benchmarks, we compare the performance differences between all three granularity types for persisting data. Moreover, we examine if the second iteration of the low-level API provides notable performance benefits compared to the initial version. Similar to the previous section, we investigate the single-threaded performance for data sizes between 8 B and 8 KiB. To assess the multithreaded scalability, we select the three aforementioned data sizes: 64 B (cache line size), 256 B (size of the DCPMM's write-combining buffer), and 4 KiB (default Linux page size). All experiments are computed with 100,000 operations per thread and a prior warmup.

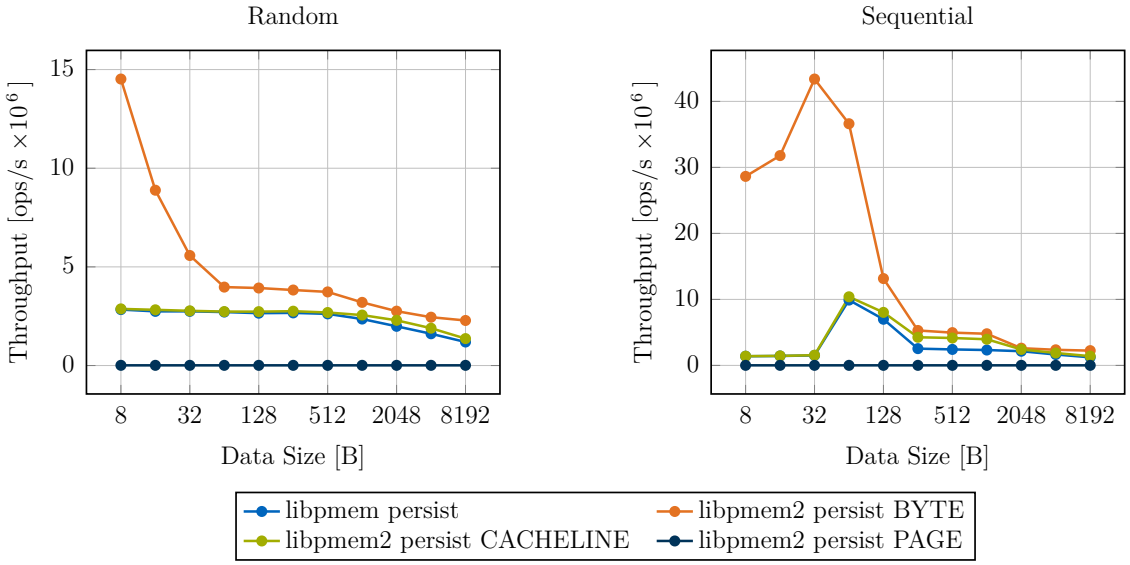


Figure 4.4: *PMDK benchmark: Performance comparison of persist operations in `libpmem` and `libpmem2` for varying data sizes*

Evaluation As depicted in Figure 4.4, each granularity type has distinctive performance characteristics. Generally, `PMEM2_GRANULARITY_BYTE` offers the highest performance among all tested mechanisms. Particularly for data sizes smaller than 64 B, we observe over 3× better throughput for random data and 5× better throughput for sequential data compared to the next best granularity. For larger-sized data, the performance is still superior, although not as dominant. The reason for the overall better throughput is the omission of the cache line flushing since `PMEM2_GRANULARITY_BYTE` presumes that the CPU caches are already covered by the *power-fail protected domain*. As for the comparison with `libpmem`, we state that the `libpmem2` granularity `PMEM2_GRANULARITY_CACHE_LINE` is the closest in terms of performance and characteristics. Both methods remain at a constant performance level for data sizes below 256 B and then start gradually declining. Regarding the performance of `PMEM2_GRANULARITY_PAGE`, we observe a significant performance disadvantage across all data sizes. For sizes larger than 4 KiB, the other mechanisms converge towards the same performance level. This performance progression is expected as `PMEM2_GRANULARITY_PAGE` persists data in page granularity. As a reminder, Linux has a default page size of 4 KiB.

Consequently, PMEM2_GRANULARITY_PAGE is best suited for applications that operate on large logical pages and do not require finer-grained persistence.

For the multithreaded performance, depicted in Figure 4.5, the trends between the granularities remain the same, with PMEM2_GRANULARITY_BYTE delivering the best performance followed by PMEM2_GRANULARITY_CACHE_LINE and PMEM2_GRANULARITY_PAGE. Taking a closer look at the comparison between libpmem and libpmem2, we see that libpmem2 with PMEM2_GRANULARITY_CACHE_LINE offers a slightly higher throughput than its predecessor libpmem. This improvement is particularly apparent for 4 KiB.

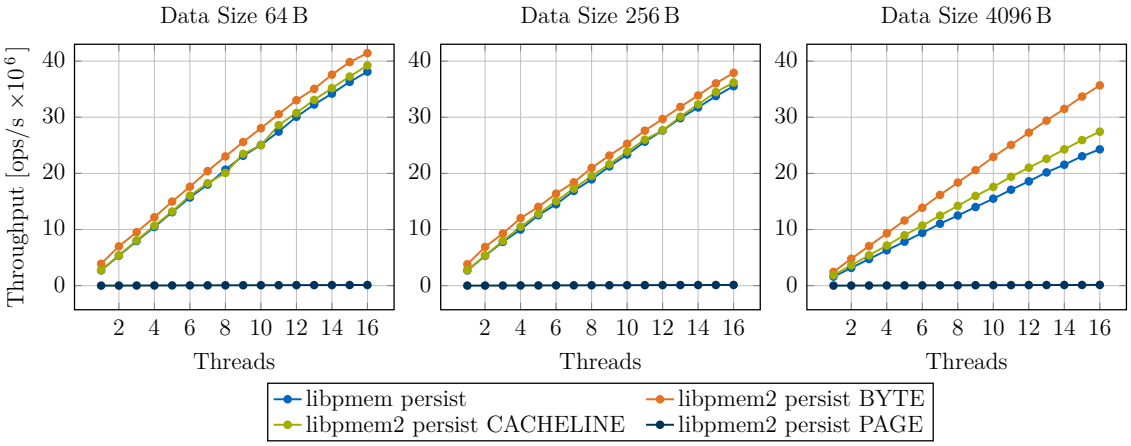


Figure 4.5: *PMDK benchmark: Comparison of the multithreaded performance of persist operations for libpmem and libpmem2*

Summary As illustrated by our plots, libpmem2 is not only the successor of libpmem but also an evolution in terms of performance. With the introduction of the granularity concept, libpmem2 provides developers more freedom in their application design and implementation. Depending on the use case and the platform, developers can now leverage different degrees of persistence inside their application. With PMEM2_GRANULARITY_BYTE, libpmem2 is also prepared for the next generation of CPUs which might expand their *power-fail protected domain* to the CPU caches.

Regarding its performance, libpmem2 matches or even outperforms its predecessor libpmem in almost every area. For PMDK's higher-level libraries, which are currently based on libpmem, switching to libpmem2 can therefore result in higher overall performance. But also for other persistent memory applications currently employing libpmem, a migration to libpmem2 can be worthwhile to improve their performance.

With the current technology, PMEM2_GRANULARITY_CACHE_LINE provides the best balance between persistence and performance for most applications. Fortunately, due to the downward compatibility, applications implementing PMEM2_GRANULARITY_CACHE_LINE can leverage the additional performance of PMEM2_GRANULARITY_BYTE without any modifications once the *power-fail protected domain* extends to the CPU caches.

pmem_memcpy()

Background Copying memory is one of the fundamental operations frequently used by developers. Instead of copying values to persistent memory using `memcpy()` with a subsequent `pmem_persist()`, developers can leverage `libpmem`'s PM-optimized version, called `pmem_memcpy()` [62]. `pmem_memcpy()` provides the same functionality as its namesake `memcpy()` but additionally ensures the data is persisted before returning. Another benefit: where applicable, `pmem_memcpy()` utilizes non-temporal stores to bypass the CPU caches, resulting in superior performance compared to `memcpy()` with a subsequent `pmem_persist()`. One important detail to note here: `libpmem` ensures for its memory functions 8 B-atomicity. More specifically, when the destination buffer address and length are 8 B aligned, `libpmem` guarantees that all stores are performed using at least 8 B store instructions. As a result, in the event of a crash, each 8-byte location has either the new or the old memory content, but never a mix of the two [29]. Rudoff [63] indicates that this behavior might improve in upcoming CPUs with the introduction of the `MOVDIR64B` instruction, which enables 64 B atomic stores.

In order to give developers more freedom, `libpmem` features an additional function called `pmem_memcpy_nodrain()` [62]. This function is similar to the earlier described function `pmem_memcpy()` but skips the final step of draining the hardware buffers. Developers can therefore optimize their applications by performing multiple copy operations with a single ensuing call of `pmem_drain()`. However, as previously reported, the `pmem_drain()` operation on Intel platforms only serves as `SFENCE` instruction because the *power-fail protected domain* already covers the CPU hardware buffers.

In our experiments, we examine the performance differences between the various approaches of copying data to persistent memory. Furthermore, we quantify the overhead of persisting data after it has been copied to persistent memory using `memcpy()`. As with previous experiments, a single thread is used to perform various operations on data sizes reaching from 8 B to 8 KiB. The selection of the data chunks is performed both in sequential and random order. For the second experiment, all operations are computed with a varying number of threads for data chunks in random order. As before, we perform 100,000 operations per thread and a warmup beforehand to prevent page allocation performance from being measured. The used data sizes for multithreading remain the same as in previous experiments: 64 B (cache line size), 256 B (size of the DCPMM's write-combining buffer), and 4 KiB (default Linux page size). Please note that in this case, the throughput is measured in GiB/s instead of ops/s to take into account the data size of each memory operation.

Evaluation Figure 4.6 depicts the progression of the different approaches over a range of data sizes. We observe that the `memcpy()` operations perform better than `libpmem` namesakes for sizes smaller than 256 B. For all data sizes onwards, the order is inverted, with the `libpmem` functions leading. As expected, omitting the order barriers increases the performance of `memcpy()` and `pmem_memcpy()`.

Looking at Figure 4.6 in more detail, we identify 256 B as a special point of interest. Up to 128 B, `memcpy()` offers the highest throughput among all operations. From 256 B

onwards, the `libpmem` variant has a superior performance. Especially for the `libpmem` methods, we notice a steep increase from 128 B to 256 B. These characteristics coincide with previously reported findings about the performance of the DCPMM [12, 15, 17]. As stated in Section 2.1, the DCPMM has an internal write-combining buffer of 256 B. Consequently, data accesses of 256 B achieve the best throughput. Another explanation for the higher throughput is the non-temporal stores. As highlighted previously, `pmem_memcpy()`, unlike `memcpy()`, uses non-temporal stores to optimize its efficiency. Particularly for large-sized data, the ability to bypass the CPU caches can result in improved performance.

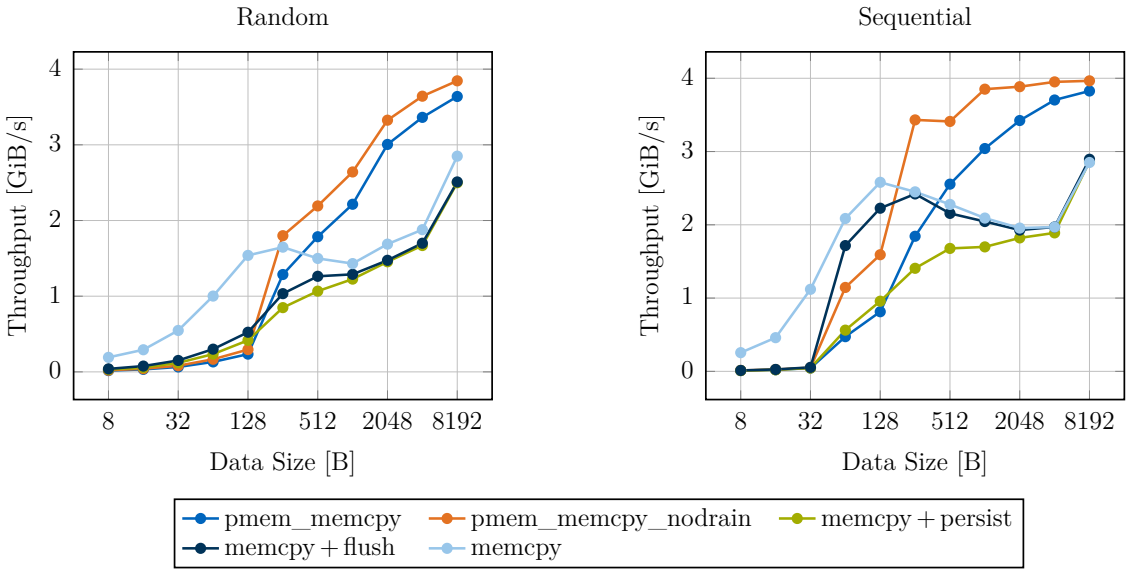


Figure 4.6: *PMDK benchmark: Performance of memcpy operations for different data sizes*

As for the multithreaded performance depicted in Figure 4.7, the presented persistent copying methods provide linear scaling in most cases. Only for large data sizes, an increase in threads may result in a throughput degradation. But there are further interesting aspects to highlight: First, with the increasing number of threads, the use of ordering barriers for the store operations becomes more important. For example, for 4 KiB and 16 threads, we see that the throughput of `pmem_memcpy()` is 1.3× higher than `pmem_memcpy_nodrain()`. We assume that the looser ordering of the `pmem_memcpy_nodrain()` operation creates temporary bottlenecks as the data is not frequently evicted to persistent memory. Second, for small data sizes, using `memcpy()` with a subsequent flushing operation leads to a higher throughput compared to equivalent methods of `libpmem`. Nevertheless, this performance advantage is only marginal and probably negligible in most applications. Last and most interestingly, the multithreaded performance of `memcpy()` only increases up to a certain limit. With growing data sizes, it requires fewer threads to reach this upper bound. For example, for 256 B, it requires five threads to reach the limit, but for 4 KiB, the limit can be reached with a second thread. Surprisingly, this limit does not seem to apply to `memcpy()` followed by a flushing operation, as it outperforms the plain `memcpy()`. A similar behavior

was documented by Kalia *et al.* [26]. Anuj Kalia's explanation for our problem is that "data evicts in near-random order [when using `memcpy()`], whereas [persistent memory] is optimized for sequential writes. `pmem_memcpy()` ensures that the destination buffer cache lines evict to [persistent memory] in sequential order (by issuing flushes)" [64].

