Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

PMicroProfile: A Micro-Architecture Aware Persistent Memory Profiling Framework

---

Author: Wiebe van Breukelen (2707183)

1st supervisor: dr. ir. Animesh Trivedi
2nd reader: prof dr. ir. Alexandru Iosup

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

July 14, 2023

Abstract

Storage is indispensable in today's society. It can be found everywhere, from personal devices to large data centers, and continues to grow rapidly, projected to reach 175 zettabytes in 2025. Meanwhile, the increasing number of storage use cases highlights why there is no one-fits-all solution for storage, as different storage devices drive specific goals, such as performance or energy consumption. One emerging storage class is Persistent Memory (PMEM), which offers byte-level data access at very low latencies in the nanosecond range. Unlike HDDs and SSDs that rely on dedicated controllers, PMEM is deeply integrated into the CPU core microarchitecture, residing on the CPU's memory bus alongside the main memory.

In recent years, significant efforts have been made to incorporate PMEM support into the software stack. However, the deep integration of PMEM into the computer architecture at the micro-architectural level poses new challenges in designing performant file systems. We find that the question of how efficiently a file system uses the microarchitectural level is not explored. Therefore, in this work, we propose a microbenchmarking framework called *PMicroProfile*. The unique approach of this framework involves capturing file system-dependent access traces and replaying those traces in a file system-independent manner to analyze the interaction between the CPU and the PMEM device. Our results demonstrate that PMicroProfile can capture accurate traces, although with a significant runtime penalty. Additionally, it proves effective in identifying performance bottlenecks. All source code, including instructions for reproducing experiments, is available at `https://github.com/stonet-research/PMicroProfile`.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor, Animesh Trivedi, for his extensive guidance and feedback throughout my Master's Project and literature study. His support has truly made the past year the most rewarding phase of my Master's Program. Secondly, I would like to thank all members of the AtLarge faculty at the VU Amsterdam for their helpful feedback and the enjoyable and educational Wednesday training sessions.

Lastly, I would like to thank the COMSEC research group at ETH Zürich, in particular Kaveh Razavi and Finn de Ridder, for granting me access to their server node and for their valuable suggestions, which allowed me to conduct my experiments effectively.

Contents

List of Figures

LIST OF FIGURES

List of Tables

LIST OF TABLES

Glossary

ADR Asynchronous DRAM Refresh

AIT Address Translation Table

APIC Advanced Configuration and Power Interface

ASLR Address Space Layout Randomization

CAS Compare-and-Swap Operation

CoW Copy-on-Write

CPU Central Processing Unit

CXL Compute Express Link

DAX Direct Access

DRAM Dynamic Random-Access Memory

eADR Extended Asynchronous DRAM Refresh

FS File System

HDD Hard Disk Drive

HPC High Performance Computing

ILP Instruction-Level Parallelism

ISA Instruction Set Architecture

IVT Interrupt Vector Table

KV Key-Value

LLC Last Level Cache

LSB Least Significant Bits

LSM Log-Structured Merge

MMIO Memory-Mapped I/O

MMU Memory Management Unit

MSB Most Significant Bits

NVDIMM Non-Volatile DIMM

NVMM Non-Volatile Main Memory

OS Operating System

PD Persistence Domain

PEBS Precise Event-Based Sampling

PMEM Persistent Memory

PMU Performance Monitoring Unit

POSIX Portable Operating System Interface

PPT Persistent Page Table

PTE Page Table Entry

RPC Remote Procedure Call

RTI Return From Interrupt

SCM Storage Class Memory

SIMD Single Instruction Multiple Data

SMP Symmetric Multi-Processing

SSD Solid State Drive

TFS Trusted File System Service

TLB Translation Lookaside Buffer

UPI (Intel) Ultra Path Interconnect

VFS Virtual File System

VM Virtual Memory

VMA Virtual Memory Area

WA Write Amplification

WAL Write-Ahead Log

WoC Write-optimized Compressed Key

WPQ Write Pending Queue

ZNS Zoned Namespace

LIST OF TABLES

1

Introduction

A world without storage is unimaginable in the current human society. Storage can be found anywhere. At the level of an individual, when we preserve our most valuable life experiences through photos and videos or at a much larger scale, such as in social networks or cloud storage. Therefore, the need for more data storage is a sustained activity for the foreseeable future, expected to increase to 175 zettabytes in 2025 (1). The importance of storage systems is also reflected in the prediction that Dutch data centers and related ICT infrastructure will support at least 35% of professionals in the Netherlands alone by 2025 (2). This clearly highlights why there is a need to design performant, scalable, secure, and sustainable storage systems.

During the past decades, the foundation of modern storage was laid with storage devices such as the Hard Disk Drive (HDD) and the Solid State Drive (SSD) (3). The origins of the HDD can be traced back to the 1950s when IBM introduced it as an alternative to punch cards, offering superior performance in terms of data *throughput*: the amount of data *read* or *written* within a fixed time interval.

On the other hand, SSDs, a form of *flash storage*, store information using semiconductors instead of spinning physical disk *platters* as found in an HDD. Eliminating physical movement significantly helped reduce data access latency from milliseconds to microseconds. Moreover, SSDs are more power efficient compared to HDDs (3, 4). However, HDDs still maintain a cost advantage, since the price per gigabyte of SSDs is approximately twice that of HDDs.

These tradeoffs between cost, throughput, latency, capacity, and other factors are why there is no monopoly on a single form of storage today. Instead, this diversification of storage demands resulted in the emergence of **heterogeneous storage systems** in which different forms of storage, each having its own strong properties, complement each other to

drive differentiated goals (2). By 2026, 95% of the business worldwide is expected to adapt or use heterogeneous storage solutions in some form (5). One of the emerging devices in the space of heterogeneous systems is **Persistent Memory**, which is the research area of this work.

1.1 Positioning Persistent Memory in the Storage Hierarchy

Persistent Memory (PMEM) is a type of non-volatile memory (NVM) that offers nanosecond-level device access *latencies* (6, 7). In the context of storage, the device access *latency* refers to the time it takes to write or read an arbitrary amount of data from or to a storage device to reach the CPU (8), as depicted in Figure 1.1. As PMEM offers nanosecond-level access latencies, it has substantially lower latencies than SSDs and approaches the latency of **Dynamic Random-Access Memory** (DRAM) (9).

Another similarity to DRAM is the way PMEM is positioned in the computer architecture. Unlike HDDs and SSDs, which rely on dedicated *interfaces* like SATA or NVMe for data access, PMEM resides on the CPU's *memory bus*, as shown in Figure 1.1. This allows byte-granularity data access similar to DRAM (10). The main difference between DRAM and PMEM lies in their persistence characteristics. Where DRAM is *volatile*, meaning data is lost at shutdown, PMEM is *non-volatile*, ensuring that data remains intact even after power is turned off.

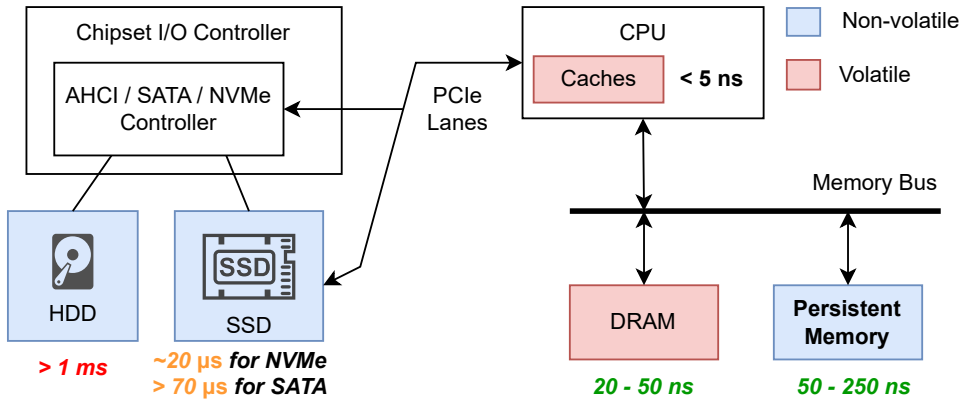


Figure 1.1: Overview of storage devices in computer architecture, including an approximation of access latencies provided by relevant papers (10, 11, 12)

This approach of integrating storage directly on the computer memory bus contradicts the view on storage over the past decades: the *two-level storage hierarchy*. This hierarchy

assumes a low-latency "primary" memory like DRAM and a secondary high-latency storage medium such as an HDD. On the contrary, PMEM establishes a **one-level storage hierarchy** in which memory and storage are closely integrated into the CPU architecture.

Over the past 10 years, multiple Persistent Memory devices have been released or are in active development (11, 13). A prominent example of real persistent memory is Intel's *Optane DC* product line, of which the first device was released in 2019 (7, 10). Although Intel discontinued this product line in July 2022 (14), many in academia and industry still consider Optane DC to be the state of the art, as indicated by the majority of studies using Optane DC to validate their claims and findings (11).

However, even with the discontinuation of Intel's Optane product line, multiple studies (13, 15, 16) anticipate that the knowledge gained from Optane-related research can be applied to a new technology released in 2019: the Compute Express Link (CXL) standard. The key feature of CXL is its ability to enable communication between CPUs, accelerators, and memories, such as GPUs and PMEM, through shared memory (17). This concept is closely aligned with the one-level storage hierarchy that PMEM establishes, highlighting that the challenges related to PMEM are worth solving, as they can be applied outside the scope of Intel's Optane DC Persistent Memory.

1.2 Persistent Memory File Systems

Over the past decades, storage devices have been considered the main performance bottleneck of file systems as the device access time could not keep up with the ever-increasing growth in CPU performance. Consequently, optimizing a file system to extract a marginal gain in performance was not considered logical; buying the latest top-of-the-line storage device makes the most sense if one would require a more performant storage solution.

Eventually, this view changed around 2006 with the end of *Dennard's scaling* (18): a prediction that the power draw of transistors remains proportional to their size. However, due to excessive heat build-up, this law broke down. As a result, single-core performance does not scale as fast as it did 30 years ago. In the meantime, storage is becoming faster and faster, approaching the latencies of fast non-volatile memories such as DRAM. Due to these factors, the **performance gap** between storage, CPU, and main memory **shrinks** (11, 19, 20, 21). This led us to **reconsider the conventional approach** in file system design. In this new view, the performance bottleneck shifts from the device (Persistent Memory) to the software stack that runs on the CPU, resulting in new opportunities for more efficient Persistent Memory file system designs.

In the background section (chapter 2), we show that these new PMEM file system designs can be categorized into three trends over time. These trends are summarized below and visualized in Figure 1.2, which illustrates the areas in the software stack to which each trend relates.

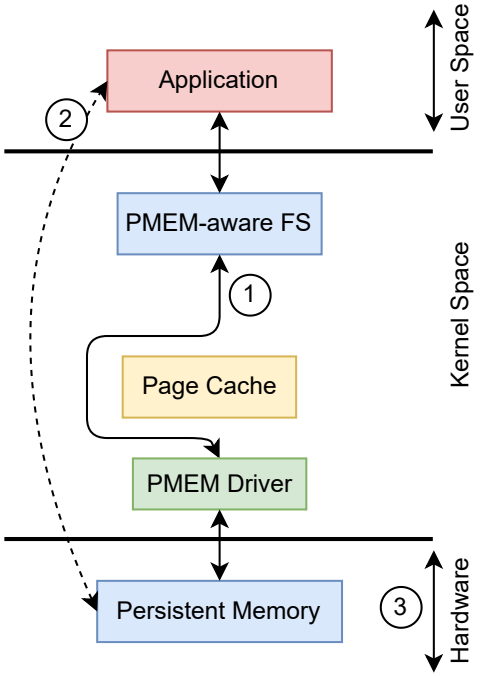


Figure 1.2: Trends Persistent Memory file systems within software storage stack

1. **Adaption of Existing File Systems.** Our literature study on PMEM file systems (11) showed that the first proof of concept PMEM-aware file system, BPFS (22), released in 2009 adapted state-of-the-art HDD and SSD optimized file systems, such as `ext4`, to support PMEM. However, with the announcement of the first real PMEM device, Intel Optane DCPMM, around 2017, it became apparent that these file systems could not keep up with Optane's low access latency due to kernel overhead, particularly from the Operating System (OS) *page cache* (23, 24). The page cache is a specialized DRAM-backed cache that stores frequently accessed data from storage media, such as an HDD (22). The deployment of a page cache is justified when data access latency exceeds the time spent inside the OS software stack (25). However, the low latencies of PMEM make this page cache redundant. Consequently, the entire design of file systems had to be reconsidered.

2. **Offloading File System to User Space.** To further reduce the overhead of the software stack, especially in the kernel and its file systems, a new trend emerged: *hybrid*

file systems. The distinct property of these file systems is that certain operations, such as reading a file, can be offloaded to user space (19, 21, 24, 26). This means that applications can directly access data without the involvement of the kernel, resulting in improved latencies and throughput.

3. **Hardware-Accelerated File Systems.** The third trend, known as *hardware-accelerated* PMEM file systems, emerged in 2022. These file systems push to reduce software overhead even further by *offloading* parts of the file system to the CPU microarchitecture. For example, the UFS (27), ctFS (23), and DaxVM (28) file systems offload part of the file system to the file system CPU's Memory Management Unit (MMU). For instance, the authors of ctFS observed that a file system's *inode tree*, which stores metadata about all files or directories stored in the file system, is constructed similarly as a *page table*, a tree-like structure the MMU uses to manage memory. An illustration of a simple inode tree is shown in Figure 1.3. This tree consists of *inodes*, which are metadata objects that store information such as file permission and file location on disk. These inodes are linked together to form a tree that the file system can traverse to retrieve information about a specific file or directory. Normally, this traversal process is performed in software. However, ctFS introduces a methodology to relocate the inode tree structure inside the MMU page table. As a result, the traversal is done transparently in hardware, reducing software overhead.