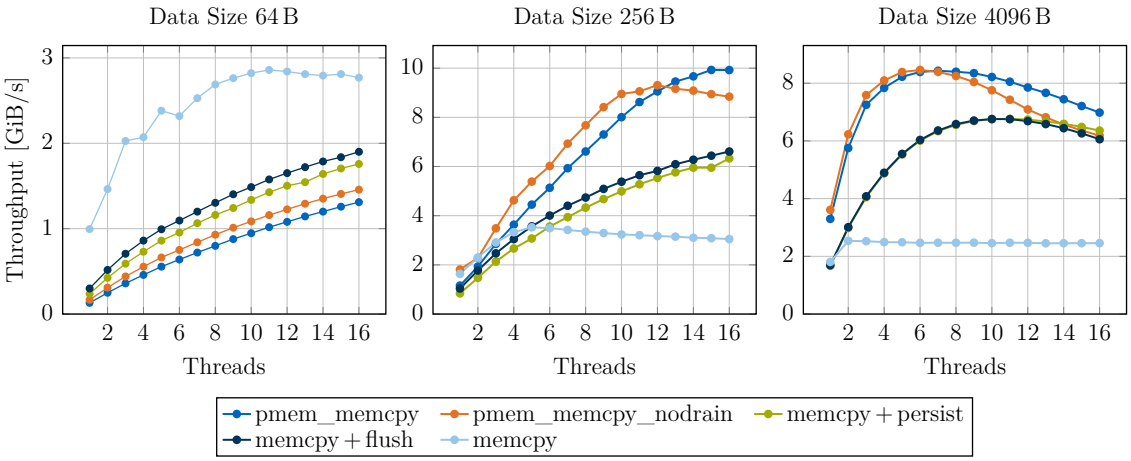


Figure 4.7: *PMDK benchmark: Multithreaded performance of memcpy operations*

Summary Our plots illustrated that various parameters affect the performance of copying to persistent memory. Generally, developers should avoid `memcpy()` and use `pmem_memcpy()` instead. In particular, for data sizes larger than 256 B, the `libpmem` namesakes provide superior performance. To further optimize the efficiency, developers should consider using `pmem_memcpy_nodrain()` with a manual subsequent `pmem_drain()`. However, they need to be aware of the weaker persistence guarantees that are associated with this approach. In terms of multithreading progression, we conclude that it scales linearly for the most part. Only for large data sizes in combination with high thread counts, there seems to be performance degradation. As a general rule: the smaller the data size, the more threads can concurrently copy to persistent memory.

`pmem_memset()`

Background In addition to the previously mentioned `pmem_memcpy()`, `libpmem` also offers a namesake for `memset()`, called `pmem_memset()` [62]. It provides the same basic functionality as `memset()` but with the guarantee of persistence on returning. Compared to `pmem_memcpy()`, `pmem_memset()` sets the contents of a memory range to a specific value instead of copying them from one memory location to another. The behavior of `pmem_memset()` can be represented as `memset()` with a following `pmem_persist()` complemented with the ability to use non-temporal stores. In this regard, `pmem_memset()` shares a lot of functionalities with its fellow memory operations `pmem_memcpy()` and `pmem_memmove()`. One important aspect we want to re-emphasize here is the 8 B-atomicity. As long as the destination buffer address and length are 8 B aligned, `pmem_memset()` uses

at least 8 B atomic store instructions. Furthermore, `pmem_memset()` also offers a supplementary function without subsequent hardware buffer draining, named `pmem_memset_nodrain()`. This function enables developers to further optimize their algorithms and data structure.

For performance evaluation of `pmem_memset()`, we apply a similar benchmark pattern as for `pmem_memcpy()`. First, examining the performance across a range of data sizes, followed by an investigation of multithreaded performance. In order to assess the single-threaded performance, we execute the experiments for data sizes between 8 B and 8 KiB. The memory destination is selected both in sequential and in random order. As for the multithreaded performance, we select the memory destination randomly and vary the number of threads. The inspected data sizes are 64 B (cache line size), 256 B (size of the DCPMM's write-combining buffer), and 4 KiB (default Linux page size). Throughout all experiments, each thread executes 100,000 operations. It is worth noting that the throughput of these experiments is also measured in GiB/s as opposed to ops/s.

Evaluation Overall, we recognize similar characteristics as for `pmem_memcpy()`. Below 256 B, the `memset()` variants achieve higher throughput. Starting with 256 B, the `libpmem` namesakes pass the `memset()` versions for both random and sequential order. The same holds true for multithreaded performance with `pmem_memcpy()` providing better throughput for the data sizes 256 B and 4 KiB, but falls short of `memcpy()` for 64 B.

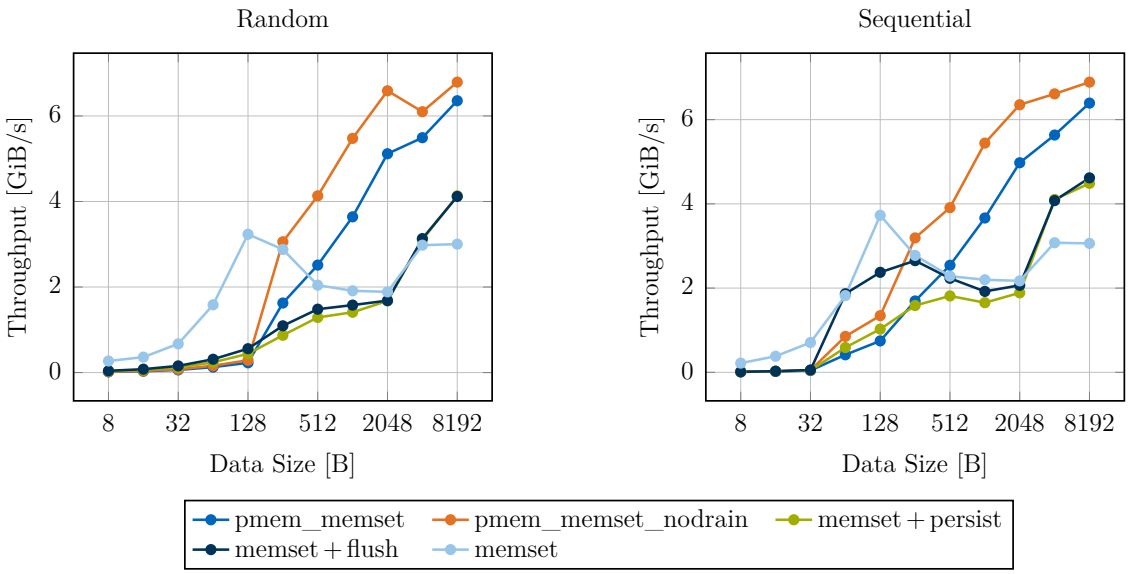


Figure 4.8: *PMDK benchmark: Performance of memset operations for different Data Sizes*

Taking a closer look at the throughput progression over various data sizes in Figure 4.8, we identify two distinct parts. The first part contains all data sizes between 8 B and 128 B, and the second the remaining data sizes from 256 B to 8 KiB. For the first part, we observe that all operations involving `memset()` outperform their `pmem_memset()` namesakes. The second part is inverted, with `pmem_memset()` operations coming out on top.

This behavior is particularly apparent for the plain `memset()` operation, which steadily increases up to 128 B and then abruptly decreases. With this progression, the general performance characteristics are comparable to those of `pmem_memcpy()`. As pointed out earlier, this behavior is most likely related to the DCPMM's internal write-combining buffer in the size of 256 B. Another important aspect that should not be overlooked in this context is the use of non-temporal stores in `libpmem`. In Section 4.2.1, we have seen that flushing larger data sizes can negatively affect the performance. In this case, bypassing the CPU cache can significantly increase the throughput.

To further improve the performance, one can omit the ordering barriers by using `pmem_memset_nodrain()`. As depicted by the plots, using `pmem_memset_nodrain()` can especially accelerate applications which are dealing with data sizes between 256 B and 4 KiB. However, as the number of threads and data size increases, the advantage fades and eventually becomes a disadvantage. We observed a similar behavior for `pmem_memcpy()`. Consequently, it is only recommended to skip draining when dealing with small data sizes and a low number of threads.

Regarding the progression of the multithreaded throughput depicted in Figure 4.9, there is another aspect we want to highlight. Analogous to `pmem_memcpy()`, we observe an upper bound for the throughput. With increasing data size, it takes fewer threads to reach this limit. From this observation, we deduce that the bandwidth of the DCPMM becomes a limiting factor for multithreaded applications. This finding is in line with multiple works indicating that a small number of threads is sufficient to saturate the write bandwidth of the DCPMM [12, 15, 17].

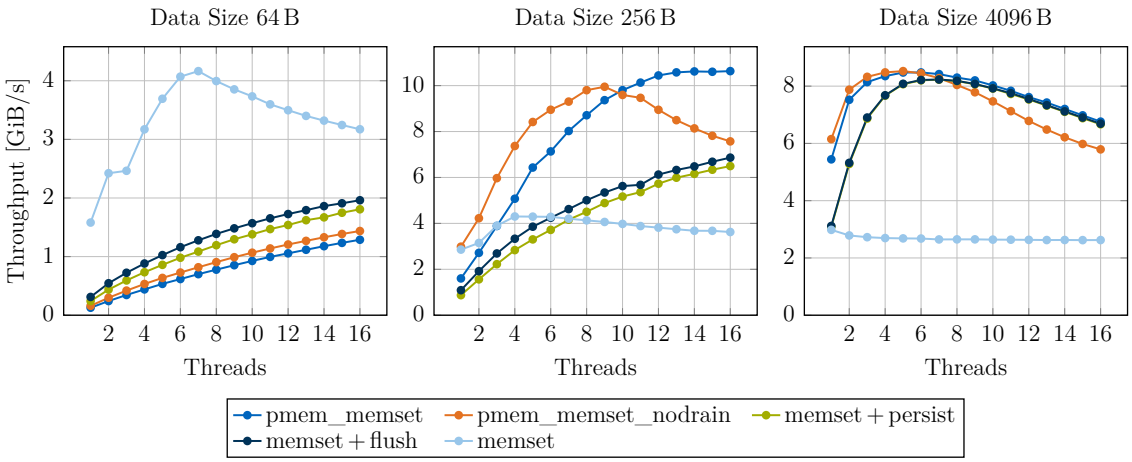


Figure 4.9: *PMDK benchmark: Multithreaded Performance of memset operations*

Summary Together, the plots indicate that `pmem_memset()` provides the best performance among all evaluated operations. Especially for larger data sizes, `libpmem`'s operations can outperform the `memset()`-based methods. As visualized by our plots, data sizes smaller than 256 B should generally be avoided. To get the best balance between granularity

and throughput, future algorithms and data structures should be optimized for the data size of 256 B. For performance sensitive applications, developers should consider using `pmem_memset_nodrain()` for larger memory ranges followed by a manual `pmem_drain()`. Depending on the data size, this can significantly increase the throughput but at the same time weakens the persistence as it cannot be guaranteed that the data reaches the media in the anticipated order. As far as multithreading performance is concerned, we see that the performance scales linearly up to a certain limit before throttling. Therefore, applications with larger data sizes should limit the number of concurrent operations to avoid performance degradation. It should be noted that some of the peculiarities seen throughout this section appear to be related to the DCPMM's performance properties.

4.2.2 Transactions (`libpmemobj`)

Background `libpmemobj` [31] was designed to facilitate persistent memory programming for developers by providing a more convenient API with additional features. It builds upon the previously explained `libpmem` library to implement a transactional object store based on memory-mapped files. In addition to the transactional object store, `libpmemobj` also provides higher-level functionality like memory management, locking, and application recovery [31].

In `libpmemobj`, memory-mapped files are represented as pools. The library hides the complexity and details of directly mapping the files and synchronizing data. Instead, it provides a more intuitive API for creating and managing these pools. Due to the address space layout randomization (ASLR) feature in many operating systems, the memory location of the pool may vary between system and application restarts. To address this issue, `libpmemobj` attaches metadata to each pool, allowing for identification across restarts [31]. One important value in this metadata is the offset to the root object. The pool's root object serves as an entry point for finding all other objects. To locate the root object itself, an application uses the *pool object pointer* (POP), which is created at each program execution and then stored in volatile memory [3, p.87]. The conjunction of POP and root object allows applications to access any object inside the pool. The developers of `libpmem`, however, realized that accessing objects via untyped direct pointers is very error-prone and hard to debug. To address this issue, they introduced typed persistent pointers, also called *typed object identifiers* (TOIDs). These typed persistent pointers are based on named unions and enable static type enforcement for persistent pointers through macros, catching potential errors at compile time [65].