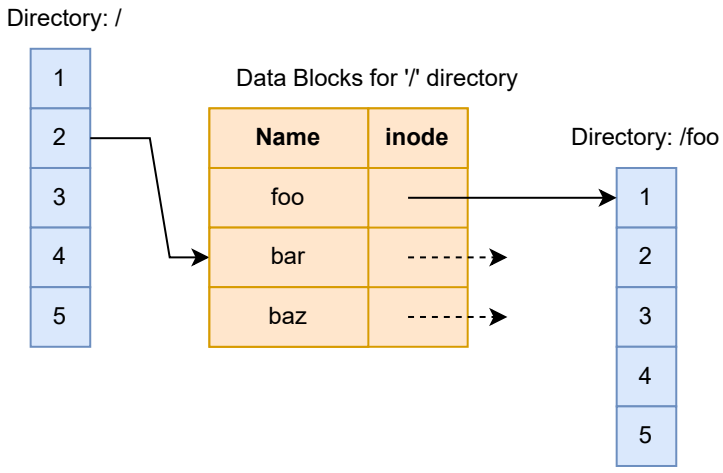


Figure 1.3: Inode tree structure

These three trends illustrate a gradual shift from performance engineering primarily at the file system level to optimizations very close to the hardware, known as **microarchitectural optimizations**. Due to this shift, we hypothesize that more performance

gains can be achieved by focusing on **how** the micro-architectural **access patterns** of PMEM file systems **affect performance attributes** such as latency and throughput.

1.3 Micro-Optimizations for Persistent Memory File Systems: an Opportunity

In traditional PMEM file systems, the primary focus for improving performance has mainly been at the software level, specifically at the file system and the kernel. These areas have shown the greatest potential to improve performance in the past (20, 21, 25, 29). However, with the emergence of memory hardware acceleration in PMEM file systems (23, 27, 28), we hypothesize that micro-architectural events become relevant factors to consider in the search for performance optimizations.

Applying micro-optimizations to enhance performance has already found its use in other storage-related systems. For example, significant research has been done on the microarchitectural behavior of in-memory database systems, exploring how these systems can leverage micro-architectural features such as instruction and data caches to achieve better performance (30, 31, 32). Another example is the use of Data Processing Units (DPUs), such as the *Nvidia Bluefield* (33), to offload file system operations to hardware (34, 35).

However, we find that the question of how to evaluate the performance of Persistent Memory file systems at a micro-architectural level has not been thoroughly explored. Therefore, in this study, we propose a framework called **PMicroProfile**. PMicroProfile aims to facilitate the identification of microarchitectural degrading performance events by capturing the PMEM access patterns of PMEM file systems and examining their interactions with the CPU microarchitecture. The following section will outline the relevant research questions and the research methodology.

1.4 Research Questions and Methodology

Using this problem statement, we can define the following research question: **How can one design a framework to evaluate the microarchitecture behavior of Persistent Memory file systems to improve performance?**

To clarify, when we refer to performance, we are specifically measuring the latency and throughput of file input/output (I/O) achieved by an application in the user space. By answering this research question, we help the systems research community and future

PMEM file system designers better evaluate the performance of their designs to extract the most performance from the hardware.

We answer this research question by answering multiple sub-questions. For each sub-question, we employ one or more of the following methodologies:

- **(Methodology M1)** Quantitative research (statistical modeling, simulations, comprehensive surveys) (36, 37);
- **(M2)** Design, abstraction, prototyping (38, 39, 40);
- **(M3)** Experimental research, designing appropriate micro- and workload-level benchmarks, quantifying a running system prototype (41, 42, 43);
- **(M4)** Open science, open source software, community building, peer-reviewed scientific publications, reproducible experiments (44, 45, 46, 47).

Sub-questions.

RQ1. **What are the performance-related idiosyncrasies of Persistent Memory at the CPU micro-architectural level?**

To answer this question, we must consider how Persistent Memory is integrated into the CPU microarchitecture. We will conduct a literature study to obtain these results **(M1)**. For our research, we have chosen to focus on Intel Optane DC Persistent Memory (DCPMM) due to its widespread adaption rate in industry and academia (48). This decision is strengthened by our literature survey on Persistent Memory File Systems (11), revealing that most of the existing literature focuses on Intel Optane DCPMM. These two factors ensure that the survey search space is set up as broadly as possible.

Using the findings of this literature study, we establish a list of Intel Optane DCPMM performance-related idiosyncrasies, which serve two purposes. First, it enables us to define a set of guidelines/requirements that are deemed essential for achieving maximum PMEM performance. Second, it helps to define the scope of our work in designing the *PMicroProfile* benchmarking framework.

RQ2. **How to design a tool that can trace the access patterns of the PMEM file system at a microarchitectural level?**

Before evaluating the micro-architecture performance of a file system based on the device idiosyncrasies found in sub-question **RQ1**, we need to understand how a file

system interacts with Persistent Memory at the CPU's micro-architectural level. Therefore, we propose the tracing tool ***pmemtrace***.

One key aspect to consider when designing such a tool is *isolation*, that is, ensuring that only PMEM-relevant events are traced while excluding any additional non-architectural sources of overhead introduced by the kernel and file system. Therefore, in addition to the actual implementation of a tracing tool, answering this question will also involve determining which events should be logged.

This is an experimental research question, and the deliverable is a functional prototype **(M2, M4)**.

RQ3. **How to design a tool that quantifies the performance of PMEM file systems using its access patterns?**

Building upon the results of the previous two sub-questions (RQ1, RQ2), we design another tool, ***pmemanalyze***, which allows one to *replay* captured access traces to provide insight into the performance bottlenecks of PMEM file systems at the micro-architectural level. This is done by defining appropriate performance metrics, according to the findings of sub-question **RQ1**.

Again, this is an experimental research question, and a running prototype is the deliverable **(M2, M4)**.

RQ4. **Can the PMicroProfile framework pinpoint microarchitectural performance degrading events to define new file system micro-optimizations?**

In an experimental evaluation **(M3)**, we evaluate the effectiveness of the PMicroProfile framework in identifying performance bottlenecks of two PMEM file systems: Ext4-DAX and SplitFS. Additionally, we verify the validity of our work. These insights form the basis of new micro-optimizations.

1.5 Thesis Contributions

Contributions in this thesis can be classified as (1) conceptual, (2) experimental, and artifacts (3). Specific contributions are as follows:

- (**Conceptual**) We conduct a literature study to determine the performance idiosyncrasies of Intel Optane DC Persistent Memory at the micro-architectural level;

- (**Artifact**) Design and implementation of *pmemtrace*: a tool that can be used to extract PMEM access traces at machine instruction-level granularity;
- (**Artifact**) Design and implementation of *pmemanalyze* tool, which evaluates PMEM file system performance using relevant micro-architectural performance metrics by replaying previously *pmemtrace*' captured traces.
- (**Experimental**) We perform a thorough performance analysis of workloads executed on the ext4-DAX (49) and SplitFS (19) file systems using the *pmemtrace* and *pmemanalyze* tools. The objective is to pinpoint variations in access patterns and performance trends and propose optimizations;

1.6 Societal Relevance

In this study, we propose PMicroProfile, a framework for conducting microarchitecture-aware performance analysis of Persistent Memory file systems. By addressing the research questions posed in this thesis, we aim to contribute to the field of High-Performance Computing (HPC) (50). We believe that our work will be particularly beneficial for HPC applications that require close proximity between data and computation, such as climate simulations where data, like the temperature at different heights across the globe, is stored on a three-dimensional grid. Previous studies (51, 52, 53) have demonstrated that data-intensive applications like these can greatly benefit from utilizing Persistent Memory. Ultimately, our work aims to help applications leverage the underlying hardware more efficiently, bringing **performance benefits** and **potentially reduced energy consumption**. This objective aligns with one of the primary challenges of storage systems outlined in the Dutch CompSys Manifesto (38), which is the integration of storage within the unified *storage-memory-compute* and *computation-in-memory* model, enabling improved cost, data, and performance efficiency for both current and future applications.

1.7 Thesis Outline

This thesis is organized as follows (refer to Figure 1.4). First, in the Background (Chapter 2), we provide an overview of how Persistent Memory is integrated into the hardware architecture and discuss how file systems achieve crash consistency. We also explore the current trends in the research field, together with related work. Readers already familiar with the principles of how Persistent Memory is integrated into the hardware architecture

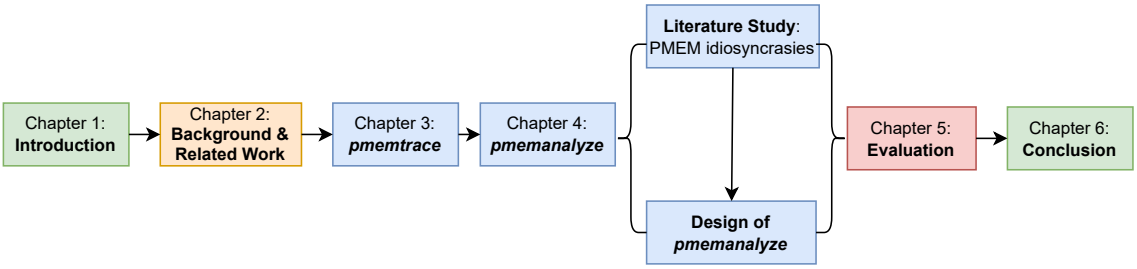


Figure 1.4: Thesis Reading Structure

can skip this chapter. Subsequently, Chapter 3 presents the design of *pmemtrace*, a tool developed to collect access traces at the granularity of machine instructions.

Chapter 4 is divided into two parts. First, we conduct a comprehensive literature study on the microarchitectural performance-related peculiarities of Intel Optane DC Persistent Memory. Second, the results of this study form the basis of the design of *pmemanalyze*, our analysis tool.

Chapter 5 presents the experimental evaluation. Specially, we use *pmemtrace* and *pmemanalyze* together to analyze the performance of two file systems, Ext4-DAX (49) and SplitFS (19).

We conclude this thesis by answering the research questions formulated in the introduction (RQ1, RQ2, RQ3, RQ4). Artifacts for reproducing a working environment, including the *pmemtrace* and *pmemanalyze* tools and the experimental setup, are provided in Chapter 7.

Plagiarism Detection

I confirm that this thesis work is my own work, is not copied from any other source (person, the Internet, or machine), and has not been submitted elsewhere for assessment.

Please note that the Background section (chapter 2) includes some small text sections and figures that are taken from a survey I conducted on Persistent Memory file systems (11), which is my original work.

2

Background and Related Work

This section discusses the concepts necessary to understand the design choices made in this thesis. In section 2.1, we discuss how the characteristics of PMEM changed the view of how storage is integrated at the hardware level. Second, section 2.2 elaborates on how the hardware properties of PMEM influenced the design of file systems, explicitly focusing on current trends in the research field.

2.1 Positioning Persistent Memory in the Hardware Architecture

This section details the position of PMEM within the computer hardware architecture. Specifically, we focus on the interaction between PMEM and other hardware components. First, we explain how a CPU retrieves data from PMEM (subsection 2.1.1), using the steps depicted in Figure 2.1 as a guideline. Next, we study the implications of caching in a PMEM context (subsection 2.1.2). Finally, we discuss how a *Persistent Domain* ensures data integrity after a power failure.

2.1.1 Accessing Persistent Memory

As mentioned in the Introduction (chapter 1), persistent memories are byte-addressable. As a result, a CPU can issue read/write requests using load and store instructions similar to accessing DRAM over the memory bus (see Figure 2.1). When the CPU issues a read request, it requests memory at a particular *virtual memory address*. Depending on the current state of the cache, the data could be contained in the cache (a *cache hit*, see **Step 1** in Figure 2.1) and, therefore, immediately transferred to the CPU, or the data must be retrieved from the PMEM device (a *cache miss*). In this subsection, we

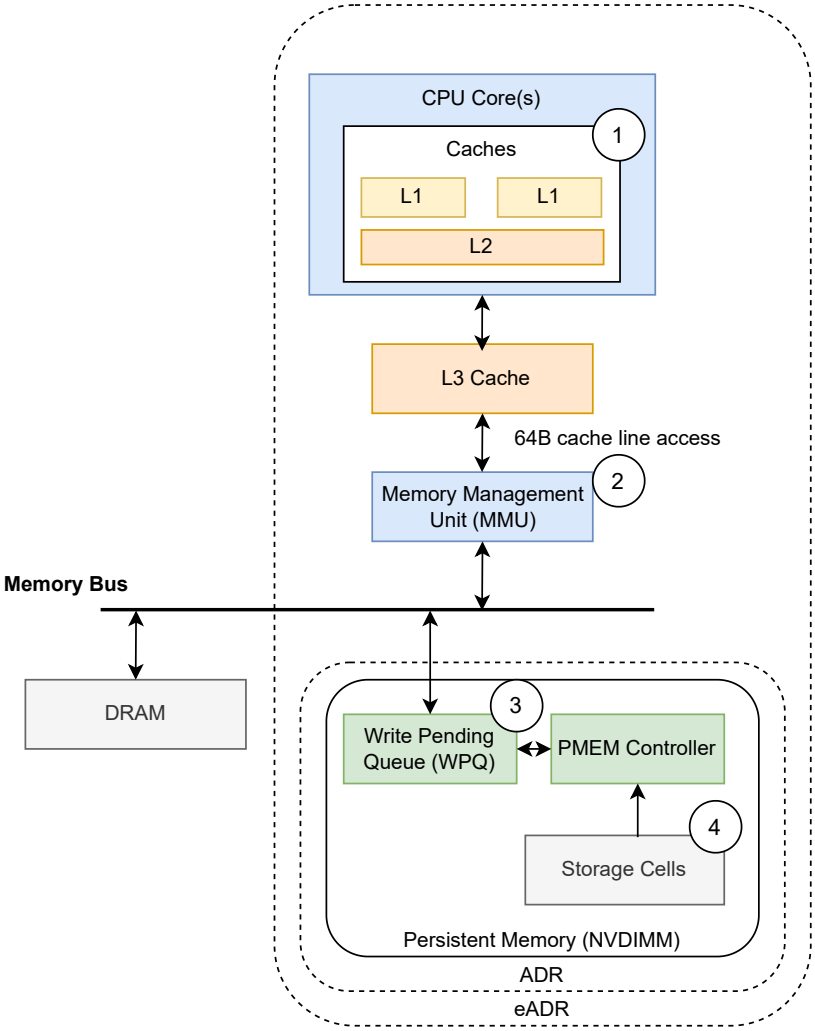


Figure 2.1: Position of Persistent Memory within the computer hardware architecture. Four steps illustrate the path taken to access data on Persistent Memory.

discuss what happens in the case of a cache miss. We discuss the case of a cache hit later in subsection 2.1.2.