Based on those described features, `libpmemobj` provides a transactional API. The basic idea of transactions is the consolidation of multiple operations into one single operation. Imagine making a bank transfer from one account to another: The money should be debited from one account and credited to the other simultaneously. Both operations should act as a single atomic operation, either the money was transferred or not. A common solution to guarantee this atomicity in transactions is logging [24, 25]. In this approach, all planned changes are persisted in a log before they are executed. In the event of a system failure, the log is used to recreate a consistent state by rolling back the changes or reapplying them. For its transactions, `libpmemobj` uses a hybrid undo/redo logging technique [66].

Before a variable is written, it has to be added to the transaction. In `libpmemobj`, this can be achieved by calling `pmemobj_tx_add_range()`. This function creates a *snapshot* of the given memory block and saves it to the undo log. In case of a failure or abort, the changes to this memory block are reverted. To function probably, the memory block has to be contained within the transaction's pool [31].

Transactions in `libpmemobj` consist of multiple stages, which are defined via macros. The mandatory macros are `TX_BEGIN` and `TX_END`, which set the start and the end of the transaction. In addition, there are three optional stages: `TX_ONCOMMIT`, `TX_ONABORT`, and `TX_FINALLY`. These stages can be used to execute additional code depending on the result of the transaction. To create more complex algorithms, transactions can also be nested. Note that if one nested transaction aborts, the entire transaction aborts as well [31].

The experiments in this section aim to quantify and examine the performance of the `pmemobj_tx_add_range()`. As explained earlier, this function is frequently used in transactions to ensure recoverability. Due to this importance, we investigate its performance across various data sizes and threads. Similar to previous sections, we measure the single-threaded performance for data sizes between 8 B and 8 KiB. The multithreaded performance is evaluated with the previously used data sizes: 64 B (cache line size), 256 B (size of the DCPMM's write-combining buffer), and 4 KiB (default Linux page size). Each experiment is conducted with 100,000 operations per thread.

Evaluation Figure 4.10 depicts the results of our experiments. In the left plot, we see the throughput progression over varying data sizes. For data sizes up to 64 B, the throughput remains on a consistent level. From this observation, we infer that logging memory ranges smaller than 64 B does not yield any additional performance. Thus, applications cannot exploit finer-grained logging to improve their efficiency. For data sizes larger than 64 B, we identify a sudden drop in throughput, almost halving the performance from 64 B to 128 B. Because logging is an essential aspect of transactions, applications that frequently modify data should avoid using data sizes larger than 64 B.

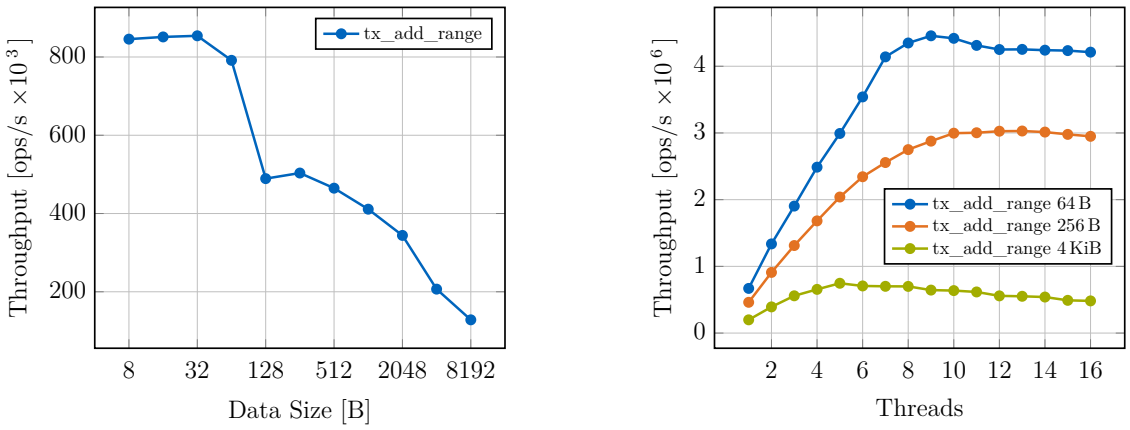


Figure 4.10: *PMDK benchmark: Performance of* `tx_add_range` *for different sized data (Left) and threads (Right)*

When scaling with multiple threads, we observe an upward sloping asymptotical curve for all data sizes. Depending on these sizes, the upper bound is reached with a smaller number of threads. We have already noticed a similar behavior earlier with `pmem_memcpy()` and `pmem_memset()`. Hence, we suspect that this characteristic is caused by the DCPMM's limited write bandwidth too. From this throughput progression, we can infer that multi-threaded applications should avoid the combination of larger data sizes and higher thread counts not to saturate the bandwidth of the DCPMM.

Summary Based on the results, we deduce that `pmemobj_tx_add_range()` is better suited for smaller data sizes. We were able to identify that 64 B has the optimal balance between performance and size. Data sizes smaller than 64 B offer the same performance level, whereas larger data sizes are considerably slower. In terms of multithreaded performance, we notice that even a small number of threads is sufficient to reach the maximum throughput. Therefore, applications should only use up to 8 concurrent threads when writing data to the log.

Taking everything into account, our general recommendation for developers is to use smaller data sizes and log the value right before modifying it. Following this approach, the logging operations are better distributed, ensuring a more even utilization of the available bandwidth.

4.2.3 Persistent Memory Allocation/Deallocation (`libpmemobj`)

Background As explained earlier, `libpmemobj` provides higher-level functionalities for persistent memory. One functionality we want to detail in this section is the dynamic memory management of persistent memory. The concept of dynamic memory management is an essential part of C and other programming languages. Dynamic memory management allows applications to request memory at run-time and release it when it is no longer needed. In volatile memory, those memory requests are satisfied by allocating unused memory from a large memory pool known as *Heap*.

With persistent memory, there are some additional challenges in allocating and deallocating memory. An important term in this context is a *persistent leak*. A persistent leak occurs if a persistent memory region is marked as allocated, but no longer in use. Unlike volatile memory leaks, persistent leaks cannot be resolved by a system reboot. As a result, the amount of available persistent memory continuously shrinks, which can negatively impact the performance of other applications. One source of persistent leaks may be that the allocation operation was interrupted after marking the region as allocated, but before writing the corresponding data. In this case, the memory region cannot be used even though it is still marked as allocated.

The opposite problem is overwriting allocated memory regions. This issue occurs if the interruption happens after the data was written, but before the region was marked as allocated. When the application restarts, the allocator assumes that this region is unused and may overwrite it.

Due to these problems, a persistent memory allocator has to atomically allocate memory, preserve allocations and locate objects across application and system restart.

To fulfill those requirements, `libpmemobj` includes its own PM-optimized memory allocator inspired by the well-known volatile Hoard allocator [67, 68]. It supports the known dynamic memory management interfaces for allocating, resizing, and freeing objects.

All memory operations of the `libpmemobj` can be divided into two steps [3, p.324]: The first step is the reservation step. For allocation, this step involves retrieving a memory block, marking it as reserved, and initializing the object's content. The second step is the publication of the executed reservation. This bifurcation allows multiple reservations to be combined and published together.

To leverage this functionality, `libpmemobj` offers two separate APIs for persistent memory management: transactional allocations and fail-safe atomic allocations [31].

Starting with transactional allocations, this is the approach most similar to the standard POSIX methods. The main transactional operations for dynamic memory management are `pmemobj_tx_alloc()` and `pmemobj_tx_free()`. To ensure consistency of the memory operations, the transactional allocator uses a redo-log to track all executed operations. Within the transaction, the reservation step is performed for each memory operation. At the time of the transaction commit, the reservation actions are published, and the redo-log is created. If the transaction is aborted before its commit, all reservation actions are canceled and discarded.

The second available allocation API is the fail-safe atomic allocation. Since these operations do not use logging, they tend to have lower overhead. As the name implies, all methods of this API are atomic with respect to other threads and possible power failures. To ensure this atomicity, the functions reserve the object in a temporary state, call the object's constructor, and mark the allocation as persistent, all in a single atomic action. The available methods are in the format of `pmemobj_alloc()` and `pmemobj_free()`.

An important detail to note is that the actual size of the persistent allocation differs from the request size by at least 64 B due to internal padding and object metadata. For this reason, the PMDK's developers state that "making allocations of a size less than 64 bytes is extremely inefficient and discouraged" [69].

To quantify the performance difference between the two APIs for dynamic memory management, we compare them using various object sizes and number of threads. Our testing focuses on the two core functions: allocation and deallocation of persistent memory. For object sizes, similar to the data sizes in previous experiments, we evaluate the performance between 8 B and 8 KiB. Concerning the multithreaded scalability, we select the following object sizes: 64 B (cache line size and, according to the developers, the smallest efficient object size), 256 B (size of the DCPMM's write-combining buffer), and 4 KiB (default Linux page size). Each thread computes 100,000 operations, same as in the previous experiments.

Evaluation We can deduce some general characteristics from the presented plots: First, deallocations are faster than allocations for both APIs. Second, the transactional memory operations introduce a measurable overhead compared to the atomic operations. However, this overhead becomes negligible with increasing object size. Third, as pointed out by the PMDK's developers, object sizes smaller than 64 B do not provide any additional performance. These observations also apply to multithreaded performance.

Figure 4.11 depicts the performance progression across various object sizes. Up to 64 B, the performance of each function remains on a steady level, with the atomic operations being around 20% faster than their transactional counterparts. From 64 B to 1024 B, the throughput of all operations decreases linearly with a small outlier at 512 B. In the same range, the performance of the allocation and deallocation operations converges for both APIs. Starting from 1024 B onwards, the throughput of both deallocation operations remains on a constant level, whereas both allocation operations continue their linear decrease, resulting in a notable performance difference.

From this throughput progression, we deduce that the aforementioned statement written by the PMDK's developers is justified. Applications should be designed and implemented in such a way that allocations of less than 64 B are avoided. For allocations greater than 64 B, developers can expect a linear decrease in performance.

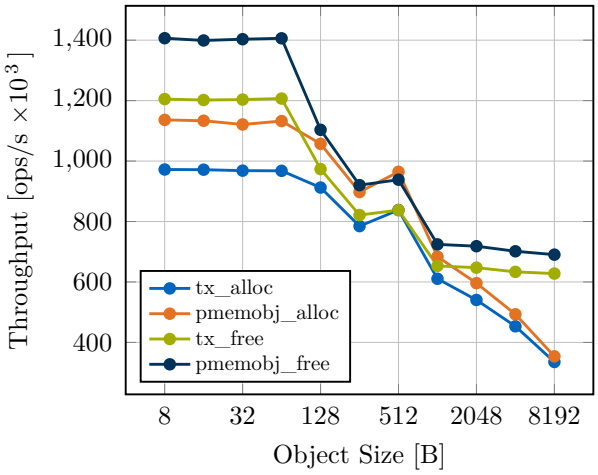


Figure 4.11: *PMDK benchmark: Performance of* `libpmemobj`*'s allocation and deallocation mechanisms for differently sized objects*

Proceeding with the multithreaded performance visualized in Figure 4.12, we notice a linear throughput increase for both deallocation operations across all examined object sizes. In contrast, the allocation operations only scale linearly up to a certain upper bound. This characteristic is particularly apparent for 4 KiB. It only requires four threads to reach the throughput limit, with throughput dropping further after eight threads. This phenomenon is already known from volatile memory allocators and is caused by lock contention. To prevent concurrent modifications of allocation data structures, allocators use locks to coordinate the access and ensure consistency [70]. The longer the allocation takes, the longer the locks are acquired, blocking the allocations of other threads. As deduced from our plots, this is especially problematic for larger allocations.

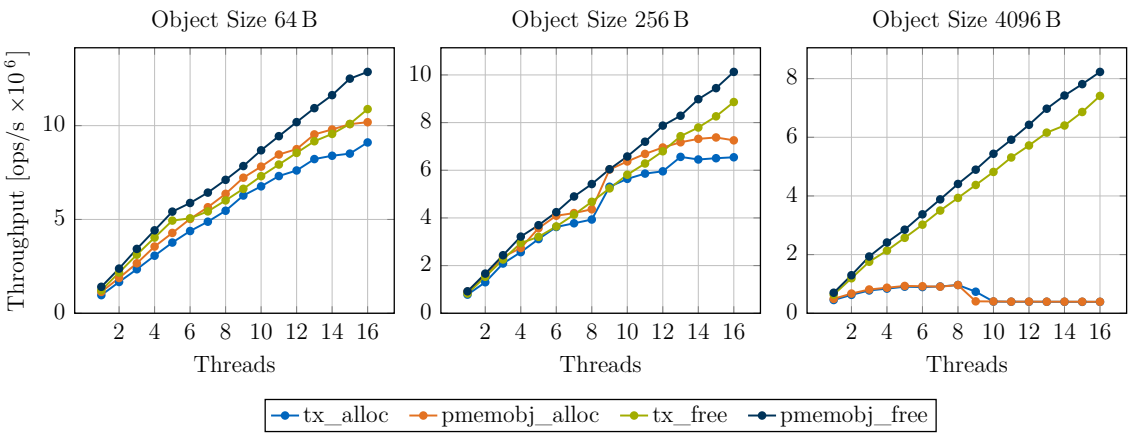


Figure 4.12: *PMDK benchmark: Multithreaded performance of transactional and atomic deallocation and allocation*

Summary Our experiments examined the performance characteristics of the `libpmemobj`'s dynamic memory management APIs. As demonstrated by our plots and stated by the PMDK's developers, allocation of objects smaller than 64 B should generally be avoided [69]. For objects larger than 64 B, we observed a mostly linear performance decrease. Across all evaluated object sizes, the atomic operations outperformed their transactional counterparts. Performance-oriented developers should therefore utilize the atomic operations for critical sections of their applications. For all other developers, we recommend the transactional API as it offers a similar functionality and interface to the POSIX-API with reasonable overhead. The same recommendation also applies to multithreaded applications. Here, it is important to add that the combination of large-sized objects and a high number of threads can result in reduced throughput. Consequently, developers should avoid large concurrent allocations or utilize multithreaded-optimized persistent memory allocators [71, 72].