What happens in case of cache miss? To understand how a cache miss is handled internally, we first need to consider how the Memory Management Unit (MMU) handles *virtual memory*. Operating systems use virtual memory to provide isolation, security, and efficient memory fragmentation between applications and the Operating System kernel (25). Applications perform read and write operations by referencing an address in this virtual memory region. In hardware, the CPU's Memory Management Unit converts this *virtual memory address* to a *physical address*, which corresponds to the actual physical

location within main memory or a memory-mapped device, such as PMEM [1] (**Step 2**). Note that this translation is performed according to a data structure that the operating system has initialized beforehand: a *page table*.

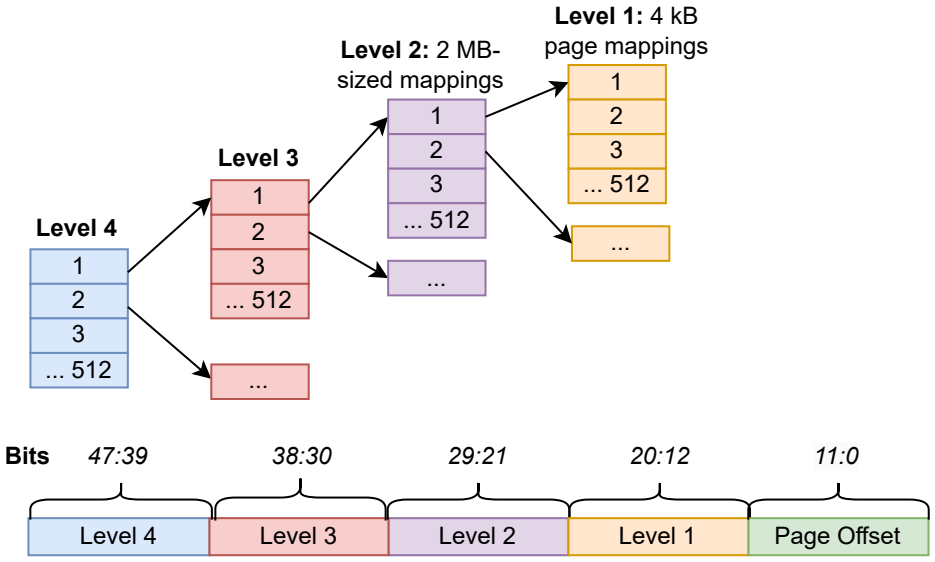


Figure 2.2: Structure of a four-level page table

In the case of the `x86-64` architecture, a page table is a hierarchical structure that maps the 48 Most Significant Bits (MSB) of the virtual address to a physical address. Note that 12 Least Significant Bits (LSB) represent the position within a *page*, the smallest unit at which virtual memory is mapped. Therefore, these 12 LSB are not part of the translation process.

The structure of a page table is quite complex; therefore, we refer to the Linux kernel documentation (54), which we briefly summarize. As shown in Figure 2.2, a page table is essentially a tree structure that consists of four levels [2], where each level can accommodate for 512 mappings. At level 1, each mapping covers one 4 kB page, so in total 2 MiB of physical memory as $4\text{kiB} * 512 = 2\text{MiB}$, that is, 12 bits of a virtual address. Likewise, at Level 2, each mapping covers one 2 MB huge page (56). A key observation is that the associated mappings become coarse-grained when the level increases, mapping larger memory regions.

As PMEM is set up as *Memory-Mapped I/O* (MMIO), in which the main memory and

[1] Also known as Memory-Mapped I/O (MMIO)

[2] Since the Intel Ice Lake microarchitecture (released in 2017), the CPU supports five-level page tables, extending the size of a virtual address to 57 bits, resulting in 128-petabyte addressable space (55).

I/O devices, share the same *physical* address space, I/O operations are performed by writing/reading to the respective device regions within this address space.

When the MMU has converted the virtual address to the respective physical address, the request reaches the Persistent Memory device, which places the request in an internal buffer queue, the *Write Pending Queue* (WPQ), see **Step 3** in Figure 2.1. When the PMEM device is ready to process a new request, the request is forwarded to the PMEM controller. This controller includes an *Address Translation Table* (AIT), which performs yet another address translation; it maps physical addresses to device addresses (**Step 4**) according to a *wear leveling* strategy. Wear leveling prolongs the lifetime of PMEM by spreading the *Program and Erase* (P/E) cycles over the memory cells. This is important because NVM cells can only endure a finite number of write cycles (7). The data is written to the NVM cells in fixed-sized chunks (also known as an *XPLine*), in the case of Intel Optane DC PMEM, 256 bytes. Consequently, writing less than 256 bytes will result in *write amplification*: the difference between the actual amount written and the amount of data intended to be written (57).

2.1.2 The Implications of Caching

Modern CPUs use a hierarchical caching architecture consisting of *levels* to reduce access latency to main memory (58), as shown in Figure 2.1. For example, for the `x86-64` architecture, the first cache level (L1) offers lower latencies than a level 2 (L2) or level 3 (L3) cache. On the contrary, a level 3 cache offers greater capacity than an L2 cache. For instance, Intel's *13900K Raptor Lake* CPU released in 2022 has 2.1 MB of L1 cache, 32 MB of L2 cache, and 36 MB L3 cache (55). In contrast to the L1 and L2 caches, the L3 cache is shared between the cores, as shown in Figure 2.1. Regarding latency, an L1 cache line fetch takes approximately 5 CPU cycles, while an L3 cache fetch costs $\sim$ 41 cycles [1] (59).

In addition to latency and capacity, two additional properties are fundamental when considering caches (25, 29, 60, 61): *access granularity* and *cache coherency protocols*.

First of all, in contrast to byte-addressable memories such as DRAM or PMEM, the cache is not byte-addressable. In the case of the `x86-64` architecture, the cache stores data in fixed fragments of 64 bytes, also known as a *cache line*. As this is the smallest unit of access, the data is fetched or written in 64 byte chunks.

Second, in shared memory systems, where memory is shared between multiple cores, the caches must confirm to a *cache coherence protocol*. This protocol ensures that each

[1] In case of a cache hit. If a cache miss occurs, the access latency increases as data must be fetched from main memory (58).

core has access to the most up-to-date version of a memory location contained in the cache (62). However, adhering to such a protocol comes at the price of performance, requiring additional CPU cycles to perform the necessary consistency checks. To amortize this impact, the CPUs may reorder writes, thereby diverting from the order the Operating System or application has issued operations (63). Although this reordering might benefit DRAM performance, it has significant implications for PMEM.

Consider a scenario in which a file system performs multiple writes to PMEM: writes X, Y, and Z. Note that, due to the cache coherency protocol, the write may be reordered. For this scenario, suppose that the issue order of the writes to PMEM is: $Y \rightarrow Z \rightarrow X$. Now, suppose that after completing write Z, the system crashes. Write X is still contained in the cache and is irreversibly lost due to the system crash. This situation presents two implications. First, the file system may incorrectly assume that the writes X, Y, and Z were successfully written to PMEM when, in reality, only the writes Y and Z reached PMEM. Second, the order in which the writes X and Y are committed is reversed, causing an inconsistency between the file system transaction log and the actual data on the PMEM device.

In summary, write reordering may be beneficial to DRAM performance; however, it poses a threat to PMEM transactions (23, 25, 26, 64). First, the order of data being written to PMEM may differ from the user's intentions. Second, in case of a system crash, in-flight data could still reside in the cache, resulting in unrecoverable data loss (60).

Fortunately, both issues are addressed in software and hardware through *cache flushing/memory fencing* and a *Persistance Domain*, respectively. These concepts are introduced in the following subsection.

2.1.3 Addressing the Issues of Caching

The issues arising from caching are addressed in both hardware and software. We start by considering the hardware mechanism that guarantees write-back of uncommitted in-flight data, the *Persistance Domain* (PD). Afterward, we discuss how fencing solves the issue of write reordering.

Persistance Domain. A *Persistence Domain* (PD) is a hardware region within a computer system in which it is guaranteed that the data persist in the event of unexpected failure, such as a power outage (65). Thus, if PMEM data is within the boundaries of this region, the data will always be written back to the device. In the context of PMEM, there are two types of persistence domains: *Asynchronous DRAM Refresh* (ADR), which must

be present, and the enhanced version called *Asynchronous DRAM Refresh* (eADR), which is optional. The boundaries of both ADR and eADR are depicted in Figure 2.1. ADR guarantees data persistence within the PMEM device, while eADR extends this guarantee to CPU caches (66).

The hardware implementation of a PD is very similar to an Uninterruptible Power Supply (UPS), but then on a much smaller scale. During regular operation, the CPU and the PMEM must always ensure that enough energy is available to write back changes to the flash cells, established by charging dedicated capacitors within the CPU and PMEM device. In the event of a power failure, the remaining energy stored in this capacitor is used to commit changes to PMEM (67).

Selective Cache-Line Flushing. As discussed in subsection 2.1.2, cache coherency protocols could void the order in which data is written to PMEM, potentially leading to data inconsistencies. Therefore, we need techniques to enforce this ordering at the cache level.

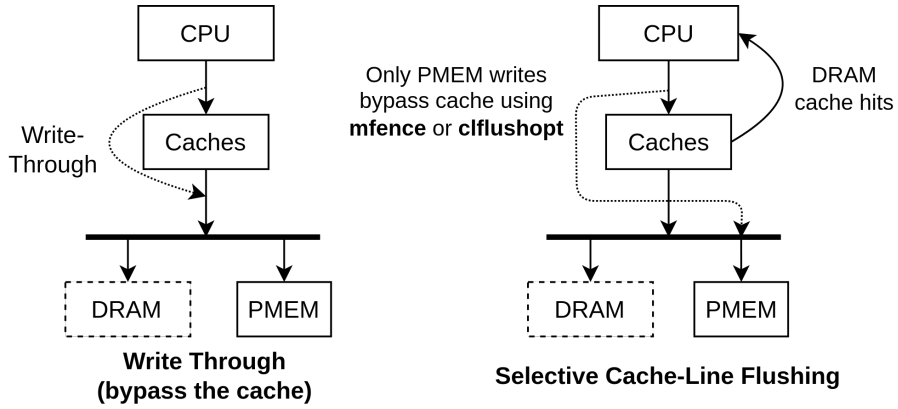


Figure 2.3: Cache bypass techniques to enforce strict write ordering

According to our literature study on PMEM file systems (11), there are two methodologies to perform write ordering:

1. The first option is to enforce ordering using **memory fences**. We use a memory fence instruction, for example `mfence`, to ensure that all load and store instructions issued before `mfence` are serialized in the order they were issued, as shown in Figure 2.3. A side effect is that the performance of other applications may degrade as its working set is (partly) evicted from the cache (55, 60).

2. Alternatively, we can perform a **fine-grained cache flush**. We keep track of the cache lines used and only flush those that contain PMEM data in flight so that DRAM reads and writes are not affected (see Figure 2.3). Intel supports this selective flushing with the `clflushopt` and `clwb` instructions with the release of the Persistent Memory Instruction Set in 2014 (68).

We use `clflushopt` and `clwb` in our work when applicable, as this bypass technique is widely considered state of the art in recent PMEM file systems (11).

2.2 Impact Persistent Memory Idiosyncrasies on File System Design

We already mentioned that the idiosyncrasies of PMEM, low-latency byte-addressable storage, resulted in multiple implications at both the hardware and software levels. At the hardware level, we discussed the impact of caching and the need for a Persistence Domain. In this section, we elaborate on how PMEM caused a change in the way file systems should be designed. First, we discuss the implications of current state-of-the-art file systems implemented in the Linux kernel. Afterward, we consider how file systems have changed, which we distill into three research trends.

2.2.1 The Overhead of Existing Linux File Systems

Applications request file system services using *system calls*. Inside the Linux kernel, these system calls are routed to the Virtual File System (VFS). The VFS defines a generic interface that is independent of the actual file system in use, for example, *ext4* or the swap space (see Figure 2.4) (69). By using this abstraction, applications are made agnostic of the file system in use, improving portability.

When the VFS receives a file I/O request, it must first determine to which file system the request should be forwarded. This is done through a *path traversal*, which recursively traverses the file path string, e.g., `"/for/bar.txt"` to look up the corresponding file *metadata*, performs the necessary sanity checks (e.g., checking for sufficient permissions) and finally performs the request action.

When using a block storage device, such as an HDD, the access latency may be orders of magnitude higher than the imposed file system latency. Therefore, the Linux kernel Virtual File System implements a *page cache* to decrease the performance impact of slow

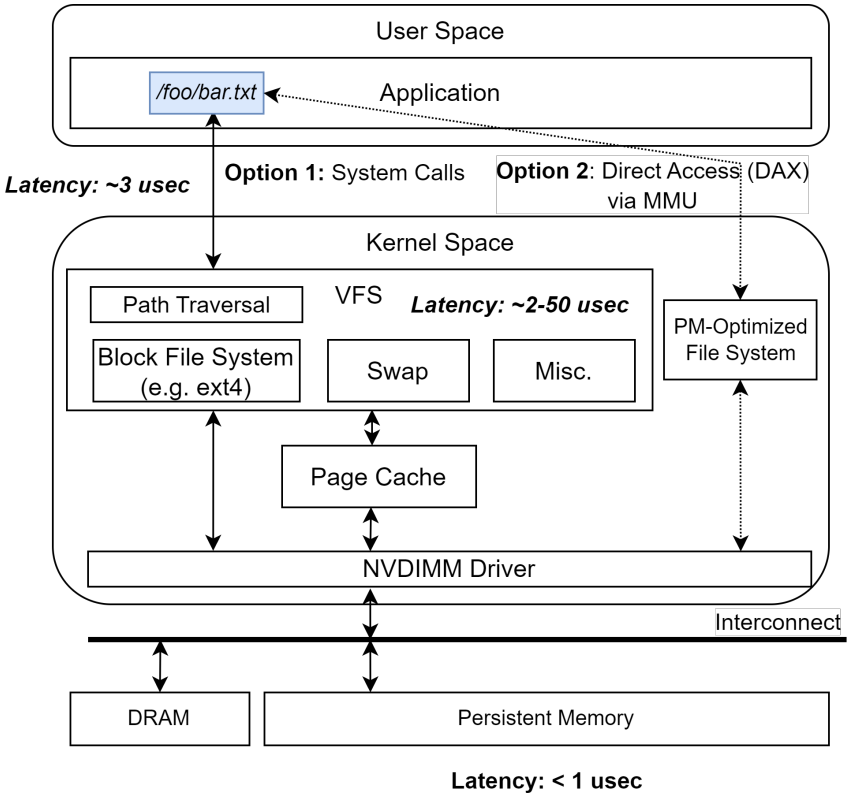


Figure 2.4: Software Architecture Persistent Memory

device latencies. As a result, file data may reside (partly) on the storage device or in the *page cache*:

(1) The file is (partially) located in the page cache. In this case, the requested data can be returned immediately without fetching from the device.