4.2.4 Persistent Locks (`libpmemobj`)

Background Applications often require multiple concurrent threads to leverage the capabilities of modern multi-core machines. Sometimes, these threads have to access the same data simultaneously. However, concurrent unrestricted access to shared data also entails risks. To illustrate the problem, we refer to the previous banking example of transferring money from one account to another. Consider multiple threads crediting received money to the account. In this scenario, both threads simultaneously read the current account balance and start their process. When finished, both threads write back their calculated values. During this process, the later ending thread might quietly overwrite the value of the first. As a result, only one calculation is included in the total balance.

As the example indicates, the access of multiple threads to shared data has to be synchronized in order to not depend on accidents of timing. One synchronization technique for coordinating the access to the shared data is *locking*. The concept of locking is that before a thread can access shared data, it must acquire a lock. If the thread attempts to

acquire a lock already owned by another thread, it has to wait until the lock is released again. Despite being an established approach, using locks on persistent memory poses new challenges.

An important term in this context is a *persistent deadlock* [3]. A persistent deadlock happens when an application crashes while holding a lock. When the application does not release the lock after recovery, other threads might wait indefinitely to acquire the lock. In order to avoid persistent deadlocks, `libpmemobj` provides POSIX-like synchronization primitives for persistent memory [31]. These synchronization primitives prevent persistent deadlock by reinitializing themselves each time the persistent pool is opened. Consequently, all synchronizations primitives are considered unlocked after the pool is opened, regardless of their previous state. `libpmemobj`'s synchronization primitives consist of mutexes (`PMEMmutex`), read-write locks (`PMEMrwlock`), and condition variables (`PMEMcond`) [31].

For the following benchmarks, we focus on mutexes, read-locks, and write-locks. In order to understand the results more clearly, we briefly explain the functional difference between read-locks and write-locks. With the acquisition of a read-lock, the thread indicates that it only wants to read the data. Thus, other threads are able to access the data for reading as well. On the other hand, threads can acquire write-locks when they want to modify data. Consequently, no other threads are able to read or write the data.

For performance evaluation of these synchronization primitives, we measure the single-threaded throughput from 10 to 500 operations per thread. The thread consecutively locks and unlocks a single object to represent the raw performance of the operation. The goal of these experiments is to quantify the overhead of `libpmemobj`'s mechanisms compared to the known volatile POSIX ones.

Evaluation Starting with the mutexes, depicted in the left plot of Figure 4.13, both variants have a similar overall performance progression. The throughput increases sharply up to 100 operations per thread. After 100 operations per thread, the performance only increases gradually, remaining in a similar order of magnitude. Comparing POSIX and `libpmemobj` mutexes, we observe that the POSIX mutex is consistently faster than the `libpmemobj` equivalent. Across all data points, the overhead of the `PMEMmutex` is around 12%.

The general characteristics of the read-write locks are similar to those of mutexes, with an upward sloping asymptotical throughput curve. Compared to the mutexes' performance, the average throughput of read-write locks is slightly slower. The upper bound for read-write locks is around 13% lower than for mutexes. Looking at the two variants of locks, the performance difference between read-locks and write-locks is around 25% for both POSIX and `libpmemobj`. The overhead of the `PMEMrwlock`, compared to the POSIX's mechanisms, is only approximately 9%.

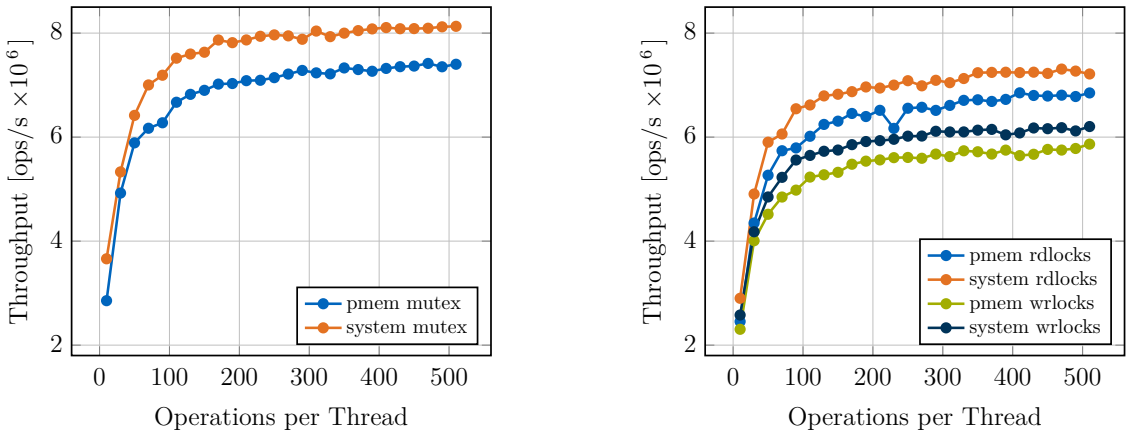


Figure 4.13: *PMDK benchmark: Performance comparison between POSIX and* `libpmemobj` *mutexes/locks*

Summary From the findings reported in Figure 4.13, we deduce that the `libpmemobj`'s persistent synchronization mechanisms provide similar performance characteristics as the known POSIX mechanisms. For both variants it is generally recommended to differentiate between the available mechanism to improve performance, e.g. by using read-locks instead of mutexes in applicable data structures and algorithms. Overall, the additional overhead of the `libpmemobj`'s primitives, compared to the native POSIX variants, is only around 9% to 12%.

In addition to `libpmemobj`'s synchronization primitives, there are other recoverable mutex algorithms for persistent memory, namely the Golab and Hendler (GH) algorithm and Jayanti, Jayanti and Joshi (JJJ) algorithm. Xiao *et al.* implemented those algorithms using the PMDK and assessed their performance on persistent memory. In their experiments, they discover that the performance properties of Intel's DCPMM can have a significant impact on the performance of those algorithms [73].

Besides the mentioned synchronization primitives, developers should also consider lock-free algorithms and data structures for their multithreaded applications. Lock-free algorithms and structures use atomic transitions between consistent states to ensure isolation between different threads. In terms of persistent memory, lock-free algorithms provide an appealing approach because they efficiently circumvent the issue of persistent deadlocks. To accomplish this, lock-free algorithms have to be carefully designed to ensure that the data is persisted in the correct order. Otherwise, modifications of subsequent threads are persisted too early, resulting in inconsistencies like in the aforementioned banking example.

4.3 Realworld Benchmarks

After having evaluated the performance characteristics of the PMDK's core components, this section focuses on the evaluation of the `pmemkv`. `pmemkv` is PMDK's persistent memory optimized key-value store. It provides an easy-to-use and familiar interface for programmers who want to take advantage of persistent memory [3]. To establish a baseline and scrutinize the distinct performance properties of `pmemkv`, we first analyze and assess the performance using the microbenchmark `pmemkv-bench`. Afterwards, we validate our findings with the YCSB and compare `pmemkv` with other PMDK-enabled modern database systems such as MongoDB, PostgreSQL, and Redis.

4.3.1 PMDK Key-Value Store (`pmemkv`)

Background The PMDK contains several libraries across various areas of application. The main goal of these libraries is to allow developers and applications to leverage the novel characteristics of persistent memory. Therefore, the libraries should be flexible and easy to use at the same time. Two requirements that are inherently contradictory: more flexibility usually implies more complexity. A data structure attempting to deliver both is a key-value store. There are several key-value stores available, but the most popular ones are Redis, LevelDB, and RocksDB [74]. Although these systems differ in their features, they all share the same basic interface, consisting of `put`, `get`, `remove`, and `exists` operations.

With `pmemkv`, PMDK offers its own PM-optimized key-value store with the aforementioned basic interface, allowing developers to easily migrate and leverage persistent memory. The native API of `pmemkv` is C with additional headers for C++. However, there are multiple language bindings available, including Java [33], JavaScript [34], Python [35], and Ruby [36]. To grant further degrees of flexibility and functionality, `pmemkv` provides multiple storage engines. All engines can be differentiated by three main characteristics [3]:

- *Persistence*: persistent engines retain their data and are power-fail safe, volatile engines are faster but only keep their content until the database is closed
- *Concurrency*: provide multithreaded scaling with thread-safe methods (e.g., `get()` and `put()`)
- *Keys' ordering*: "sorted" engines support range query methods (e.g., `get_above()`)

A detailed overview of the provided stable engines and their characteristics can be found in Table 4.2.

Engine	Description	Persistent	Concurrent	Sorted
cmap	Concurrent hash map	Yes	Yes	No
vcmap	Volatile concurrent hash map	No	Yes	No
vsmap	Volatile sorted hash map	No	No	Yes
blackhole	Accepts everything, returns nothing (Testing)	No	Yes	No

Table 4.2: *List of the stable pmemkv engines and their properties [5]*

Throughout this section, we solely focus on the `cmap` engine, as it is currently the only stable persistent engine. Internally, the `cmap` engine utilizes the concurrent hash map and persistent string from the `libpmem-cpp` library. Since `cmap` is implemented as a concurrent engine, multiple threads can simultaneously call the essential methods `get`, `put`, and `remove`.

For the performance evaluation of the individual methods, we employ `pmemkv-bench`. As explained in Chapter 3, `pmemkv-bench` is based on LevelDB's `db_bench`. The goal of our benchmarks is to identify general performance characteristics and investigate the multithreading performance of `pmemkv`. Therefore, we vary the key (8 B to 512 B) and value size (1 KiB to 8 KiB) for our single-threaded experiments. When the size of the key or the value is not varied by an experiment, we set the keysize to 16 B and the value size to 1 KiB, respectively. In our multithreaded benchmarks, we also use these values to evaluate the performance in the range of 1 to 32 threads.

In addition to the individual behavior of each method, we further examine the real-world scenario where multiple threads concurrently read and write to the key-value store. Hence, we study the performance progression over different read-percentage and number of threads.

Each data point presents the average of 10 consecutive runs. Between each run, the database is emptied to prevent previous executions from affecting subsequent measurements. In the *write* experiments, we load the database with 1 M *write* operations per thread, both in sequential and random order. The same number of entries (1 M) is also used in the *read* experiments. To establish a uniform baseline for reading performance, all keys are inserted in ascending order in advance. After all keys are inserted, we start the measurement for 10 M *read* operations per thread. We intentionally set the number of *read* operations higher than the number of entries to account for `pmemkv`'s caching behavior. With our `pmemkv-bench` adjustments, explained in Chapter 3, we can set the miss rate to 10%, resulting in a more realistic experimental setup.

Evaluation Starting with the writing performance of `pmemkv`, illustrated in Figure 4.14, we identify three unique characteristics. First, the throughput of inserting entries with random keys is around 16% higher than the throughput of sequential keys. The most likely explanation for this behavior is `pmemkv-bench` itself. When generating random numbers, these numbers are usually generated according to a certain distribution. In the case of `pmemkv-bench`, this distribution partially regenerates already inserted keys. Consequently, these already existing keys are not inserted again, instead, the previous value is overwritten. As no new entry has to be created and allocated when overwriting, the performance is significantly improved. Our testing revealed that approximately 30% of the keys were already present in the database. As a result, the throughput of sequential key order, which inserts truly unique keys with each operation, is significantly lower.

The second characteristic we want to highlight is the throughput progression for varying key sizes. For the key sizes of 8 B and 16 B, we observe the same performance level. After 16 B, the throughput gradually decreases, with a steeper drop after 256 B. Based on this progression, we infer that developers should use smaller key sizes, if applicable, to optimize the throughput of `pmemkv`. However, it should be noted that this statement does not apply

to value sizes. As seen from the middle plot, the throughput for value sizes decreases linearly.

The last examined characteristic for writing is the multithreaded performance. As depicted in the rightmost plot, the throughput increases steeply until it peaks at 16 threads for random insertion and at 18 threads for sequential insertion. Further increasing the number of threads has the opposite effect of reducing the performance. Like in previous sections, this behavior can be attributed to Intel's DCPMM. Especially, the limited write bandwidth of the DCPMM throttles the throughput of *write* operations for larger number of threads.

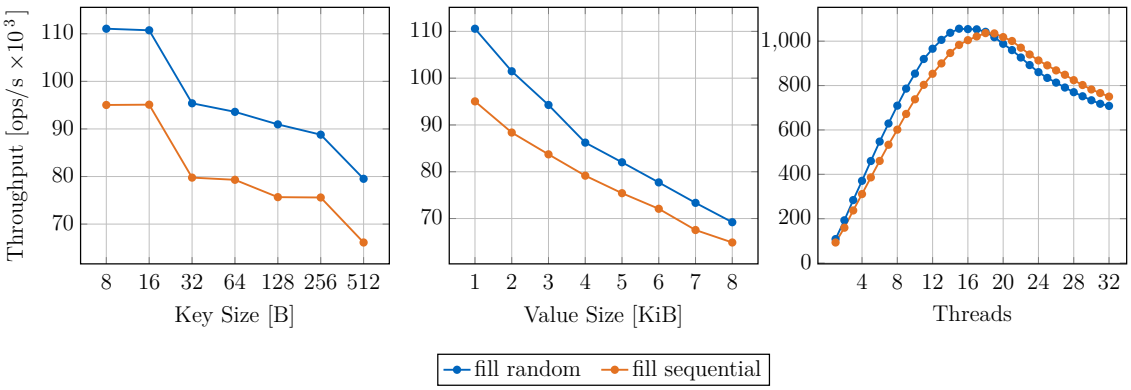


Figure 4.14: *pmemkv-bench: Performance of random/sequential fill operations for varying key size (Left) and value size (Middle) along with multithreaded performance (Right)*

In the experiments for *reads*, the throughput order is reversed between sequential and random. *Read* operations with consecutive keys are now up to 50% faster, as depicted in Figure 4.15. Due to our modifications, detailed in Chapter 3, we can ensure that both random and sequential key order find and read the same number of entries. However, the performance gap is larger than anticipated. We, therefore, further investigated the cause for the large discrepancy by measuring the individual sections of the *read* benchmark. Based on our measurements, we can state that our modifications have the same runtime for both variations and are thus not the reason for the performance difference. Instead, we observe that the actual `get()` operation itself takes longer for randomly selected keys. This insight is surprising, as hash maps typically have an average amortized complexity of $\mathcal{O}(1)$ and a worst-case complexity of $\mathcal{O}(n)$. We assume that the recurring keys generated by the used distribution happen to be stored in buckets with longer node lists. Consequently, the algorithm has to search through the bucket's node list. The average complexity of searching a list is $\mathcal{O}(n)$, which significantly reduces throughput. Nevertheless, we cannot attribute this behavior solely to `pmemkv-bench` and the distribution used. For this reason, the results from Section 4.3.2 should also be considered in order to evaluate the actual *read* performance of `pmemkv`.

As far as the multithreaded performance of *reads* is concerned, the performance progression of both variants is similar. The throughput increases linearly across all investigated number of threads. As with the single-threaded experiments, reading in sequential key

order provides the highest performance. Upon closer inspection of random reads, we notice a slight performance drop for more than 24 threads. This small deviation can be explained by the number of physical cores in our server. After 24 threads, the processor has to resort to virtual cores. As can be seen from the rightmost plot, this change reduces the slope of the throughput for more than 24 threads.

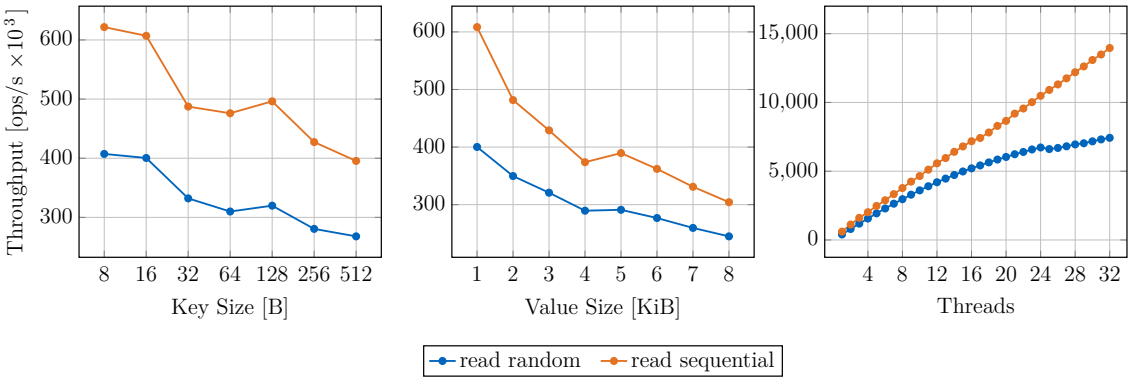


Figure 4.15: *pmemkv-bench: Performance of random/sequential read operations for varying key size (Left) and value size (Middle) along with multithreaded performance (Right) (Miss rate: 10%)*

After examining the individual performance of *reads* and *writes*, the final series of `pmemkv-bench` experiments evaluate the combination of both operations. For this purpose, each thread alternately performs random *read* and *write* operations on a shared key-value store. In this context, Figure 4.16 illustrates the throughput progression for varying read percentages and number of threads. The first plot focuses on the progression over varying read percentages with five selected numbers of threads (2, 4, 8, 16, 32). It can be seen that the higher the reading percentage, the more threads can be utilized. For example, when the reading percentage ranges from 10% to 50%, eight threads provide the best performance among all thread counts. Thus, the largest number of threads results in the best throughput only when the *read* operations account for 90% of all operations.

To further examine this characteristic, the second plot in Figure 4.16 details the throughput progression of selected read-percentages over a varying number of threads. From the left plot, we are able to identify 60%, 70%, and 80% as particularly interesting. For these three read percentages, the course of the detailed throughput curve is similar. It sharply rises up to a certain upper bound before gradually declining. The higher the read-percentage, the more threads can be used before the upper bound is reached, resulting in higher throughput. For example, the maximum throughput limit of 80% reads is achieved with 16 threads and hence 1.5× higher than the throughput of 60% read, which already peaks at around 12 threads.

From this observation, it can be deduced that the write bandwidth of DCPMM also affects the multithreaded performance of `pmemkv`. As mentioned earlier, only a small number of threads is required to saturate the available write bandwidth. Any additional threads only create more overhead and thus even reduce the throughput. Developers should therefore limit the number of concurrent threads if their application has a high write load.

As our experiment demonstrates, less than 16 threads are sufficient to achieve maximum throughput.

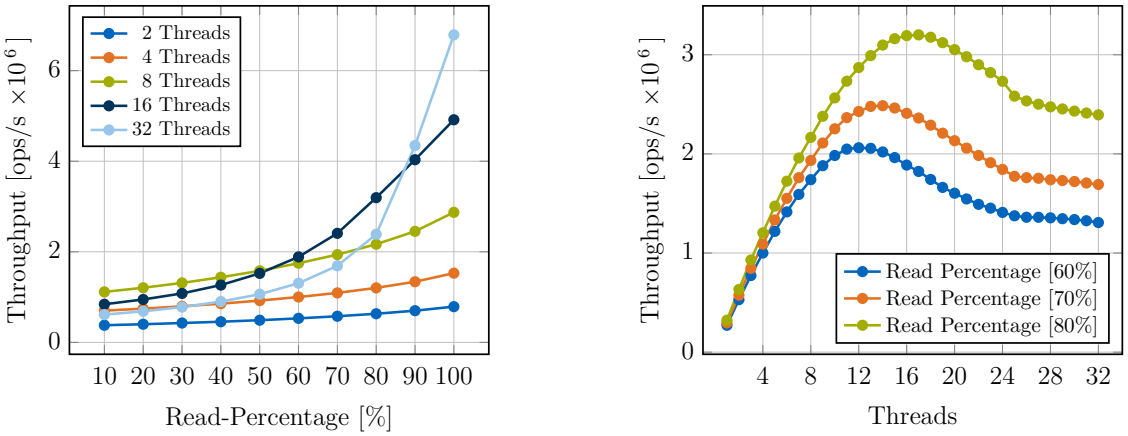


Figure 4.16: *pmemkv-bench: Multithreaded performance for random read/write operations for different read percentages (Left) and a more detailed investigation of selected percentages (Right).*

Summary `pmemkv` offers developers an easy way to integrate and exploit persistent memory in their applications. It provides the familiar interfaces for key-value stores and multiple storage engines for different degrees of flexibility and performance. In our experiments, `pmemkv` performed well across all parameters and operations. Especially smaller key and value sizes can benefit from the combination of `pmemkv` and persistent memory. However, our experiments also showed the limitations of `pmemkv`. In particular, the low write bandwidth of DCPMM affects the performance of `pmemkv` in write-intensive multithreaded workloads. For this reason, `pmemkv` should be used predominantly in applications that involve many smaller and fine-grained read accesses.

As for the `pmemkv-bench`, we propose a structural redesign of the framework. In order to improve accuracy and reliability, `pmemkv-bench` should migrate to a collection-based approach. Instead of creating the keys as they are used, they should be generated in advance and stored in a collection. This approach has two advantages: First, the measurements are more accurate because they do not reflect the generation of the keys. Second, the number of elements to be processed is known prior to measuring, and thus unwanted tendencies can be prevented (e.g. recurring keys).

4.3.2 Yahoo! Cloud Serving Benchmark (YCSB)

This section validates and complements our `pmemkv-bench` experiments with the Yahoo! Cloud Serving Benchmark (YCSB) [6]. The YCSB consists of a Java client application and a set of predefined workloads called the *YCSB Core Package*. For the experiments in this section, only the following four core workloads A to D are used:

- Workload A (Update Heavy): Read 50%, Update 50%
- Workload B (Read Heavy): Read 95%, Update 5%
- Workload C (Read Only): Read 100%
- Workload D (Read Latest): Read 95%, Insert 5%

More details about the YCSB and the workload properties can be found in Chapter 3. Similar to `pmemkv-bench` measurements, each data point in this section corresponds to the average of 10 consecutive runs. Between each run, the database is cleared to ensure the same baseline for all runs. Since all workloads have the same data set [6], we load the database once at the beginning of each run and then execute all workloads consecutively. For the individual experiments, the database is loaded with 100 K entries and 1 M operations are executed per workload. Unless altered, the value size is set to 1 KiB and the key size to 32 B.

Performance Evaluation of `pmemkv` We evaluate the performance of `pmemkv` using its Java bindings [33] and Intel's `pmemkv` YCSB driver [75]. As before, we benchmark `pmemkv` only using the persistent `cmap` storage engine.

Figure 4.17 depicts the single-threaded performance over varying value sizes, and Figure 4.18 the scalability across multiple threads. It is noticeable that the more read-dominated workloads perform better overall. This becomes particularly obvious when comparing *Load* (0% Read) with *Workload C* (100% Read). *Workload C* has a $3 - 3.5\times$ better throughput than *Load*. An explanation for this performance difference is the hash map itself. When inserting values into the hash map, the load level increases steadily. Once the load level exceeds a certain threshold, the algorithm initiates rehashing in order to maintain the $\mathcal{O}(1)$ complexity for reads. Since rehashing involves recalculation of all inserted values, this process is very costly and significantly degrades the insertion performance. In return, the performance of subsequent read operations remains constant [76]. Another aspect we want to emphasize again is the read-write asymmetry of Intel's DCPMM. As seen throughout the previous experiments, the lower write bandwidth of the DCPMM impacts the performance of libraries like `pmemkv`. This asymmetrical behavior is also reflected in this experiment.

In terms of general throughput progression, we can distinguish between the read-heavy workloads (B, C, D) and the write-heavy workloads (Load, A). Starting with single-threaded experiments, we observe a gradual decrease in throughput across all value sizes for the write-heavy workloads. In contrast, the read-heavy workloads remain on a steady level up to 512 B before declining more steeply. It should be further noted that the progression of *Workload B* and *Workload D* are almost identical. Hence, it can be deduced that updating and inserting have identical performance for `pmemkv`.

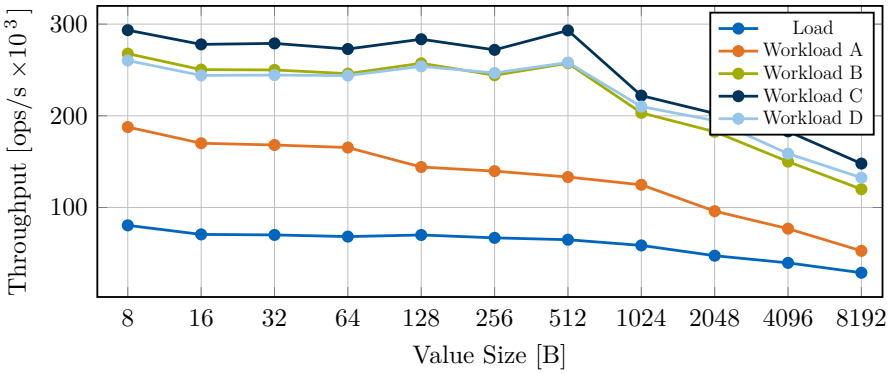


Figure 4.17: *YCSB: Single threaded performance of pmemkv with* `cmap` *engine across varying value sizes*

A similar overall pattern emerges for multithreaded experiments. The performance between the read-heavy and write-heavy workloads remains asymmetrical, with *Load* being around 3× slower than *Workload C*. Despite this asymmetry, all operations can increase their throughput by a factor of $3 - 3.6$ from 1 to 16 threads.

Overall, we conclude that the `pmemkv` performance scales well across the different value sizes and thread counts. The YCSB benchmarks further confirm the earlier findings that the read-intensive workloads are better suited for `pmemkv`.