(2) The requested file blocks are not in the cache. A request is sent to the corresponding device driver, which sends the actual command to the storage device.

The use of a page cache in the case of PMEM is controversial, as the latency of the VFS and its page cache is higher than the access latency of PMEM (22, 23, 24, 25), as seen in Figure 2.4. This gives rise to alternative file system designs, which we discuss in the following subsection.

2.2.2 The Current Trends in the Research Field

In order to illustrate current research developments on file system design, we briefly discuss the positioning of existing PMEM file systems using Figure 2.5 as a guideline [1]. An in-

[1] about PMEM file systems

depth discussion of the implementation and design choices of these file systems can be found in our literature study (11).

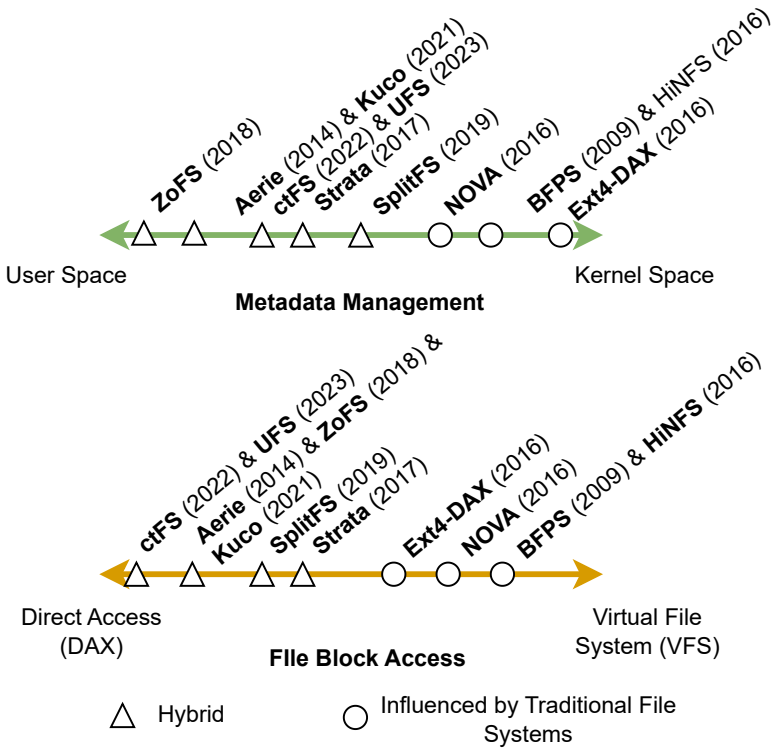


Figure 2.5: Persistent Memory file systems positioning: metadata management and file block access. The triangle (△) and circle (○) symbols depict whether the file system is using a split/hybrid architecture, in which the file system is partitioned over the kernel and user space, or uses a *monolitic* design in which it is integrated into the Virtual File System.

Note that Figure 2.5 displays two scales. The first scale, *metadata management*, shows whether the file system manages metadata in the kernel or user space. The second scale, *file block access*, shows the method of data access: Direct Access (DAX) or the Virtual File System (VFS). Please note that we will discuss the principles behind DAX in more detail shortly.

The first PMEM file systems started to emerge around 2009. These file systems adapt the VFS and its file systems to support PMEM, as shown in Figure 2.6. As a result, most of these file systems are POSIX compliant, meaning that existing applications have access to PMEM through system calls such as `read()` or `write()`. An example is ext4-DAX, which avoids expensive data copying to/from the page cache by performing zero-copy reads and writes directly to the storage device within existing ext4 file systems (70). Although these file systems offer good backward compatibility, they still suffer performance overhead due

to the VFS infrastructure (and its data structures) and the cost of constantly trapping into the kernel each time an application wants to perform a file operation (19, 26).

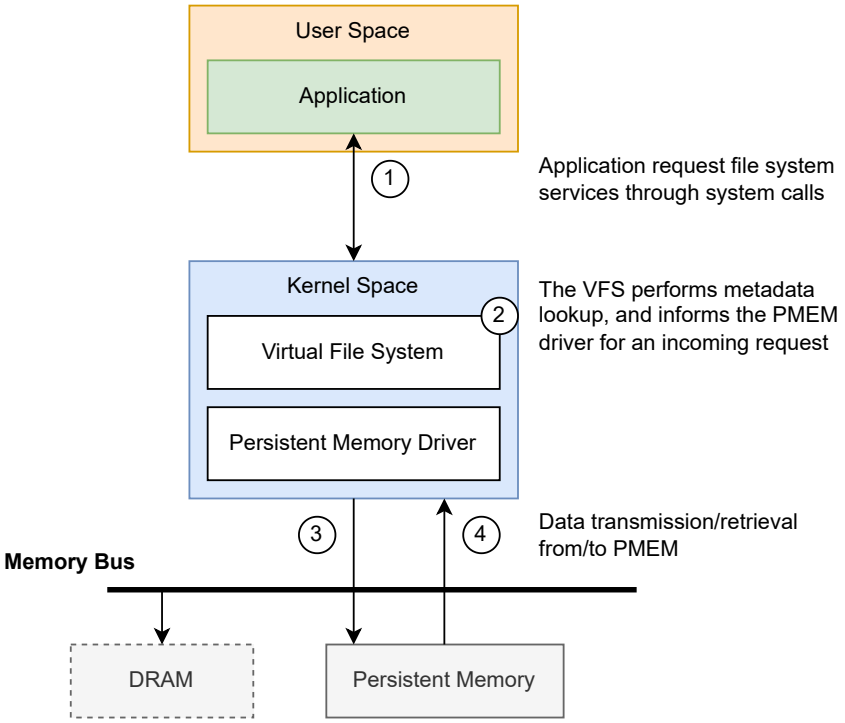


Figure 2.6: Pre-2013 integration of Persistent Memory file systems in the Linux storage stack

To address these limitations, both academia and industry have re-evaluated the design of high-performance file systems. This has resulted in three distinct trends in file system design. In the following sections, we will discuss these trends in more detail.

Trend 1: Integration in Virtual Memory using Direct Access (DAX). Direct Access (DAX) allows for a direct mapping of PMEM Memory-Mapped I/O (MMIO) pages within the user space, as shown in Figure 2.4 (70). These pages together form a so-called *Virtual Memory Area* (VMA). Using DAX has potential performance benefits as the number of kernel traps is reduced. Additionally, as we will discuss shortly, it allows for the implementation of non-POSIX file system interfaces, as file systems are not bound by the set of available kernel system calls.

To illustrate how DAX operations work, we elaborate on the three steps illustrated in Figure 2.7. (1) When an application requests file access, it notifies the PMEM file system, which subsequently performs a DAX request by notifying the kernel. (2) Following this, the kernel performs the necessary sanity checks and returns a pointer to a particular

region of virtual memory (a VMA) that has been mapped within the application's address space. (3) At this stage, an application can directly access the corresponding PMEM stored data without the involvement of the kernel or file system.

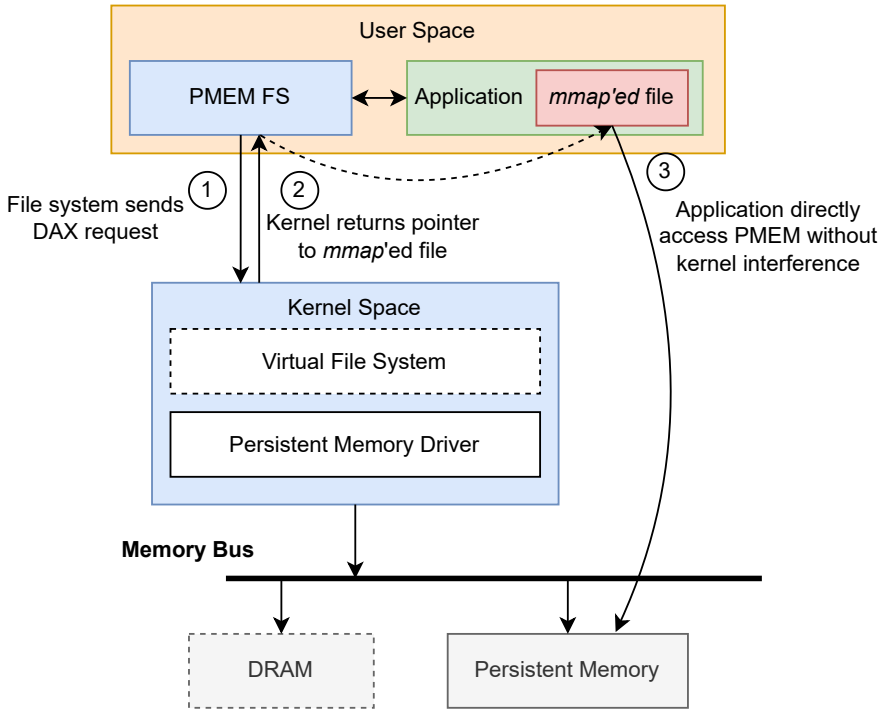


Figure 2.7: Persistent Memory file systems using Direct Access (DAX)

Trend 2: Offloading File System to User Space. A trend that extends the idea of DAX is a user-space offloaded file system, also known as a *hybrid file system* (19). The distinct feature of a hybrid file system is that the *data path*, operations that involve only the movement of file data, are offloaded to user space. On the other hand, operations related to file metadata, known as the *control path*, require synchronization in the case of multi-tenancy. Therefore, these operations are handled separately by an in-kernel library. Such a design can also be seen in other areas of computer science, for instance, offloading network packet processing to user space using DPDK.

Using this hybrid design, the number of kernel traps can be significantly reduced (19, 24). For example, when an application tries to write repetitively to the same location within a file, it only has to ask permission from the kernel library once. Other read-and-write operations that address this file can be handled entirely in the user space.

Trend 3: Hardware-Accelerated File Systems. Traditional file systems use separate data structures, known as file metadata, to map file blocks to the actual location on the storage device. As Trends 1 and 2 discussed, this conversion is performed *either* in the Operating System kernel or user space.

However, the authors of the ctFS (23), UFS (27), and DaxVM (28) file systems identified another novel approach. Instead of performing the translation in software, they offload the translation hardware, the CPU's Memory Management Unit (MMU). Now, the MMU translates a file block offset to a corresponding physical location within PMEM. The novelty of this approach lies within the idea that we can now leverage both hardware and software optimizations to accelerate file system performance.

As mentioned earlier in the Introduction (chapter 1), this thesis can be placed under the trend of hardware-accelerated file systems, as it aims to improve the latency and throughput of such file systems.

2.3 Intel Hardware Performance Counters

In this work, we will use Intel's hardware performance counters to perform performance evaluation. In the case of Intel processors, performance events can be categorized as *architectural* or *non-architectural* (55). Architectural events are micro-architecture specific, meaning they apply only to a particular CPU architecture, in this case, Cascade Lake. These events are measured per core, which means that each core implements the same performance parameters, such as the number of retired instructions.

On the other hand, non-architectural events exceed the limits of the CPU-specific microarchitecture and, therefore, can be used in a wider range of architectures. These events, also known as *uncore* events, occur at the level of the CPU package (71). Examples of uncore events include the Last Level Cache (LLC) shared between cores and the integrated memory controllers (iMCs), responsible for transferring data from and to memories, for example, DRAM or Intel Optane DC Persistent Memory. A limitation of uncore events is that they occur at the socket level; therefore, it is impossible to relate events directly to one CPU core. To address this, Intel introduced *offcore* events, which are events that are specific to individual cores and have not yet reached the uncore level. Figure 2.8 illustrates this distinction between on-core and uncore events.

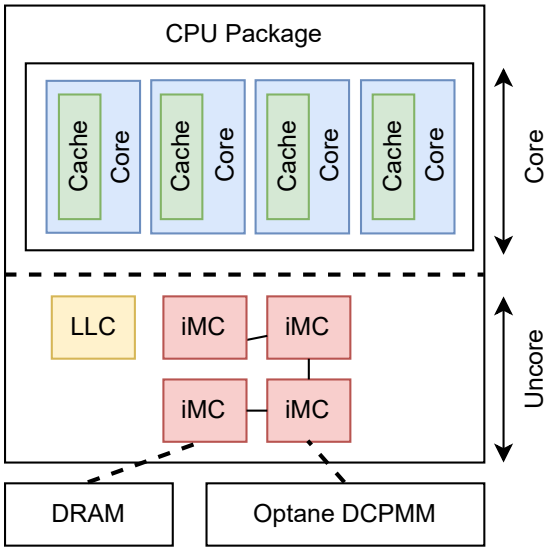


Figure 2.8: Locating Intel performance events: *core* and *uncore*

2.4 Related Work

We position this thesis at the intersection of the operating system and the hardware, and therefore, we discuss existing research that analyzes the microarchitectural characteristics of Persistent Memory.

Intel DCPMM performance idiosyncrasies. As discussed in the Introduction (chapter 1), this work aims to establish a methodology that evaluates whether the access patterns exhibited by the file system at the microarchitectural level affect performance. To our knowledge, no methodology or tool evaluates the microarchitectural behavior of PMEM file systems. This *top-down* perspective separates this work from others.

To the best of our knowledge, there are three studies (6, 9, 10) that also consider the performance of Intel DCPMM persistent memory at the microarchitectural level. Compared to our work, they tend to perform a *bottom-up* performance evaluation, which means that they explain performance behavior by crafting specialized workloads to reverse engineer PMEM internals. Each paper provides recommendations that system designers should consider when designing PMEM (file) systems. For example, van Renen et al. (6) were the first to show that system designers should strive for 256 byte accesses to minimize read and write amplification. Second, they found that using explicit cache line write back (`clwb`) instructions after PMEM reads/writes yields a positive effect on performance, as neighboring 64 byte cache lines can be merged into 256 byte requests, decreasing read and

write amplification. A similar study by Yang et al. (10) formalizes these and other findings in a set of best practices.

Another study by Xiang et al. (9) involved the use of specially crafted microbenchmarks to examine the impact of Intel Optane internal buffering and the performance impact of cache line flushing. The main takeaway of this paper is that Intel Optane's read-and-write performance exhibits very different performance characteristics. Therefore, they should be considered separately in the performance analysis.