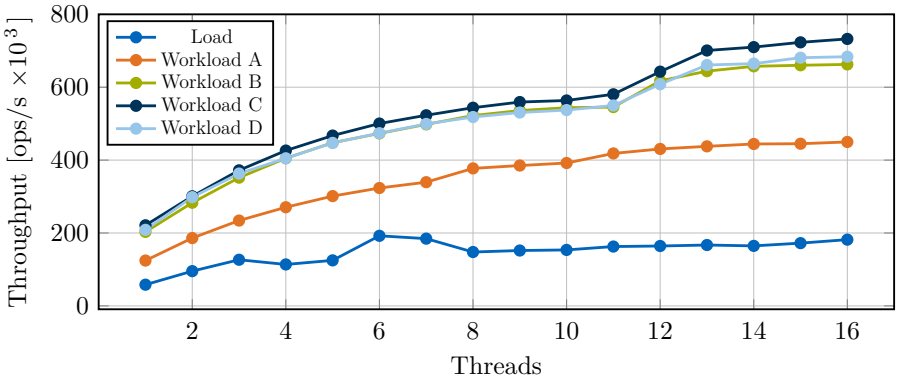


Figure 4.18: *YCSB: Multithreaded Performance of pmemkv with* `cmap` *engine*

Comparison with Modern Databases After having established the `pmemkv`'s YCSB performance, we proceed to the comparison with other database systems. As detailed in Chapter 3, we compare the single-threaded performance of `pmemkv` with other PM-aware databases, namely PM-Redis, PM-Postgres, and MongoDB-PMSE. The results of our comparison are shown in Figure 4.19.

Across all databases, `pmemkv` offers the highest performance. Especially, for read-dominated workloads, `pmemkv` has a 4× better throughput compared to the next best alternative. This trend is seen across all databases, with better throughput for read-intensive workloads B, C, and D. For the remaining databases, the in-memory key-value store Redis delivers the second-best performance. It is about 2× faster than MongoDB and Postgres. From this observation, we deduce that in-memory databases in particular provide a solid foundation for persistent memory modifications. Their internal algorithms and data structures are already designed for faster memory access and byte-addressability. Using the PMDK's functionality ensures the persistence of these data structures without logging all operations to the *append-only file* (AOF).

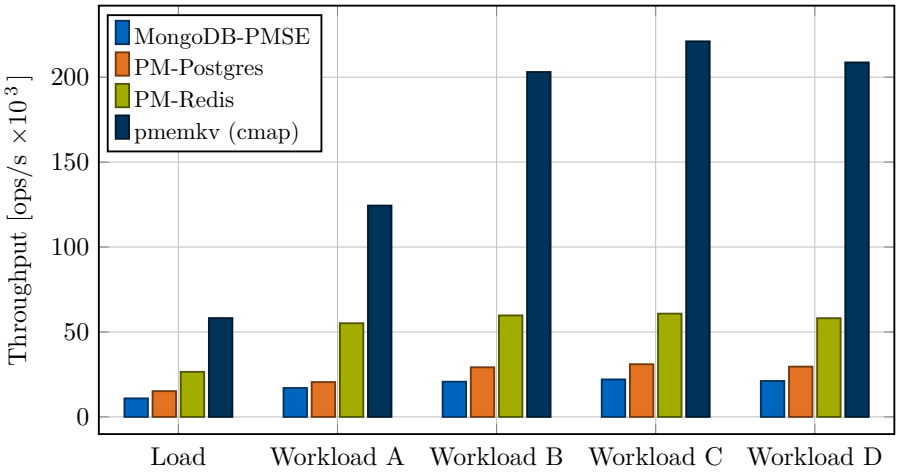


Figure 4.19: *YCSB: Comparison of different databases utilizing the PMDK (100 K records and 1 M operations)*

The two remaining databases, MongoDB and Postgres, are found to have the lowest throughput among the databases examined. It can be assumed that this observation correlates with the high cost of the client-server communication as well as the overhead induced by the query processing engine. To investigate the behavior of both systems in detail, we compare these two PMDK-enabled database systems with their unmodified counterparts. In order to minimize the impact of the storage medium on the results, all databases store their contents on persistent memory. This way, only the difference in performance resulting from the use of the PMDK is measured. Figure 4.20 presents the results of this comparison.

Different characteristics can be observed for both database systems. In the case of Postgres, the PMDK-enabled variant shows a slightly higher throughput across all workloads. More specifically, PM-Postgres performs around $2-4\%$ better than the unmodified version. This is in line with earlier results reported by Menjo [77]. For MongoDB, we notice the opposite. The unmodified version performs better in all workloads except *Workload A*. A particularly large difference occurs at *Load*. In this workload, the unmodified version of MongoDB is around 80% faster than Mongo-PMSE. By comparison, the difference in the other core workloads is only about $2\% - 6\%$. Only in *Workload A*, the PMDK-enabled

MongoDB-PMSE can achieve a 4% higher throughout than MongoDB.

Based on our results, we deduce that replacing file operations in existing database system with PMDK's memory operation improves performance only to a certain extent. Nevertheless, it should be noted that in a highly optimized modern database system, a small percentage can already mean a significant difference in absolute terms.

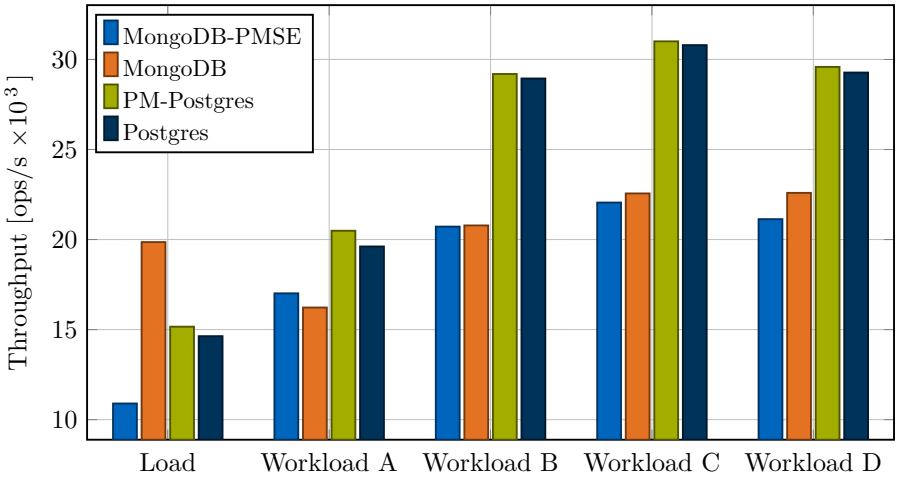


Figure 4.20: *YCSB: Comparison of MongoDB and PostgreSQL with their PM-enabled counterparts (100 K records and 1 M operations)*

Summary In this section, we used the YCSB to determine the potential of the PMDK for databases. `pmemkv` has demonstrated feasible performance when the PMDK's core concepts and features are considered in the design and implementation. Particularly for read-dominated workloads, the proper use of persistent memory and the PMDK can significantly improve performance. Using other database systems as an example, we found that substituting the existing file operations with persistent memory instructions can enhance the throughput of current database systems. However, in order to fully exploit the potential of persistent memory and the PMDK, the database systems have to be profoundly reengineered to take advantage of the direct access and the byte-addressability of persistent memory.

5 Related Work

In this thesis, we evaluated the performance of the PMDK's core components and explored the opportunities that the PMDK can provide for modern software and database architectures. This chapter presents related literature to these areas of application by giving an overview of previous research. We break down the related contributions into separate fields: evaluation of non-volatile memory hardware, alternative persistent memory libraries, and persistent memory applications.

5.1 Persistent Memory Hardware

After the release of the first commercially available non-volatile memory hardware, the Intel Optane DC Persistent Memory Module (DCPMM), several papers investigate its performance characteristics. The work of Izraelevitz *et al.* [15] is one of two initial studies to examine the performance of the DCPMM. In their extensive analysis, they measure the latency and bandwidth for DCPMM as volatile main memory and as persistent, byte-addressable memory. Their numbers confirm that DCPMM occupies the space between DRAM and SSD in terms of latency and bandwidth. Aside from this, they highlight the asymmetrical load and store performance of the DCPMM, which has important implications for future data structures and application areas.

The second initial study of the DCPMM is the paper of van Renen *et al.* [17]. As opposed to Izraelevitz *et al.*, their paper not only describes the performance characteristics but also proposes low-level guidelines for efficient persistent memory usage. For instance, they recommend developers to frequently force data out of the CPU caches to achieve a higher write bandwidth. For performance-critical code, they further suggest to still use DRAM because of its superior latency and bandwidth.

According to both studies, the smallest efficient access granularity for DCPMM is 256 B. All smaller data sizes have the same latency, thus wasting bandwidth. For data sizes larger than 256 B, one should ensure that they are multiples of 256 B. This observation is explained by the size of the DCPMM's internal write-combining buffer.

A later analysis of the DCPMM by Yang *et al.* [12] confirms the inefficiency of data accesses smaller than 256 B. Their paper investigates the performance properties of the Intel DIMMs at the micro and macro level. The researchers discovers that the DCPMM is much more affected by access size, type, pattern, and degree of concurrency than DRAM.

Other fundamental characteristics of the DCPMM are revealed by Lersch *et al.* [78]. In their paper, the authors point out that the bandwidth for persistent memory is a scarce resource. Even a small number of threads is sufficient to saturate the available bandwidth. Therefore, to maximize persistent memory performance, they propose to have many smaller DCPMM DIMMs rather than a few large ones to leverage the available memory channels.

A more reality-focused performance evaluation is done by Weiland *et al.* [16], whose paper investigates the benefits of DCPMM for high-performance scientific applications. They demonstrate that primarily scientific applications, which are limited by the main

memory capacity, can benefit from running DCPMM in *Memory Mode*. This finding is supported by Mironov *et al.* [14], who examine the performance gains of scientific workloads with DCPMM in *Memory Mode*. The two papers agree that scientific applications need profound modifications to exploit the full potential of DCPMMs in *App Direct Mode*. However, both emphasize that this effort can pay off in the long run and significantly accelerate execution in future projects.

5.2 Persistent Memory Libraries

The PMDK is the most used library and the de-facto standard for persistent memory programming. Different authors have analyzed the overhead of the PMDK. In their work, Wang and Diestelhorst [79] discuss the transactional overhead introduced with the PMDK's `libpmemobj` API. For their validation, they use small kernel workloads implemented with the transactional API of `libpmemobj` and a PMDK-enabled Redis-server in combination with TPCC payloads. They propose the usage of hardware acceleration in areas with considerable overhead, e.g. logging, persisting, ordering, and pointer translation. A similar approach for performance evaluation is taken by Islam and Dai [80]. In their work, they design *storeds-bench* and implement storage data structures in four different versions: DRAM, volatile PMDK (`libvmem`), persistent PMDK (`libpmemobj`), and transactional PMDK (`libpmemobj`). The authors discover that read-dominated workloads are much less affected by persistent memory than write-dominated ones. Additionally, they note that the performance of data structures using the PMDK may notably diverge from their volatile DRAM counterparts. As a consequence, they advise developers to evaluate their data structures on persistent memory to account for the distinct characteristics of persistent memory.

All previous approaches, in contrast to this thesis, have investigated the overhead of the PMDK in conjunction with data structures. With our contribution, we pursue a systematic performance evaluation of the PMDK's core components and the PMDK-enabled key-value-store `pmemkv`.

Aside from the PMDK, there are several alternative libraries, some of which take unique approaches to assist developers in leveraging persistent memory. In this section, we want to highlight a few of them.

In 2011, Coburn *et al.* [45] introduced NV-Heaps, a C++ library for Linux focusing on user-defined objects. It provides basic functionalities such as type-safe pointer to persistent memory, a persistent memory allocator, and a garbage collection via reference counting. The library aims to provide both atomic and transactional semantics with familiar programming interfaces. NV-Heaps, however, requires processor architectural support in the form of atomic 8 byte writes, and epoch barriers originally developed for BPFS [81]. Another limitation of NV-Heaps is that it only allows closing pools at program exit in order to avoid unsafe volatile to non-volatile pointers.

In the same year, Volos *et al.* [19] introduced Mnemosyne, a C-based user-level framework consisting of Linux kernel modifications, a C library, and a custom compiler/linker. It includes persistent memory regions, low-level primitives for persistent variables, and consistent updates of arbitrary data structures through a transaction mechanism. Mnemosyne, unlike NV-Heaps, does not require any special hardware support. However, it also lacks some features from NV-Heaps, such as type-safe pointers and garbage collection.

In addition to the two mentioned libraries, there are some recent additions to the persistent memory library landscape, namely Pangolin [82], Corundum [83], and Clobber-NVM [25].

With Pangolin, Zhang and Swanson [82] present a fault-tolerant persistent object library with the familiar interface of the PMDK's `libpmemobj`. Pangolin combines checksums, parity, and micro-buffering to protect integrity and detect corruption on persistent memory. With its architecture, Pangolin guarantees that it can recover from any 4 KB page loss in a pool while requiring less storage overhead than other libraries.

Corundum takes a different approach. Hoseinzadeh and Swanson [83] designed the library on the language-specific features of Rust. Rust's static-checking, for example, helps Corundum to enforce key persistent memory programming invariants such as unsafe pointer creation between volatile and non-volatile memory regions. Corundum further ensures that modifications of persistent memory solely happen inside transactions and only after all updates are logged. As a consequence, Corundum does not support log-free programming but prevents most persistent memory allocation errors. The dependence of Corundum on Rust, however, also has its downsides, like the susceptibility to memory leaks in cyclic-data structures [84].

With Clobber-NVM, Xu *et al.* [25] present a joint compiler/runtime library that uses a novel logging strategy called clobber-logging. This new logging strategy uses the recovery-via-resumption technique: An undo-log only logs the transaction inputs that are overwritten during the transaction execution, thus reducing the frequency of the logging operations. After a failure, Clobber-NVM recovers to a consistent state by restoring all overwritten values and re-executing any interrupted transactions. To identify those transaction inputs, Clobber-NVM uses a custom compiler resulting in significantly reduced logging overhead at runtime. During execution, the runtime library manages the callbacks inserted by the compiler and initiates the recovery of interrupted transactions after a crash. The runtime library itself and the clobber log for logging the transaction inputs are implemented with the PMDK's `libpmemobj`.