Persistent Memory micro-benchmarking tools. Two other papers (72, 73) propose micro-benchmarking tools (PMIdioBench and PerMA-bench, respectively) that are agnostic to the actual device used, resulting in a broader set of system design recommendations independent of the underlying PMEM device. The authors of PerMA-bench make an observation that is also applicable to the motivation of this research: no standardized benchmarking framework can perform low-level performance analysis over a wide variety of workloads (the *top-down* analysis perspective). The PerMA-bench tool allows system designers and programmers to define workloads within user space using a YAML configuration file, for example, to specify the flushing methodology, number of threads, type of write operation (`mov`, AVX-256, AVX-512), etc. PMIdioBench proposes a similar benchmarking tool, although less configurable than PerMA-bench.

There is definitely common ground between PerMA-bench, PMIdioBench, and our work. All examine PMEM's microarchitectural behavior by running commodity workloads instead of synthetic, device-tailored workloads. However, both tools do not offer a methodology to evaluate the impact of PMEM file system access patterns at the microarchitectural level, as we aim to establish in this work.

Although not an academic contribution, we consider Intel's *VTune* profiling utility (74) and the `perf` tool (75) to be related work, as both are capable of collecting DCPMM performance counters. Both tools offer extensive profiling capabilities; still, they are application- and file-system agnostic, meaning the end user should decide how to interpret these performance counters, which is very difficult without prior in-depth knowledge.

2.5 Summary

In this chapter, we learned how the unique properties of persistent memory, byte-addressable low-latency nonvolatile storage, result in two implications when integrated into today's commodity server hardware. First, caching may result in irreversible data loss in the event

of an unexpected power failure. The Persistance Domain (PD) addresses this issue by defining a region in the hardware where data is guaranteed to persist during a power failure. Second, selective cache-line flushing enforces a total write order, avoiding an inconsistent state between storage and the file system.

Moreover, we discussed how the emergence of Persistent Memory led to significant changes in traditional file system design, which can be categorized into three trends: the integration of persistent memory into virtual memory, *hybrid file systems*, and finally, CPU microarchitecture-aware file systems. The latter trend is the area of research in which this work is positioned.

3

pmemtrace: A Tool to Collect Access Traces at Instruction Granularity

To assess the performance of PMEM file systems at the micro-architectural level, it is necessary to have access to representative workloads. Other studies (6, 9, 10, 72, 76, 77) that evaluate the low-level performance of PMEM use *synthetic workloads*, meaning that a workload is carefully crafted to trigger certain micro-architectural behaviour. For example, one could design a workload that generates access patterns that may exhibit many mispredictions to evaluate the effectiveness of CPU prefetching, a technique to load instructions or data into the CPU cache proactively.

However, we decided against this common approach of using synthetic workloads in this study. Instead, we focus on understanding how certain access patterns of the file system can affect performance at the lower levels of the storage hierarchy, i.e., the interaction between the CPU and PMEM. Our main interest lies in how these access patterns influence performance; therefore, our goal is to minimize potential disturbances. For example, we want to exclude any overhead imposed by the kernel and file system, as it could mask interesting events that occur at the bottom of the storage stack. For this reason, **this work presents a unique approach to collect representative workloads**. Instead of generating synthetic workloads, we develop a tracing tool, *pmemtrace*, that can capture PMEM-relevant at the architectural level (i.e., reads, writes, and cache hints) that are imposed by the file system layer on top.

For instance, when an application initiates a write operating with a 256-byte chunk of data, a PMEM-aware file system receives this request, looks up the file metadata, and determines the appropriate MMIO location where the data should be written. After writing

the data, the file system performs several cache line flushes to ensure the data is permanently stored. Meanwhile, *pmemtrace* records the relevant machine instructions, such as `movnti` and `clflush`, in a trace file. All machine instructions contained in the trace file can then be replayed in our performance analysis framework, *pmemanalyze*, whose implementation and design are discussed later in chapter 4.

First, in section 3.1, we formulate the set of design requirements. Subsequently, section 3.2 discusses the tracing methodologies explored and which methodology was ultimately selected. Then, section 3.3 elaborates on the fundamental design/implementation considerations of *pmemtrace*. We conclude this chapter with an in-depth discussion of the strengths and limitations of *pmemtrace*.

3.1 Design Requirements

In this subsection, we define the set of requirements for the PMEM trace collector *pmemtrace*. Doing so allows us to reason why specific research directions were explored, allowing an informed decision on which tracing methodology is most suitable for device-level access tracing.

Table 3.1 displays the set of design requirements. In the following two paragraphs, we will discuss the motivation behind these requirements and how they relate to solving part of the research problem statement stated in section 1.3.

ID	Requirement	Priority
RQ1	PMEM read and write operations should be collected at the smallest byte-granularity possible, for example, `movnti m32, r32`) in case of a 4 bytes write.	High
RQ2	The tracing tool must be able to capture events in program execution order	High
RQ3	The tracing tool must not taint the application functional behaviour, cause an application crash, or a kernel panic	Medium
RQ4	The tracing tool should be able to capture cache flushing events, e.g., `clwb` or `clflushopt`	Medium
RQ5	The tracing tool should be able to collect events in a multicore/Symmetric Multi-Processing (SMP) environment	Medium
RQ6	The tracing tool should implement a sampling-based data collection strategy if, at the user's discretion, runtimes become infeasible.	Low

Table 3.1: Set of requirements *pmemtrace* tracing tool, ordered by priority level

Tracing Accuracy and Validity. Tracing accuracy is an essential quality attribute to consider when designing a tracing tool (78). We define tracing accuracy as the degree to which a *tracer* can capture PMEM access patterns generated by both applications *and* the file system. In this context, we have identified three important classes of CPU instructions: device reads, writes, and associated cache flush operations.

PMEM reads and writes are self-explanatory, as these operations are taken into account when examining the interaction between file systems and PMEM devices. Ideally, the tracking of read/write operations should closely align with the CPU microarchitecture, that is, at the granularity of *x86* load and store operations (**RQ1**). This tracking should occur during program execution (**RQ2**), without causing crashes or fatal kernel panics (**RQ3**). Additionally, cache flushing operations ensure the durability of data in the event of failures (as discussed in section 2.1). Therefore, *pmemtrace* should also be able to capture relevant machine instructions, that is: `clflush`, `clflushopt`, and `clwb` (**RQ4**). Finally, *pmemtrace* should be able to log (PMEM-relevant) retired CPU instructions on multiple cores simultaneously. We hypothesize that this capability could reveal interesting access patterns, such as read/write amplification at the device (**RQ5**).

Tracing Overhead. We hypothesize that a close-to-hardware tracing approach may lead to severe performance degradation throughout the system. However, the validity of this work remains unaffected, as we prefer trace accuracy over execution run time. Therefore, we assign the related requirement, **RQ6**, a low priority.

In the event that a user of the *pmemtrace* tool finds the increased execution run time to be impracticable, we should be able to switch to a sampling-based data collection strategy, as formulated in requirement RQ6.

In summary, *pmemtrace* should precisely collect relevant events (reads, writes, and cache flushes) at the precision of CPU instructions, i.e. assembly code. Additionally, we prefer trace precision over execution run-time, which means that switching to sampling-based data collection is done at the user's discretion. In the next section, we discuss three promising tracing methodologies that can be used to collect device access traces and evaluate them according to the requirements in Table 3.1.

3.2 Tracing Methodologies

In this section, we discuss three tracing methodologies that we consider viable for the collection of CPU instruction-level access traces. The resulting insights determine the final

design and implementation of *pmemtrace*.

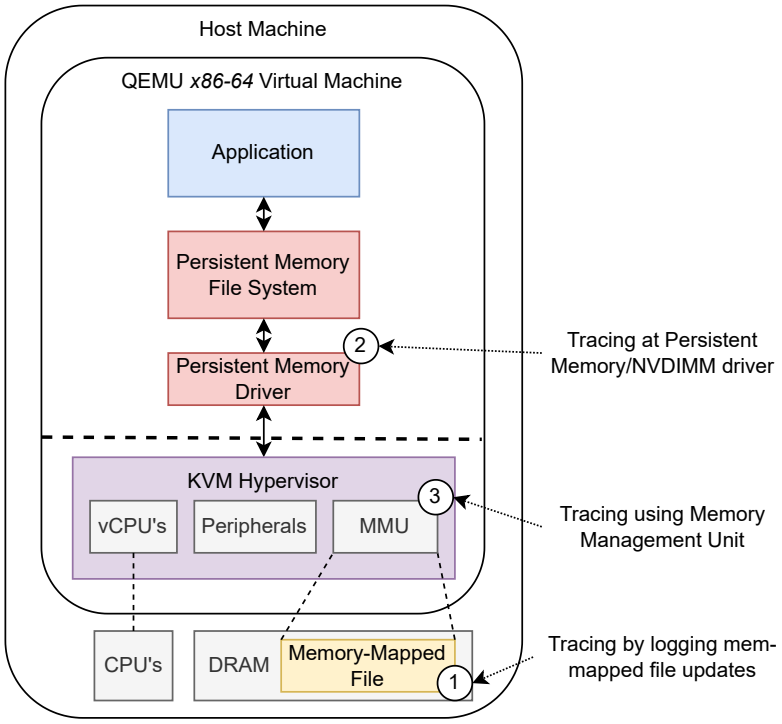


Figure 3.1: Overview of considered tracing methodologies

For rapid prototyping, we use a QEMU Virtual Machine (version `7.2.0`, 8 cores, 8 GB RAM, Intel `13900K` CPU) with 28 GiB emulated Persistent Memory, running Ubuntu 20.04, Linux kernel version `5.4.232`, together with KVM (Kernel Virtual Machine): an open-source kernel module that allows the host kernel to act as a hypervisor. An artifact on how to reproduce this setup is provided in section 7.1.

In addition to faster prototyping, the use of a virtual machine has another potential advantage. As Persistent Memory is emulated inside QEMU by means of a *memory-mapped file* (see Figure 3.1, methodology 1), we hypothesize that by adapting QEMU, the hypervisor or the PMEM driver might allow capturing the PMEM read/write/flush events, emitted by a file system inside the VM, within the host operating system. This insight drives the first methodology we evaluate: *tracing by logging memory-mapped file updates*.

Methodology 1: Tracing QEMU Memory-Mapped File Updates. Before considering the design of this tracing methodology, we first elaborate on how QEMU emulates PMEM within its software stack. QEMU emulates a PMEM device by mapping a portion of the host's main memory (DRAM) inside the Virtual Machine as a Memory-Mapped I/O

(MMIO) device, as illustrated in Figure 3.1. Requesting this DRAM region (also known as an *anonymous memory region*) is done by means of a *mmap* system call. For completeness, the corresponding C code that performs this action is located within the `mmap_activate` function (QEMU source file: `util/mmap-alloc.c`). To ensure that the VM can actually detect the emulated PMEM, QEMU inserts a new entry in *NVDIMM Firmware Interface Table* (NFIT). This entry contains multiple data fields that are defined according to the *Advanced Configuration and Power Interface* (ACPI) standard: an interface that the OS uses to detect hardware components (79). An NFIT entry contains multiple metadata fields that are used by the OS to extract the necessary configuration information, for example, the *region offset* and *region length* fields specify where the PMEM device is mapped within physical memory. The respective source code for configuring the NFIT entry inside QEMU can be found in the file `hw/acpi/nvdimm.c`, function `nvdimm_build_device_structure`.

Recall that QEMU emulates a PMEM device by inserting a DRAM-backed memory region inside the physical address space of the VM: an *anonymous memory region*. Alternatively, instead of persisting write updates to DRAM, QEMU can also be set up so that it mirrors all changes made to this anonymous memory region to a file, also known as a *memory-mapped file* [1]. We hypothesize that an accurate PMEM access trace can be obtained by logging accesses to this memory-backed file. To test whether this methodology is viable, we performed the following experiment.

1. A QEMU VM instance is configured using the QEMU configuration found in Listing 8.1. Most important to note is that both the *share* and *pmem* settings are set to "on", as this allows file write-through (80);

2. We modify the QEMU source code so that it prints the file descriptor of the host memory-backed file to the console by adding a print statement in the `mmap_activate` function:

```
1 activated_ptr = mmap(ptr, size, prot, flags | map_sync_flags,
     fd, map_offset);
2 printf("%d\n", fd); // Add this print statement
```

Listing 3.1: Changes `util/mmap-alloc.c` Source File

3. Then, the extracted file descriptor (FD), together with the Process ID (PID) of the QEMU process, are included as arguments when invoking *strace*: `strace -p PID -e`

[1] QEMU PMEM file write-though can be enabled by setting the `share` flag to 'on' (80)

`trace=file -e read=FD -e write=FD`. Strace prints all reads and writes to this file to the console.

4. Within the VM, we mount a new ext4-DAX file system and run a simple workload that repeatedly appends 128 bytes to a file while logging the output of *strace* in the host OS;

Although the capture of read and write operations worked reliably, closer examination revealed three limitations that void the high-priority requirements formulated in Table 3.1. First, we are only able to trace read/write at the block size of the host file system, in our case 4096 bytes in the case of the *ext4* architecture. Consequently, smaller (adjacent) reads and writes within a single block remain undetected (RQ1). This issue gives rise to another limitation: the ability to capture events in real time. As PMEM traffic is intercepted at block size granularity, the property of capturing reads/writes in program order (RQ2) is void. Lastly, to the best of our knowledge, *QEMU* and *strace* are unaware of cache line flushing inside the Virtual Machine; hence, those events cannot be easily captured, causing the requirement RQ4 to be void. These limitations infer that this methodology is not viable for collecting detailed access traces. Therefore, we consider an alternative methodology.

Methodology 2: Tracing Linux PMEM/NVDIMM Kernel Driver Events. Compared to Methodology 1, where PMEM events are collected *outside* the VM, this section considers a completely different methodology in which PMEM events are collected *inside* the VM. This methodology performs tracing at the device driver (refer to Figure 3.1). We start by describing the relevant software components, in this case, the positioning of the PMEM (or, in Linux terminology, NVDIMM) driver inside the Linux Kernel. Again, we use the Linux kernel (version `5.4.232`) for the experiment. Subsequently, we discuss how to collect device events within the NVDIMM driver.

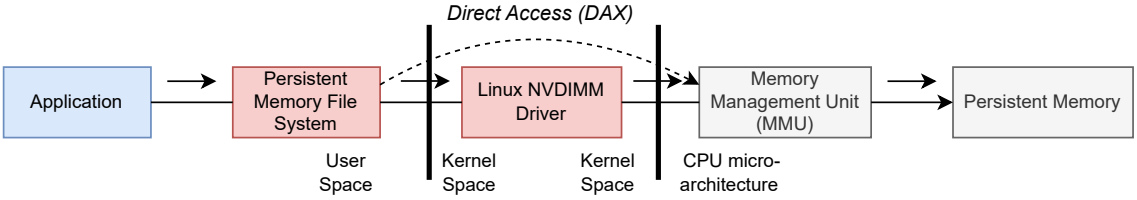


Figure 3.2: Positioning Linux NVDIMM driver in software/hardware architecture

The Linux NVDIMM driver can be best seen as the 'glue' between the file system and the actual device, as illustrated in Figure 3.2. It allows file systems (and the applications

on top) to benefit from increased device compatibility at a relatively low cost. Note that the figure is simplified; for example, the PMEM file system might be partially (or fully) located inside kernel space, as previously described in Background section 2.2.

The NVDIMM kernel module can be initialized by invoking the `pmem_attach_disk` function inside the `drivers/nvdimm/pmem.c` file. The invocation of this function is done automatically when the module registers itself in the kernel. As the NVDIMM kernel module is built on top of the existing Linux block device infrastructure (contained in the `/block` folder), the PMEM is registered as a block device, which is done by invoking the `blk_alloc_queue_node` function. Each PMEM device is assigned a unique identifier so that other in-kernel software components can detect it as such a type of device. Additionally, the NVDIMM driver initializes multiple metadata structures, of which the most important one is the mapping of PMEM MMIO physical pages [1] inside the kernel page table by calling the `devm_memremap_pages` function.

When a file system issues a read/write/flush operation, its places a new request in the block device `request_queue`. Internally, the request is forwarded to the `pmem_make_request` function inside the `drivers/nvdimm/pmem.c` file. In turn, `pmem_make_request` performs the requested action (read, write, or flush) by invoking one of the following functions: `pmem_read`, `pmem_write`, or `nvdimm_flush`. Subsequently, the requested action is performed by reading/writing or flushing the respective memory address(es) within virtual memory. Finally, the MMU ensures that this virtual memory access is converted to the actual location on the physical device so that the operation can be completed (see Figure 3.2).

To trace these events, we can capture the calls to `pmem_read`, `pmem_write`, and `nvdimm_flush` functions. One way to achieve this is by using the eBPF framework, which enables us to add kernel probes that log the function arguments whenever one of these functions is invoked. In section 8, we have provided an example source code to do this.

Although this design is promising, it still has one major limitation, namely tracing Direct Access (DAX) operations. Recall that DAX operations allow applications or file systems in user space to bypass the kernel and its drivers (Background subsection 2.2.2), as shown by the striped arrow in Figure 3.2. Consequently, a tracing tool solely based on the capture of driver events would never be able to capture DAX operations, severely affecting the ability to extract accurate traces (RQ1, RQ2). Therefore, in the next paragraph, we propose a

[1] The physical location of the device can be extracted using the ACPI NFIT table as discussed previously.

novel methodology that, in addition to classical POSIX *read()*- and *write()*-like operations, can also intercept DAX operations.

Methodology 3: Tracing using the CPU Memory Management Unit. The third and final methodology uses the CPU MMU to capture PMEM access traces. We use this design in our trace tool *pmemtrace*. For now, we only discuss the high-level design considerations of *pmemtrace*, which means that we do not go into the actual implementation details. This will be addressed in a separate section later.

As this methodology makes use of the MMU, we first need to consider what functionality the MMU brings to the table. In subsection 2.1.1, we have already discussed how an MMU enables Operating Systems to provide isolation, security, and efficient memory fragmentation. In short, the MMU converts a *virtual address* into a *physical address* by looking up the corresponding *Page Table Entry* (PTE) inside a *page table*. The structure of a PTE depends on the computer architecture and its instruction set. In the case of `x86-64` architecture, it contains multiple bitfields, for example, the 48 Most Significant Bits (MSB) of the physical address, permission flags, and a 'present' bit indicating whether the PTE is valid or invalid.

Zooming in on the Linux NVDIMM driver, we can certainly conclude that it also uses the MMU extensively. Recall that the driver constructs a direct mapping (also known as an *identity mapping*) between the physical MMIO pages and the kernel's virtual address space. Furthermore, the DAX mechanism takes advantage of MMU capabilities, as it enables the OS to establish an on-demand mapping between the user space and the device. This close integration between virtual memory and an MMIO-driven device allows for another tracing methodology, which we will introduce using the following example:

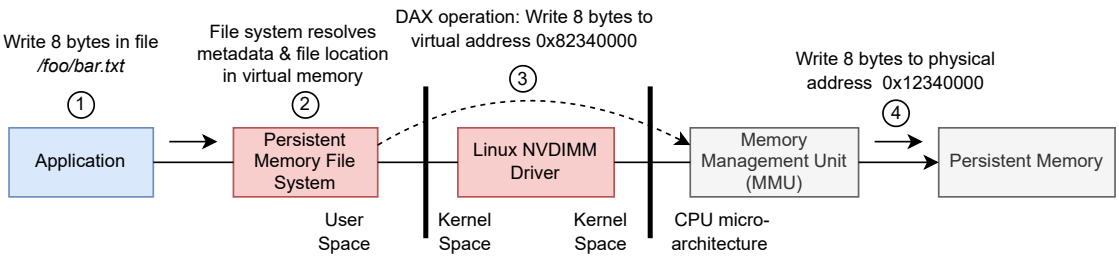


Figure 3.3: Example of MMU translation in Persistent Memory file system

1. Consider Figure 3.3. Suppose that an application wants to write 8 bytes to a file. To do this, it performs a POSIX `write()` write call, which is received by the file system.

In this example, we assume the file system is located in user space.

2. The PMEM file system receives the request, looks up the corresponding file metadata and determines that 8 bytes should be written to virtual address `0x82340000`.

3. This file system uses DAX mappings, which means that it can perform the write while completely bypassing the kernel, as indicated by the striped arrow.

4. The MMU now needs to translate the virtual address into a physical address. Depending on the current state of the MMU, we end up in one of the following three states. (1) In case of a TLB hit, the corresponding PTE can be directly fetched from the fast Translation Lookaside Buffer (TLB). (2) In case of a TLB miss, the MMU looks up the corresponding PTE in the active page table. (3) In the event of a page fault, the MMU informs the Operating System that the mapping is invalid. The task of the Operating System is to insert the new mapping, after which the CPU can *Return From Interrupt* (RTI), i.e., resume execution. Finally, the bytes are written to PMEM.

Note that the translation from virtual to physical address is done transparently in hardware *except* when a page fault is handled inside the OS. We can use this observation to our advantage. In addition to handling the fault page, we can also log the fault address, the issuing instruction, its operands, and the actual data. Together, these log entries then form an access trace. This is the essence of *pmemtrace*.

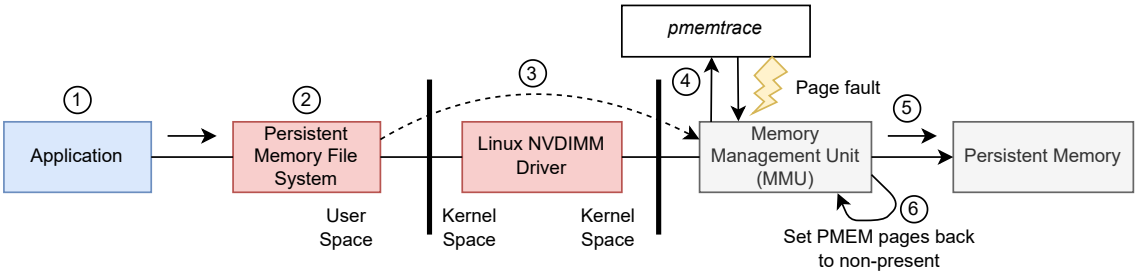


Figure 3.4: Tracing PMEM accesses through deliberate page faulting

The actual implementation is more subtle. For instance, one should only expect intermittent page faults at one memory address. In other words, when a page fault has been handled, that is, a new page has been inserted into the page table, it is unlikely to fault again in the near future. This has implications for tracing, as all succeeding accesses are not logged. To mitigate this issue, *page present bit* should be set to 'clear' to ensure that

the following reads, writes, and flushes are captured after the file system has performed the corresponding action, as depicted in Figure 3.4. The following section discusses this and other edge cases in conjunction with further implementation details.

3.3 Implementation of *pmemtrace*

This section will discuss the details of the implementation of *pmemtrace*. We start by elaborating on the implementation of both the `pmemtrace-K` and `pmemtrace-U` components. As shown in Figure 3.5, the purpose of `pmemtrace-K` is to intercept PMEM accesses and redirect these trace events through a pipe to the user space. In turn, `pmemtrace-U` provides a convenient interface in user space that enables users to run arbitrary shell commands.

Subsequently, we discuss the issues arising when tracing user space PMEM accesses and show how these complications are solved. Finally, we discuss the (potential) limitations of the work.

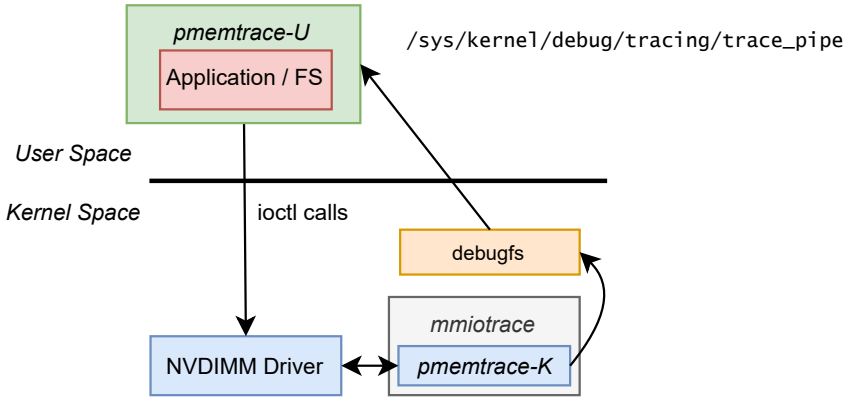


Figure 3.5: High-level design overview *pmemtrace*

3.3.1 Adaption of Existing Linux Kernel Tracing Infrastructure

The Linux kernel offers various low-level introspection tools. For example, *uprobe* and *kprobe* allows developers to dynamically break the execution of the program in the user and kernel space, respectively (81). These probes may be used for various reasons, such as debugging code or performing a performance evaluation.

In addition to these general-purpose probes, the kernel also offers so-called *kmmio* probes: a specialized probe that can be used to trace Memory-Mapped I/O (MMIO) device accesses. Within the kernel, these *kmmio* probes are implemented within the *mmiotrace* framework.

Originally developed for the *nouveau* graphics driver, *mmiotrace* can be used to reverse engineer the low-level communication of any MMIO-capable device (82). As illustrated in Figure 3.4, *mmiotrace* uses deliberate page faulting to capture MMIO events.

As mentioned earlier, PMEM is an MMIO device; therefore, we argue that using *mmiotrace* to trace PMEM accesses would be a promising direction to pursue. However, we found that, without extensive changes, *mmiotrace* cannot trace PMEM accesses due to the following three issues:

- One significant limitation of *mmiotrace* is that it only captures accesses made within MMIO mappings that are established through one of the kernel's `*__ioremap()` functions (see `arch/x86/include/asm/io.h`; Linux kernel source code). Unfortunately, the PMEM driver does not use one of the `ioremap` functions; instead, it uses the `devm_memremap_pages` function as mentioned earlier in section 3.2 (Methodology 3). As a result, accesses within these MMIO regions are not recorded by *mmiotrace*. Although the exact reason for not using an `ioremap` function in the PMEM driver is not immediately clear, we speculate that the Linux kernel considers PMEM as a special type of MMIO device, requiring more extensive configuration;

- Secondly, *mmiotrace* is only capable of capturing regular temporal `mov` instructions, for example `mov` and `movs`. It cannot capture non-temporal PMEM moves (i.e. `movnti` or cache line flushes (e.g., `clflush`, `clwb`). Consequently, instruction properties, e.g. its operands, cannot be traced;

- Third, *mmiotrace* can only trace MMIO accesses within the kernel, i.e. it cannot trace accesses in user space;

In the remainder of this section, we discuss how we adapt the *mmiotrace* infrastructure to resolve these issues. Specifically, we elaborate on the unique features of *pmemtrace*, and how these features are implemented in the source code. This is done using the example provided in Figure 3.6.

The left side of Figure 3.6 shows an execution with two machine instructions: a 64 byte non-temporal move (`movnti`) followed by a cache line flush (`clflushopt`) to persist changes. For simplicity, we assume a single core execution without *Instruction-Level Parallelism* (ILP), which means that the instructions are executed in the order depicted. The following enumeration explains how *pmemtrace* logs the instruction operand, the write address, and the operand size. Note that *pmemtrace* also logs other values, such as the current value

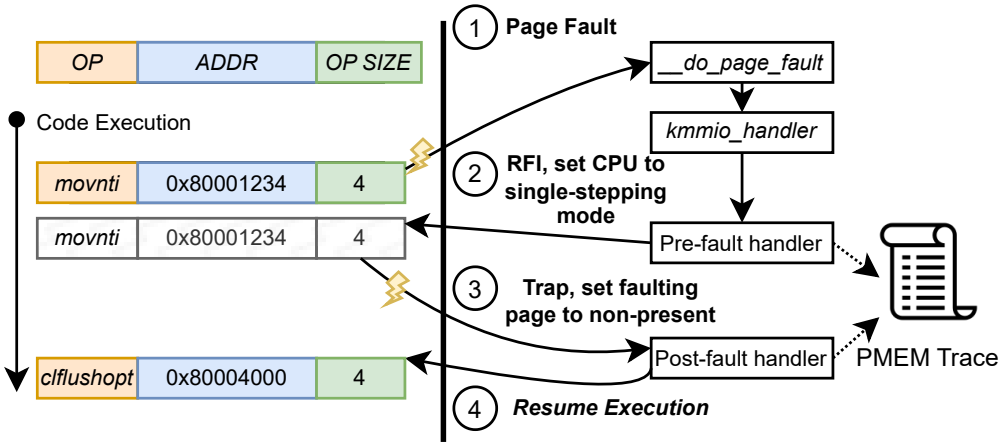


Figure 3.6: Example of logging a pending write using *pmemtrace*

of the instruction pointer (RIP), the current system time, and the CPU affinity; however, this has been omitted from the figure for simplicity.