5.3 Persistent Memory Applications

In Section 4.3, we mainly demonstrated the advantages of persistent memory for future databases. The same path is also taken by *Memhive* [85], a commercially available PMDK-enabled PostgreSQL database with persistent memory support. *Memhive* aims to provide significantly improved performance and transaction throughput while simultaneously eliminating the need for cache-warmup.

Apart from databases, there are other applications that can significantly benefit from persistent memory. The most promising areas of applications are presented in this section.

In their paper, Pydipaty *et al.* [86] investigate the performance of the distributed storage platform *Ceph* in a system with persistent memory. The authors demonstrate that the performance can be increased by 100%. Based on their findings, they assume that in future systems, the bottleneck for distributed storage platforms might shift from I/O to computation. Consequently, they propose new approaches for storage and network design, such as exploiting the byte-addressability of persistent memory.

The impact of persistent memory on high-performance computing (HPC) applications is examined by Liu *et al.* [20]. The better pricing, higher capacities, and the performance close to DRAM make persistent memory very appealing for applications with high demands on I/O performance and storage capability. Liu *et al.* detail the impact of block-based non-volatile memory on POSIX I/O and MPI I/O, an interface for performing parallel I/O operations within distributed memory programs. Similar to previous reports [14, 16], the authors suggest reconsidering the existing I/O-stack and optimizing performance for non-volatile memory instead of block-based storage.

Due to the special characteristics of persistent memory, the evaluation of data structures is only possible on real hardware. For this purpose, Hao *et al.* [87] propose PiBench, an interactive benchmarking framework for persistent memory indexes. With PiBench, the authors aim to provide researchers and developers with a consistent environment to benchmark their index implementation, analyze the results and compare them with each other. During implementation, special attention was paid to the reproducibility of the results. After assessing common indexes with PiBench, the authors stress that future indexes should reduce the persistent memory accesses to not exhaust the limited bandwidth.

6 Limitations and Future Work

This section evaluates our accomplished goals and critically discusses respective limitations of our experimental methodology and the reported findings. Based on these limitations, we propose possible future work.

Methodology The benchmarking environment plays a significant role in any performance evaluation. As explained in Section 3.2, we decided to use Docker containers as foundation for our benchmarking suite to ensure a consistent testing environment. Furthermore, the containerization of our benchmark suite facilitates the execution of experiments on different hardware configurations. Nevertheless, this flexibility comes at the cost of increased overhead. We attempted to minimize this overhead as much as possible by using Docker Direct Access. To exclude a decisive overhead, we performed selected experiments without Docker, but could not detect any noticeable difference. However, there is still a possibility that our containerized approach has influenced the measurements. Analyzing and quantifying these effects on persistent memory remains an open topic for future work.

Regarding the benchmarks themselves, we primarily focused on microbenchmarking and macrobenchmarking. This is an effective approach to evaluate the performance of individual components, but it is not sufficient to directly derive the real-world performance. In order to reliably quantify the actual performance improvements from modifications for individual applications, it is inevitable to implement them. As the example of MongoDB and PostgreSQL in Section 4.3 demonstrated, superficial changes are not necessarily adequate to reflect the actual benefits. Therefore, new concepts have to be considered as part of the modifications and integrated into the application accordingly. Despite the limited expressiveness concerning real-world performance, we do believe that our microbenchmarks are comprehensive enough to evaluate and compare the performance of individual methods. For more sophisticated results, future work can use our benchmark suite as a basis to investigate further aspects of the PMDK. Since our work only concerns the local libraries, an evaluation of the remote libraries `librpmem` and `librpma` would be a useful possibility to broaden our results.

During our evaluation, we mainly focused on the average throughput for each operation. While this metric is valuable to most data structures and algorithms, there are other relevant parameters to consider, such as latency and bandwidth. Aside from additional parameters, one can also examine other statistical measures, in the context of benchmarking particularly percentiles. As the main objective of our thesis is to identify the PMDK's characteristics across different dimensions, we chose throughput as our primary metric. We do, however, measure the other metrics in the majority of our experiments as well, so no adjustments are required to obtain them. Consequently, other researchers and developers can employ our benchmarking suite to evaluate the PMDK against other criteria and metrics.

Findings Our presented findings are occasionally surprising, but overall plausible and reproducible. While conducting our experiments, one framework in particular caught our attention with unexpected results: `pmemkv-bench`. The framework exhibits a noticeable discrepancy between random and sequential operations. After some investigation and exploration, we identified the random distribution and key computation as the main cause. To some extent, we were able to improve the behavior by making our own adjustments, which are described in Section 3.2. However, these modifications could not eliminate all peculiarities and thus the random key generation remains an issue. For this reason, other benchmarking frameworks, such as the YCSB, are preferable for analyzing and evaluating the performance of the `pmemkv`'s performance.

As explained in Section 4.1, we conduct our experiments on a machine with Intel DCPMM hardware. The Intel DCPMM, as seen throughout the thesis, has its own set of performance properties. This makes it difficult to distinguish whether some of the properties shown can be attributable to the DCPMM or the PMDK. For example, the observed throughput increase at 256 B is most certainly related to the DCPMM's write-combining buffer, which also has a size of 256 B. Another characteristic that might be related to the DCPMM is the asymmetrical read/write performance we observed with the YCSB in Section 4.3.

At the time of writing, the DCPMM is the only commercially available persistent memory hardware. Hence, it is difficult to validate our findings on alternative persistent memory, which might have different performance properties. Due to our container-based approach, the validation can easily be done at a later time, once the alternative hardware becomes available. In the future, for example, a re-evaluation with the second generation of Intel's DCPMM [88] is conceivable.

A re-evaluation should also be performed as soon as processors with optimized CPU instructions, namely `CLWB` and `MOVDIR64B`, become available. In Chapter 2, we presented various CPU instructions for flushing the CPU caches. Currently, the optimal method `CLWB` only serves as a wrapper for `CLFLUSHOPT`, so both invalidate the cache line when flushing it to persistent memory. Starting with Ice Lake, Intel fully implemented CLWB without cache line invalidation. This change in conjunction with the upcoming instruction `MOVDIR64B`, which ensures 64 B atomicity for stores, may significantly improve the efficiency of memory operations to persistent memory. We therefore propose to reinvestigate the possible implications of these instructions in future work.

7 Conclusion

The primary accomplishments of our thesis can be summarized in three points:

Firstly, we approached the topic of persistent memory from a theoretical angle. We explained the properties of persistent memory in comparison to conventional memory and storage devices. These properties were then described in more detail using the first commercially available non-volatile memory technology, the Intel DCPMM. Especially, the peculiarities such as the asymmetrical read-write performance, the 256 B writing-combining buffer, and the two operation modes were highlighted. Following the hardware background, we proceeded with the essential concepts of persistent memory programming, such as cache line flushing and ordering barriers. These concepts are pivotal to comprehend the PMDK's structure and functionality, which we subsequently presented and described.

After having established the background knowledge, we outlined the design and implementation of our versatile open-source benchmarking suite. To provide a consistent testing environment, the benchmark suite is built on Docker and employs multiple benchmarking frameworks, more precisely, `pmembench`, `pmemkv-bench`, and the YCSB. In addition, these frameworks were modified and complemented with configuration files and scripts to enable automated benchmark execution. Through this approach, we were able to scrutinize the PMDK's performance at the micro and macro level.

Finally, in more than 570 hours of machine time on a persistent memory server, we thoroughly evaluated the PMDK's performance. By utilizing `pmembench`, we were able to analyze the performance of the PMDK's core components. We started by examining the low-level functionality of `libpmem` and `libpmem2`. Based on our measurements, we identified 64 B and 256 B as the optimal ratio between performance and data size. Moreover, we discovered that the larger the data size, the fewer threads are required to achieve the maximum throughput. After having determined the low-level characteristics, we evaluated the mechanism of `libpmemobj`, including transactions, dynamic memory management, and synchronization primitives. Our findings show that all mechanisms scale well across data sizes and number of threads with reasonable overhead compared to their volatile counterparts. Following the evaluation of the core components, we introduced the PMDK's key-value store `pmemkv` and compared its performance with other state-of-the-art database systems using the YCSB. Our results demonstrate that incorporating the new concepts of the PMDK can significantly improve performance by up to four times. Particularly for read-dominated workloads, we noticed that the PMDK's direct, finer-grained access to persistent memory provides major benefits.

In conclusion, these contributions support developers who want to leverage the PMDK to exploit the benefits of persistent memory by examining its characteristics and suggesting guidelines for optimal use.

List of Figures

List of Tables

Bibliography

[1] IDC and Statista, *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025 (in zettabytes)*, Jun. 2021. [Online]. Available: https://www.statista.com/statistics/871513/worldwide-data-created/ (visited on 07/17/2021).

[2] Intel, *Intel Optane Persistent Memory*. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html (visited on 07/18/2021).

[3] S. Scargall, *Programming Persistent Memory: A Comprehensive Guide for Developers*. Berkeley, CA: Apress, 2020, ISBN: 978-1-4842-4932-1. DOI: 10.1007/978-1-4842-4932-1.

[4] Intel, *Persistent Memory Development Kit (PMDK)*, 2021. [Online]. Available: https://pmem.io/pmdk/ (visited on 06/30/2021).

[5] Intel, *Pmemkv*, 2021. [Online]. Available: https://pmem.io/pmemkv/ (visited on 07/03/2021).

[6] B. F. Cooper, *YCSB*. [Online]. Available: https://github.com/brianfrankcooper/YCSB (visited on 07/27/2021).

[7] R. Love, *Linux Kernel Development*, 3rd. Addison-Wesley Professional, 2010, ISBN: 978-0-672-32946-3.

[8] H.-S. P. Wong *et al.*, "Phase Change Memory," *Proceedings of the IEEE*, vol. 98, no. 12, pp. 2201–2227, Dec. 2010, ISSN: 0018-9219, 1558-2256. DOI: 10.1109/JPROC.2010.2070050.

[9] M. Hosomi *et al.*, "A novel nonvolatile memory with spin torque transfer magnetization switching: Spin-ram," in *IEEE InternationalElectron Devices Meeting, 2005. IEDM Technical Digest.*, Tempe, Arizon, USA: IEEE, 2005, pp. 459–462, ISBN: 978-0-7803-9268-7. DOI: 10.1109/IEDM.2005.1609379.

[10] Intel and Micron, *3D XPoint™: A Breakthrough in Non-Volatile Memory Technology*, 2015. [Online]. Available: https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html (visited on 07/02/2021).

[11] Intel, *PMDK Introduction*. [Online]. Available: https://docs.pmem.io/persistent-memory/getting-started-guide/what-is-pmdk (visited on 07/06/2021).

[12] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," in *Proceedings of the 18th USENIX Conference on File and Storage Technologies (FAST '20)*, Santa Clara, CA, USA: USENIX Association, Feb. 2020, pp. 169–182, ISBN: 978-1-939133-12-0.

[13] S. Gugnani, A. Kashyap, and X. Lu, "Understanding the idiosyncrasies of real persistent memory," *Proceedings of the VLDB Endowment*, vol. 14, no. 4, pp. 626–639, Dec. 2020, ISSN: 2150-8097. DOI: 10.14778/3436905.3436921.

[14] V. Mironov, I. Chernykh, I. Kulikov, A. Moskovsky, E. Epifanovsky, and A. Kudryavtsev, "Performance Evaluation of the Intel Optane DC Memory With Scientific Benchmarks," in *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, Denver, CO, USA: IEEE, Nov. 2019, pp. 1–6, ISBN: 978-1-72816-007-8. DOI: 10.1109/MCHPC49590.2019.00008.

[15] J. Izraelevitz *et al.*, "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module," *arXiv:1903.05714 [cs]*, Aug. 2019. arXiv: 1903.05714 [cs].

[16] M. Weiland *et al.*, "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*, Denver Colorado: ACM, Nov. 2019, pp. 1–19, ISBN: 978-1-4503-6229-0. DOI: 10.1145/3295500.3356159.

[17] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, "Persistent Memory I/O Primitives," in *Proceedings of the 15th International Workshop on Data Management on New Hardware - DaMoN'19*, Amsterdam, Netherlands: ACM Press, 2019, pp. 1–7, ISBN: 978-1-4503-6801-8. DOI: 10.1145/3329785.3329930.

[18] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, Intel.

[19] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight Persistent Memory," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, Newport Beach, California, USA: Association for Computing Machinery, May 2011, pp. 91–104, ISBN: 978-1-4503-0266-1.

[20] W. Liu, K. Wu, J. Liu, F. Chen, and D. Li, "Performance Evaluation and Modeling of HPC I/O on Non-Volatile Memory," in *2017 International Conference on Networking, Architecture, and Storage (NAS)*, Shenzhen, China: IEEE, Aug. 2017, pp. 1–10, ISBN: 978-1-5386-3486-8. DOI: 10.1109/NAS.2017.8026869.

[21] S. R. Dulloor *et al.*, "System software for persistent memory," in *Proceedings of the Ninth European Conference on Computer Systems - EuroSys '14*, Amsterdam, The Netherlands: ACM Press, 2014, pp. 1–15, ISBN: 978-1-4503-2704-6. DOI: 10.1145/2592798.2592814.

[22] N.-A. Tehrany, "Evaluating Performance Characteristics of the PMDK Persistent Memory Software Stack," Bachelor Thesis, Vrije Universiteit Amsterdam, Aug. 2020.