1. At the end of the PMEM driver initialization function, `pmem_attach_disk` (source file: `drivers/nvdimm/pmem.c`), we add the following call:

```
mmiotrace_ioremap(pmem->phys_addr, pmem->size, (unsigned long)
    pmem->virt_addr, NULL);
```

   This call ensures that the entire virtual memory region, which is backed by PMEM MMIO pages, is set to 'not present'. From now on, any read or write access to this range of pages will result in a page fault.

2. Now, suppose that a file system performs a 4 byte write at address `0x80001234` by issuing an *movnti* instruction (see Step 1 in Figure 3.6). As this virtual address is contained within the address range [`pmem->virt_addr`, `pmem->virt_addr` + `pmem->size`], a page fault occurs. The corresponding interrupt vector points to the `__do_page_fault()` function. The first action that this function performs is to check if the faulting address is within the aforementioned PMEM address range by calling the `kmmio_handler()` function. If this expression evaluates to false, the fault is handled like a 'normal' page fault. If the expression is evaluated as true (that is, the faulting address is contained within the region), the `kmmio_handler` invokes the *pre-fault handler*, see Figure 3.6.

3. The `pre()` function (file: `arch/x86/mm/mmio-mod.c`) extracts the relevant data fields to store within the trace: the type of operation, the instruction opcode, and its operands (see Listing 3.2). We modify the functions `get_ins_opcode`, `get_ins_mem_width`, and `get_ins_reg_val` in order to extract this information.

```
// ...
struct mmiotrace_rw *my_trace = &get_cpu_var(cpu_trace);
my_trace->opcode_cpu = get_ins_opcode(instptr);

my_trace->phys = addr - trace->probe.addr + trace->phys;
my_trace->opcode_cpu = get_ins_opcode(instptr);

ip = (unsigned char *) instruction_pointer(regs);

// REG_READ, RED_WRITE and INS_CACHE_OP are the three events
   being traced.
if (type == REG_READ) {
    my_trace->opcode = MMIO_READ;
    my_trace->width = get_ins_mem_width(instptr);
} else if (type == REG_WRITE) {
    my_trace->opcode = MMIO_WRITE;
    my_trace->width = get_ins_mem_width(instptr);
    my_trace->value = get_ins_reg_val(instptr, regs);
} else if (type == INS_CACHE_OP) {
    my_trace->opcode = MMIO_CLFLUSH;
} else {
    pr_info("Unknown instruction: %x\n", my_trace->opcode_cpu);
}
// ...
```

Listing 3.2: Capture of Relevant Trace Events

4. After executing the pre-fault handler, *mmiotrace* sets the fault paging to *present*, and Returns From Interrupt (RFI) in CPU single-stepping mode (Step 2) by setting the Trap Flag (TF) in the CPU `EFLAGS` register (55). Setting the CPU to single-stepping modes causes the CPU to interrupt after every executed instruction. This feature allows for a design in which the post-fault handler is always invoked after executing the faulting instruction.

5. At Step 3, the `movnti` instruction has been executed, and the CPU traps back into the kernel to execute the post-fault handler. This handler performs three actions. First, in the case of a read operation, it extracts the read value by referencing the register

file. Second, it logs the completed action by writing to a dedicated pipe: `/sys/kernel/debug/tracing/trace_pipe` (see Figure 3.5). Third, the faulting page, e.g. `0x80001234`, is set back to non-present in order to capture all future events.

6. At Step 4, the program can resume execution.

In summary, we have seen how an adapted version of *mmiotrace*, *pmemtrace*, can record PMEM events with instruction-level granularity. Still, we have not yet addressed one limitation: the ability to trace PMEM accesses in the user space.

3.3.2 Tracing User Space DAX Accesses

Recall that *mmiotrace* changes page table metadata bits to capture read, write, and flush events. However, a limitation of the current version of *mmiotrace* is that it can *only* trace these events by modifying the kernel page table allocated during system boot. It is unable to trace events in user processes, as in Linux each process is assigned a new copy of this page table. Consequently, DAX events issued in the user space cannot be traced. To overcome this limitation, it is necessary to dynamically attach and detach tracing probes within the user space. This process involves multiple steps, which are illustrated in Figure 3.7.

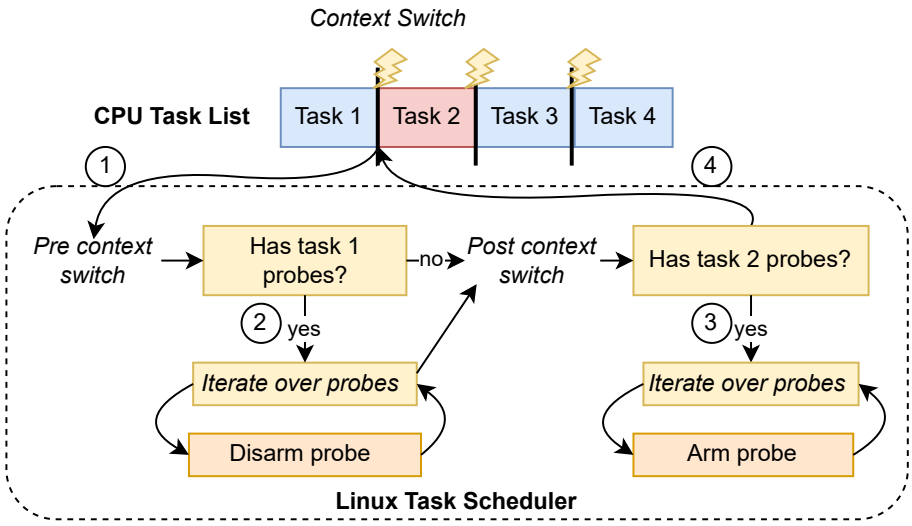


Figure 3.7: Modifications Linux's task scheduler

1. We add two new variables to the Linux abstract datatype that embodies all task-related properties (`struct task_struct`): `has_pmem_probes` and `is_pmem_sampling` (file: `include/linux/sched.h`). The first variable denotes whether this task has

one or more active tracing probes. The purpose of the second variable will become clear when we discuss sampling-based tracing methodology in subsection 3.3.4.

Inside the Linux main scheduler function, `__schedule` (file: `kernel/sched/core.c`), we add two new function invocations: `mmiotrace_detach_user_probes` and `mmiotrace_attach_user_probes` (both are defined in `arch/x86/mm/mmio-mod.c`).

2. The `mmiotrace_detach_user_probes` function is invoked **before** the scheduler performs the task switch (see Figure 3.7). This function ensures that all probes currently attached to the current running process, in case of Figure 3.7 Task 1, are disabled so that other tasks cannot trigger spurious events after the scheduler performs the context switch.

3. The `mmiotrace_attach_user_probes` function is invoked directly **after** the context switch was made, and just before the CPU execution returns to the user space. This function ensures that all trace probes assigned to Task 2 are enabled.

4. Finally, the CPU starts executing Task 2.

The final step is to register the actual probes in the event that an application requests a DAX memory mapping using the `mmap` system call. The `dax_associate_entry` function is responsible to establishing such a mapping between the applications's address space and PMEM. To track accesses in its region, we add a `mmiotrace_ioremap` function call, which in turn creates a new probe. Likewise, if an application issues the `unmap` system call or terminates, the `mmiotrace_iounmap` function call in `dax_disassociate_entry` ensures the tracing probe is deactivated and removed.

3.3.3 Tracing CPU Fence Instructions

In Background subsection 2.1.3, we discussed why explicit cache line flushing and memory fences are essential for data consistency. In summary, the first ensures that data always persist in the event of a failure, while the second ensures that PMEM read/write operations are always executed in order. Listing 3.3 shows a typical compiled-generated machine code implementation of a durable PMEM write. CPU fencing instructions (`mfence`, `sfence` and `lfence`) enforce ordering, while the `clflushopt` instruction ensures data at register `r0` is directly written to PMEM. This is a typical access pattern when one wants to bypass the cache; a flushing operation must always be encapsulated by a memory fence (83).

In previous sections, we discussed how non-temporal instructions, such as `movnti` and `clflushopt`, are captured by *pmemtrace*. However, we have not yet considered how to detect fencing instructions. To better illustrate this, consider Listing 3.3. When the CPU executes the `movnti` instruction, it writes data to the virtual address contained in `r1`. This causes a trap in the kernel, which is then recorded by *pmemtrace*. Since fencing instructions such as `mfence` only perform serialization and do not operate on memory directly, they cannot be captured by deliberate page faulting. As a result, we developed an alternative methodology in the form of an LLVM (84) optimization pass, `FenceInstrument`, which instruments fencing instructions by callbacks.

```
mfence
// Two operands: 'r0' is the source address (e.g. a user space
   buffer), 'r1' is the destination address (e.g. PMEM-backed
   virtual memory)
movnti r0, r1
clflushopt r0
mfence
```

Listing 3.3: Typical Compiler-generated Low-level Implementation of PMEM Write

The `FenceInstrument` pass starts by traversing the program's *Abstract Syntax Tree* (AST): a tree that resembles the program source code in a structured manner (85). In LLVM, an AST is constructed from a wide variety of different objects. For example, a function/basic block object (`llvm::BasicBlock`) contains multiple instruction objects of type (`llvm::Instruction`). In turn, the object `llvm::Instruction` is polymorphic, giving rise to an even larger set of subtypes, e.g. `llvm::FenceInstr`. Hence, to test whether an instruction is a fencing instruction, we can walk the AST and check whether the instruction is of type `FenceInst`, i.e.: `isa<FenceInst>(inst)`. If this condition is true, the pass inserts multiple assembly instructions before the fencing instruction, as shown in Listing 3.4. Specifically, it inserts kernel callback in the form of a new system call: `sys_trace_fence` (number: 350). According to the `x86_64` Linux system call convention (86), the system call number of the corresponding system call must be stored in the `rax` register. The first argument is stored in the `edi` register, in this case, it denotes the type of CPU fence (see Listing 3.4). The original values of both registers are pushed to the stack and restored after performing the `syscall` such that their original values are not tainted. In kernel space, the `sys_trace_fence` function writes a new entry to the trace log (see `arch/x86/mm/mmio-mod.c` file for code implementation, and Returns From Interrupt (RTI). For reference,

the source code of the pass can be found in the `llvm_patch/FenceInstrument.cpp` file within the Git repository

Finally, fences *within* the kernel are traced slightly differently. Instead of automatically inserting `sys_trace_fence` callbacks, fences in the kernel should manually be annotated by inserting a `sys_trace_fence` function call just above the fencing instruction. Recompiling the entire kernel within the LLVM compiler pass included should log all kernel fences without manual configuration, however, this is considered future work.

```
1  // Save register state.
2  push %rax
3  push %rdx
4  // Invoke kernel callback in form of syscall nr. 350
5  movl $350, %eax
6  movl $0, %edi // 0 = mfence, 1 = sfence, 2 = lfence.
7  syscall
8  // Restore register state.
9  pop %rdx
10 pop %rax
11
12 mfence
```

Listing 3.4: LLVM Optimizer Pass: Instrumentation of Fencing Instructions

In the next section, we quantify the performance impact of *pmemtrace* and show how a sampling-based tracing strategy can reduce overhead at the cost of tracing accuracy.

3.3.4 Quantifying Tracing Overhead

In this subsection, we first assess the performance of *pmemtrace*. Specifically, we show that tracing at an instruction level granularity results in a significant amplification of the application run time. Then, we show that this incurred overhead can be partially mitigated by switching to a sampling-based tracing strategy.

To evaluate the impact of tracing, we perform an experiment in which different-sized sets of randomly generated data is appended to the back of a file. Then, the impact of tracing can be quantified by comparing the run times with and without tracing. An artifact to reproduce the results is provided in section 7.2. Table 3.2 displays the results of the experiment.

Clearly, the amplification of the run time when tracing is enabled is extreme (up to 296×). However, it is important to note that achieving the highest trace accuracy is of primary interest here (**RQ1**). We strive for performance when we replay these traces inside

Size Written	Avg. runtime w/o tracing	Avg. runtime with tracing	Amplification
4 MiB	0.009 s ($\sigma = 0.000$)	2.590 s ($\sigma = 0.027$)	287×
8 MiB	0.019 s ($\sigma = 0.003$)	5.155 s ($\sigma = 0.030$)	271×
16 MiB	0.036 s ($\sigma = 0.004$)	10.301 s ($\sigma = 0.026$)	286×
32 MiB	0.070 s ($\sigma = 0.002$)	20.781 s ($\sigma = 0.103$)	296×

Table 3.2: Run time amplification tracing file append with random data in QEMU VM, 10 runs per working set size

the *pmemanalyze* framework. Still, we believe that these run times are infeasible, and hence we deem it necessary to switch to a sampling-based strategy in which PMEM-related read, write, flush, and fencing machine instructions are traced during program execution.

The implementation of a sampling-based strategy is carried out as follows:

1. Recall that we decided to divide `pmemtrace` into two user and kernel space components: `pmemtrace-U` and `pmemtrace-K` (see Figure 3.5). To achieve data sampling, `pmemtrace-K` spins up a separate kernel thread: `pmemtrace_sampler` (source file: `arch/x86/mm/mmio-mod.c`). This thread repeatedly arms and disarms the tracing probes based on two settings passed by `pmemtrace-U`; the *sample rate* (range: $0-240$ hertz) and the *duty cycle*. The duty cycle, expressed as a percentage between 0% and 100%, determines the proportion of time the tracing probes for `kmmio` are armed for the selected frequency. An example of how to use `pmemtrace`'s CLI interface is provided in the following example:

```
# Usage: pmemtrace [args] [trace name] [command]
sudo pmemtrace --sample-rate 30 --duty-cycle 0.5
    simple-file-append sudo bash -c \
     "head -c 8M </dev/urandom >/mnt/pmem_emul/some_file.txt"
```

Listing 3.5: Example of using *pmemtrace* to trace an 8 MiB random file append operation

Another sampling strategy is based on the number of page faults handled. In other words, when *pmemtrace* has captured a user-specified number of page faults, data collection is temporally disabled until the number of page faults has reached a predefined threshold.

```
sudo pmemtrace --sample-rate-pfaults 10 simple-file-append
    sudo bash -c \
     "head -c 8M </dev/urandom >/mnt/pmem_emul/some_file.txt"
```

Listing 3.6: Example of page fault based sampling strategy

2. In user space, `pmemtrace-U` buffers all events retrieved through the *trace pipe*, as illustrated in Figure 3.5. After the provided shell command finishes execution, the output is compressed using Apache Parquet (87) and stored inside the current working directory. This trace file is now ready to be replayed in the performance analysis framework *pmemanalyze*.