[23] *Direct Access for Files (DAX)*. [Online]. Available: https://www.kernel.org/doc/Documentation/filesystems/dax.txt (visited on 07/28/2021).

[24] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging," *ACM Transactions on Database Systems*, vol. 17, no. 1, pp. 94–162, Mar. 1992, ISSN: 0362-5915, 1557-4644. DOI: 10.1145/128765.128770.

[25] Y. Xu, J. Izraelevitz, and S. Swanson, "Clobber-NVM: Log less, re-execute more," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Virtual USA: ACM, Apr. 2021, pp. 346–359, ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446730.

[26] A. Kalia, D. Andersen, and M. Kaminsky, "Challenges and solutions for fast remote persistent memory access," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, Virtual Event USA: ACM, Oct. 2020, pp. 105–119, ISBN: 978-1-4503-8137-6. DOI: 10.1145/3419111.3421294.

[27] Intel, *Announcing the Persistent Memory Development Kit*, Dec. 2017. [Online]. Available: https://pmem.io/2017/12/11/NVML-is-now-PMDK.html (visited on 07/06/2021).

[28] SNIA, *NVM Programming Model*, 2017. [Online]. Available: https://www.snia.org/sites/default/files/technical_work/final/NVMProgrammingModel_v1.2.pdf (visited on 07/20/2021).

[29] Intel, *Libpmem*, 2021. [Online]. Available: https://pmem.io/pmdk/libpmem/ (visited on 07/20/2021).

[30] Intel, *Libpmem2*, 2021. [Online]. Available: https://pmem.io/pmdk/libpmem2/ (visited on 07/20/2021).

[31] Intel, *Libpmemobj*, 2021. [Online]. Available: https://pmem.io/pmdk/libpmemobj/ (visited on 07/20/2021).

[32] Intel, *Libpmemobj-cpp*, 2021. [Online]. Available: https://pmem.io/libpmemobj-cpp/ (visited on 07/20/2021).

[33] Intel, *Pmemkv-java*, 2021. [Online]. Available: https://pmem.io/pmemkv-java/ (visited on 07/17/2021).

[34] Intel, *Pmemkv-nodejs*, 2021. [Online]. Available: https://pmem.io/pmemkv-nodejs/ (visited on 07/17/2021).

[35] Intel, *Pmemkv-python*, 2021. [Online]. Available: https://pmem.io/pmemkv-python/ (visited on 07/17/2021).

[36] Intel, *Pmemkv-ruby*, 2021. [Online]. Available: https://github.com/pmem/pmemkv-ruby (visited on 07/17/2021).

[37] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley & Sons, Apr. 1991, ISBN: 978-0-471-50336-1.

[38] G. Heiser, *Systems Benchmarking Crimes*, 2021. [Online]. Available: https://www.cse.unsw.edu.au/~gernot/benchmarking-crimes.html (visited on 07/27/2021).

[39] J. Ousterhout, "Always measure one level deeper," *Communications of the ACM*, vol. 61, no. 7, pp. 74–83, Jun. 2018, ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3213770.

[40] N. Poggi, "Microbenchmark," in *Encyclopedia of Big Data Technologies*, S. Sakr and A. Y. Zomaya, Eds., Cham: Springer International Publishing, 2019, pp. 1143–1152, ISBN: 978-3-319-77525-8. DOI: 10.1007/978-3-319-77525-8_111.

[41] Intel, *PMDK-benchmark*. [Online]. Available: https://github.com/pmem/pmdk/tree/master/src/benchmarks (visited on 07/17/2021).

[42] Intel, *Pmemkv-bench*, 2021. [Online]. Available: https://github.com/pmem/pmemkv-bench (visited on 07/03/2021).

[43] Google, *LevelDB*, 2021. [Online]. Available: https://github.com/google/leveldb (visited on 07/03/2021).

[44] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud Computing - SoCC '10*, Indianapolis, Indiana, USA: ACM Press, 2010, p. 143, ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807152.

[45] J. Coburn *et al.*, "NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, Newport Beach, California, USA: Association for Computing Machinery, May 2011, pp. 105–118, ISBN: 978-1-4503-0266-1. DOI: 10.1145/1961295.1950380.

[46] E. R. Giles, "Container-based virtualization for byte-addressable NVM data storage," in *2016 IEEE International Conference on Big Data (Big Data)*, Washington DC,USA: IEEE, Dec. 2016, pp. 2754–2763, ISBN: 978-1-4673-9005-7. DOI: 10.1109/BigData.2016.7840923.

[47] R. Jandow, *PMDK Performance Evaluation*, 2021. [Online]. Available: https://github.com/RobertJndw/PMDK-performance-evaluation (visited on 08/07/2021).

[48] *Memkind*, 2021. [Online]. Available: http://memkind.github.io/memkind/ (visited on 07/17/2021).

[49] D. Inc., *Docker*, 2021. [Online]. Available: https://www.docker.com (visited on 07/17/2021).

[50] P. G. D. Group, *PostgreSQL*, 2021. [Online]. Available: https://www.postgresql.org (visited on 07/17/2021).

[51] Intel, *PMEM-Redis*. [Online]. Available: https://github.com/pmem/pmem-redis (visited on 07/17/2021).

[52] M. Inc., *MongoDB*. [Online]. Available: https://www.mongodb.com (visited on 07/17/2021).

[53] Intel, *PMSE*. [Online]. Available: https://github.com/pmem/pmse (visited on 07/17/2021).

[54] R. Jandow, *PMDK*, 2021. [Online]. Available: https://github.com/RobertJndw/pmdk (visited on 07/28/2021).

[55] R. Jandow, *Pmemkv-bench*, 2021. [Online]. Available: https://github.com/RobertJndw/pmemkv-bench (visited on 08/07/2021).

[56] R. Jandow, *Pmemkv-ycsb*, 2021. [Online]. Available: https://github.com/RobertJndw/YCSB (visited on 07/28/2021).

[57] T. Menjo, *Applying PMDK to WAL operations for persistent memory*, Feb. 2021. [Online]. Available: https://www.postgresql.org/message-id/CAOwnP3O3O1GbHpddUAzT=CP3aMpX99=1WtBAfsRZYe2Ui53MFQ@mail.gmail.com (visited on 05/22/2021).

[58] A. Rudoff, *Pmem_drain an empty function on Intel platforms?* PMEM Google Group. [Online]. Available: https://groups.google.com/g/pmem/c/L_LVHNf4al0/m/SOP5paeEBwAJ (visited on 08/04/2021).

[59] Intel, *Libpmem pmem_flush() Documentation*, 2021. [Online]. Available: https://pmem.io/pmdk/manpages/linux/v1.11/libpmem/pmem_flush.3 (visited on 07/20/2021).

[60] *Persistent Huge Pages*. [Online]. Available: https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt (visited on 07/25/2021).

[61] S. Iwata, "An investigation of performance problems with msync() system calls on filesystem DAX," in *Proceedings of the 14th ACM International Conference on Systems and Storage*, Haifa Israel: ACM, Jun. 2021, pp. 1–1, ISBN: 978-1-4503-8398-1. DOI: 10.1145/3456727.3463831.

[62] Intel, *Libpmem pmem_memcpy() and pmem_memset() Documentation*, 2021. [Online]. Available: https://pmem.io/pmdk/manpages/linux/v1.11/libpmem/pmem_memmove_persist.3 (visited on 07/20/2021).

[63] A. Rudoff, *8-byte atomicity & larger store operations*, PMEM Google Group. [Online]. Available: https://groups.google.com/g/pmem/c/6_5daOuEI00 (visited on 07/29/2021).

[64] A. Kalia, *Memcpy() slower without subsequent pmem_flush/pmem_persist*, Private Communication, Jun. 2021.

[65] Intel, *Libpmemobj - Type safety macros*, 2021. [Online]. Available: https://pmem.io/2015/06/11/type-safety-macros.html (visited on 07/20/2021).

[66] Intel, *Libpmemobj - Introduce hybrid transactions*, Jun. 2018. [Online]. Available: https://github.com/pmem/pmdk/pull/2716 (visited on 07/20/2021).

[67] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson, "Hoard: A Scalable Memory Allocator for Multithreaded Applications," p. 12,

[68] U. Upadhyayula, *Introduction to Persistent Memory Allocator and Transactions*. [Online]. Available: https://software.intel.com/content/www/us/en/develop/videos/introduction-to-persistent-memory-allocator-and-transactions.html (visited on 07/30/2021).

[69] Intel, *Libpmemobj - pmemobj_alloc() Documentation*, 2021. [Online]. Available: https://pmem.io/pmdk/manpages/linux/v1.11/libpmemobj/pmemobj_alloc.3 (visited on 07/20/2021).

[70] R. C. Weisner, *How Memory Allocation Affects Performance in Multithreaded Programs*. [Online]. Available: https://www.oracle.com/technical-resources/articles/it-infrastructure/dev-mem-alloc.html (visited on 07/30/2021).

[71] W. Cai, H. Wen, H. A. Beadle, C. Kjellqvist, M. Hedayati, and M. L. Scott, "Understanding and Optimizing Persistent Memory Allocation," *arXiv:2003.06718 [cs]*, Mar. 2020. arXiv: 2003.06718 [cs]. [Online]. Available: http://arxiv.org/abs/2003.06718 (visited on 08/01/2021).

[72] A. Demeri, W.-H. Kim, R. M. Krishnan, J. Kim, M. Ismail, and C. Min, "Poseidon: Safe, Fast and Scalable Persistent Memory Allocator," in *Proceedings of the 21st International Middleware Conference*, Delft Netherlands: ACM, Dec. 2020, pp. 207–220, ISBN: 978-1-4503-8153-6. DOI: 10.1145/3423211.3425671.

[73] J. Xiao, Z. Zhang, and W. Golab, "Benchmarking Recoverable Mutex Locks," in *Proceedings of the 32nd ACM Symposium on Parallelism in Algorithms and Architectures*, Virtual Event USA: ACM, Jul. 2020, pp. 583–585, ISBN: 978-1-4503-6935-0. DOI: 10.1145/3350755.3400269.

[74] solid IT, *DB-Engines Ranking of Key-value Stores*. [Online]. Available: https://db-engines.com/en/ranking/key-value+store (visited on 08/01/2021).

[75] Intel, *Pmemkv-ycsb*, 2021. [Online]. Available: https://github.com/pmem/YCSB (visited on 07/28/2021).

[76] A. A. Malakhov, "Per-bucket concurrent rehashing algorithms," *arXiv:1509.02235 [cs]*, p. 7, Sep. 2015. arXiv: 1509.02235 [cs].

[77] T. Menjo, *Introducing PMDK into PostgreSQL*, May 2018. [Online]. Available: https://www.pgcon.org/2018/schedule/events/1154.en.html (visited on 05/22/2021).

[78] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm, "Evaluating persistent memory range indexes," *Proceedings of the VLDB Endowment*, vol. 13, no. 4, pp. 574–587, Dec. 2019, ISSN: 2150-8097. DOI: 10.14778/3372716.3372728.

[79] W. Wang and S. Diestelhorst, "Quantify the performance overheads of PMDK," in *Proceedings of the International Symposium on Memory Systems*, Alexandria Virginia USA: ACM, Oct. 2018, pp. 50–52, ISBN: 978-1-4503-6475-1. DOI: 10.1145/3240302.3240423.

[80] A. A. R. Islam and D. Dai, "Understand the overheads of storage data structures on persistent memory," in *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, San Diego California: ACM, Feb. 2020, pp. 435–436, ISBN: 978-1-4503-6818-6. DOI: 10.1145/3332466.3374509.

[81] J. Condit *et al.*, "Better I/O through byte-addressable, persistent memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles - SOSP '09*, Big Sky, Montana, USA: ACM Press, 2009, p. 133, ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629589.

[82] L. Zhang and S. Swanson, "Pangolin: A Fault-Tolerant Persistent Memory Programming Library," in *Proceedings of the 2019 USENIX Annual Technical Conference*, Renton, WA: USENIX Association, Jul. 2019, pp. 897–912, ISBN: 978-1-939133-03-8.

[83] M. Hoseinzadeh and S. Swanson, "Corundum: Statically-enforced persistent memory safety," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, Virtual USA: ACM, Apr. 2021, pp. 429–442, ISBN: 978-1-4503-8317-2. DOI: 10.1145/3445814.3446710.

[84] *Reference Cycles Can Leak Memory*. [Online]. Available: https://doc.rust-lang.org/book/ch15-06-reference-cycles.html (visited on 08/02/2021).

[85] *Memhive PostgreSQL*. [Online]. Available: https://www.memhive.io (visited on 07/03/2021).

[86] R. Pydipaty, J. George, A. K. Saha, and D. Dutta, "The Effect of Non Volatile Memory on a Distributed Storage System," in *2017 IEEE 24th International Conference on High Performance Computing Workshops (HiPCW)*, Jaipur: IEEE, Dec. 2017, pp. 11–17, ISBN: 978-1-5386-1439-6. DOI: 10.1109/HiPCW.2017.00011.

[87] X. Hao, L. Lersch, T. Wang, and I. Oukid, "PiBench online: Interactive benchmarking of persistent memory indexes," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2817–2820, Aug. 2020, ISSN: 2150-8097. DOI: 10.14778/3415478.3415483.

[88] Intel, *Intel Optane Persistent Memory 200 Series*. [Online]. Available: https://ark.intel.com/content/www/us/en/ark/products/series/203877/intel-optane-persistent-memory-200-series.html (visited on 07/09/2021).