We evaluate the performance impact of sampling by measuring two metrics, run time and *accuracy*. Accuracy is an important metric as due to sampling, a substantial number of events might be lost, affecting the trace's accuracy. We quantity the tracing accuracy as the ratio between the number of bytes to be read/written and the number of bytes actually captured by the *pmemtrace* tool. Figure 3.8 and Table 3.3 display the results of the experiment. Figure 3.8 shows the impact of the tunable sampling parameters, duty cycle and frequency, on the tracing accuracy. Each subfigure corresponds to a selected duty cycle, with the horizontal axis depicting the frequency and the vertical axis displaying the amount of write (in MB) per second. This allows us to compare tracing overhead and accuracy across different sampling settings.

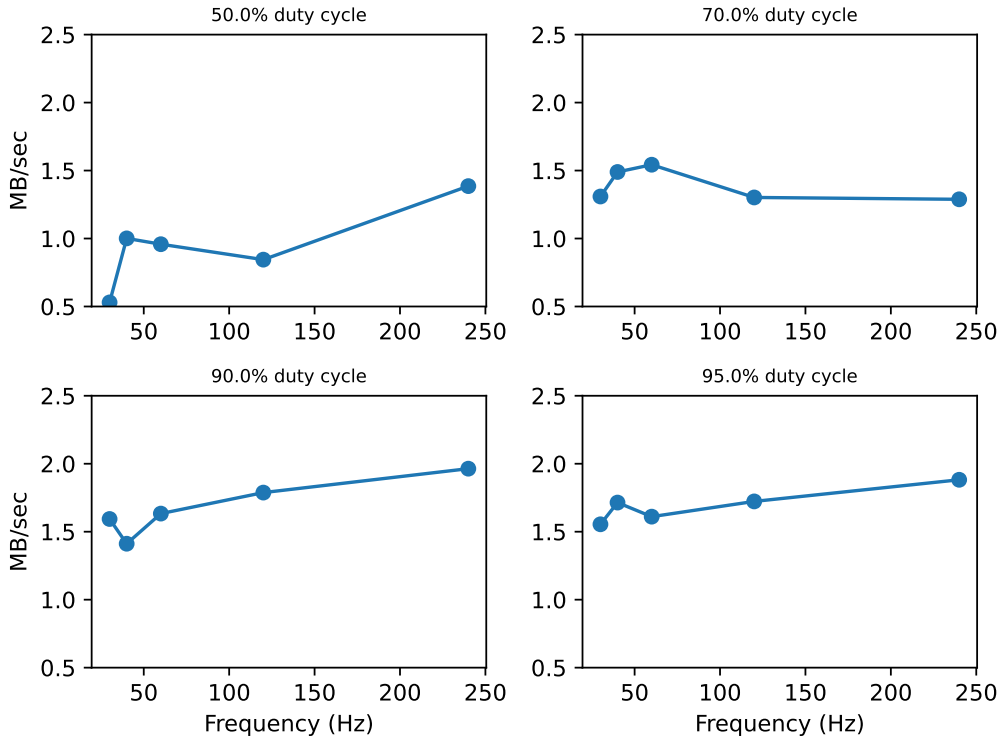


Figure 3.8: Impact frequency (hertz) and duty cycle (%) on amount of writes captured (MB per second, higher is better), 16 MiB random write.

As shown in Figure 3.8, the duty cycle has the greatest impact on accuracy. For example, a duty cycle of 95% with a sampling frequency of 120 hertz results in the capture of approximately 1.75 MB per second. Table 3.3 shows the amplification in run-time, using the same random append workload as in the experiment without sampling. We used a sampling frequency of 120 hertz and a duty cycle of 95%, since in the case of this experiment these parameters showed relatively high accuracy at a low run-time cost (see Figure 3.8).

Size Written	μ runtime w/o sampling	μ runtime sampling	Trace accuracy (%)	Amplification
4 MiB	2.590 s ($\sigma = 0.027$)	0.594 s ($\sigma = 0.081$)	15.8%	77×
8 MiB	5.155 s ($\sigma = 0.030$)	1.499 s ($\sigma = 0.100$)	15.62%	82×
16 MiB	10.301 s ($\sigma = 0.026$)	2.684 s ($\sigma = 0.142$)	15.10%	80×
32 MiB	20.781 s ($\sigma = 0.103$)	5.553 s ($\sigma = 0.314$)	18.1%	83×

Table 3.3: Impact sampling on execution time and trace accuracy. Sampling Settings: 120 hertz, 95% duty cycle, 10 runs

The results shown in Table 3.3 show a clear relationship between tracing accuracy and run-time; if the size of the work set increases exponentially, the average run time also scales exponentially. When run times become infeasible for a given workload, a lower duty cycle (e.g., 50%) effectively decreases run times at the cost of accuracy.

In summary, we show that a sampling-based data collection approach helped reduce the overhead of instruction-level tracing. In the next section, we discuss the limitations of *pmemtrace* and how these limitations relate to the design requirements established earlier in section 3.1.

3.4 Limitations and Discussion

This section discusses the two limitations of *pmemtrace*: the amplification of runtime and multicore instability.

Run-time Amplification. Even though sampling helped to decrease run times, *pmemtrace* still introduces a lot of overhead to the application (or file system) being traced. We believe that the majority of this overhead is attributable to CPU single-stepping. Recall that *pmemtrace* uses CPU single-stepping to regain control of the execution so that it can reset the faulting page back to not present after completing the read/write (see Figure 3.6). This is essential for logging subsequent events at the same virtual address. Vogl et al. (88) show that the use of Instruction Level Monitoring (ILM) techniques based on CPU single-stepping execution, such as `pmemtrace`, results in massive application slowdowns; up to

a factor of 545. Manual analysis using `perf` revealed that in the case of *pmemtrace* on average 61% of the total run time can be attributed to the CPU single-stepping execution. Figure 3.9 displays this and other forms of overhead for multiple working sets [1].

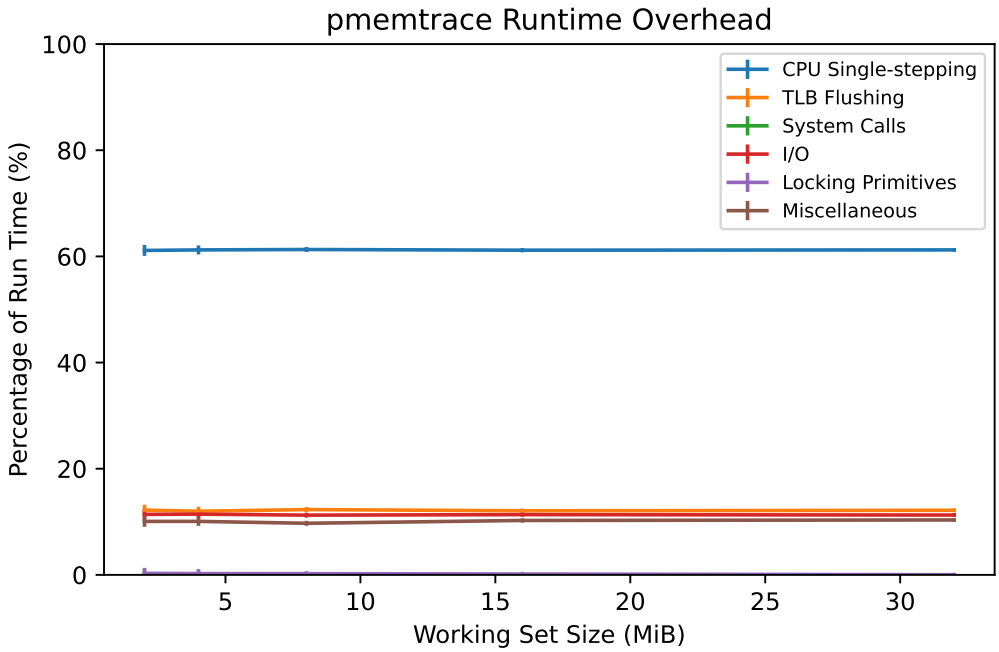


Figure 3.9: *pmemtrace* runtime overhead in kernel, 10 runs for each working set.

Van Bulck et al. (89) suggests an alternative to single-stepping. They proposed a tracing technique that mimics CPU single-stepping using APIC hardware timers. Due to time constraints, we have not been able to implement this variant, which can be considered promising future work.

Multicore Tracing Implications. Recall that *pmemtrace* sets all PMEM-backed pages to not present. If an application accesses one of these pages, a CPU exception is generated, which is then handled by the kernel page fault handler. Although this methodology has proven itself in a uniprocessor execution environment (82), we will show it has two implications in a multicore environment: lost I/O events and race conditions. We demonstrate both issues through an example.

Suppose an application, scheduled on core 0, wants to read PMEM at address `0x1000`. This will result in a CPU exception, as the corresponding page was set to 'not present.' To handle the exception, the CPU jumps to the kernel page fault handler, whose address can

[1] An artifact to reproduce this and other experiments is provided in section 7.2

be found by looking up the corresponding exception handler in the *Interrupt Vector Table* (IVT). Now, *pmemtrace* uses the existing *mmiotrace* kernel infrastructure to mark the page as present and resume execution in single-stepping mode. While this CPU is single-stepping execution to replay the faulting instruction, other cores can access the address `0x1000`, leading to a complication. As the corresponding page is marked present, the MMU can resolve the corresponding physical address without the involvement of the kernel page fault handler. Consequently, these events are irreversibly lost. In the worst case, this may also lead to data race conditions, as two cores may access the same page simultaneously, resulting in deadlocks.

We found that these two issues are particular to this form of tracing and are not well addressed in the related literature. There is one related (conceptual) contribution by Kim et al. (90) that proposes a solution to these issues through the use of a *shadow page table*. When a processor encounters a page fault, it copies the existing page table and inserts a page at the faulting address. Meanwhile, all other processors continue to reference the original page table. Therefore, only the processor that handled the page fault has access.

Although we are confident that this contribution *may* mitigate event loss and data races, we do not believe that it solves the problem efficiently. Since cores must coordinate all page table changes, all reads and writes affecting the same virtual address are effectively serialized, degrading performance as when switching to uniprocessor execution. Additionally, we envision that the required extensive kernel code changes and (potential) implications are substantial. As a result, we switch the kernel to uniprocessor mode while tracing; see the `enter_uniprocessor` function defined in `arch/x86/mm/mmio-mod.c`. For experimental purposes, multicore event capture can still be enabled by setting the `--enable-multicore` flags, as illustrated in Artifact 1 (section 7.1).

Evalation by Requirements. We conclude this chapter by evaluating *pmemtrace* according to the functional requirements set earlier. Table 3.4 summarizes these requirements, including an assessment of whether a requirement is passed (green), partially passed (orange), or failed (red).

We implemented a tracing methodology that can trace PMEM read, write, and flushing operations at byte-granularity in real time without tainting the application's functional behavior or causing a crash. Sampling support was included as runtimes turned out to become infeasible in high data velocity workloads. Thus, we consider the requirements RQ1, RQ2, RQ3, RQ4, RQ6 as passed. We were unable to implement stable multicore/SMP support due to data racing (RQ5).

ID	Requirement	Passed
RQ1	Capture Instructions at Byte-Granularity	✓
RQ2	Capture in Real-Time	✓
RQ3	No Crash/Kernel Panics	✓
RQ4	Capture Flushing Events	✓
RQ5	SMP Support	✗
RQ6	Sampling-based Tracing	✓

Table 3.4: Evaluation of requirements *pmemtrace* tracing tool

3.5 Conclusion and Future Work

We designed and implemented *pmemtrace*, a tool that can capture PMEM traffic at instruction-level granularity by leveraging the CPU's Memory Management Unit (MMU) paging infrastructure. Our evaluation showed that *pmemtrace* can accurately collect all read, write, and flushing events, however, at a high runtime cost. A sampling-based data collection approach helped to reduce this overhead; see Table 3.3.

We believe memory tracing through deliberate page faulting enables a comprehensive analysis of PMEM access patterns. Still, its limitations are evident: speed and multicore scalability. In our view, future work should focus on mitigating these limitations. A promising research direction would be investigating whether the LLVM `FenceInstrument` optimization pass can be extended to trace accesses. In concrete terms, this implies that every memory load and store instruction should be encapsulated similarly, e.g., surrounding an `mov` instruction by a kernel trap. The main challenge here is distinguishing between instructions that will read/write PMEM and those that will only affect DRAM.

3. *PMEMTRACE*: A TOOL TO COLLECT ACCESS TRACES AT INSTRUCTION GRANULARITY

4

pmemanalyze: A Tool to Analyze Micro-Architectural Overhead

This chapter discusses the design, implementation, and evaluation of *pmemanalyze*: a tool that replays *pmemtrace*-captured file system access traces in order to evaluate the microarchitectural performance impact of a file system. We start by explaining the motivation and design considerations behind *pmemanalyze*, specifically addressing the need to include microarchitecture-related events in performance analysis (section 4.1). We then present a literature study on the microarchitectural performance characteristics of Intel Optane DC Persistent Memory (section 4.2), which helps us to define the relevant performance metrics to be integrated into the *pmemanalyze* tool (section 4.3).

4.1 Motivation and Design Considerations

In order to emphasize the need for a tool that can evaluate the microarchitectural performance of PMEM file systems, it is essential to quantify the portion of overhead that can be attributed to the (near) hardware interaction. Essentially, this determines to which extent micro-optimizations can improve quality attributes such as latency and throughput.

Quantifying the Impact of Hardware on File System Performance. In order to identify the specific areas where file system overhead occurs, we performed an experiment in which hardware and software overhead is categorized into different groups. This performance breakdown was performed for four file systems (ext4-DAX (49), SplitFS (19), NOVA (20) and UFS (27)) and its results are depicted in Figure 4.1. For each file system,

we run the *Flexible I/O Tester* (FIO) benchmark (91). This benchmark can be configured to perform a pre-selected set of file operations automatically. In our case, we generate 20.000 files and perform 256 byte sequential and random reads and writes on these files. We specifically chose a granularity of 256 bytes to stress PMEM's byte-addressable properties. Detailed instructions on how to reproduce these results can be found in the Artifact provided in section 7.2.

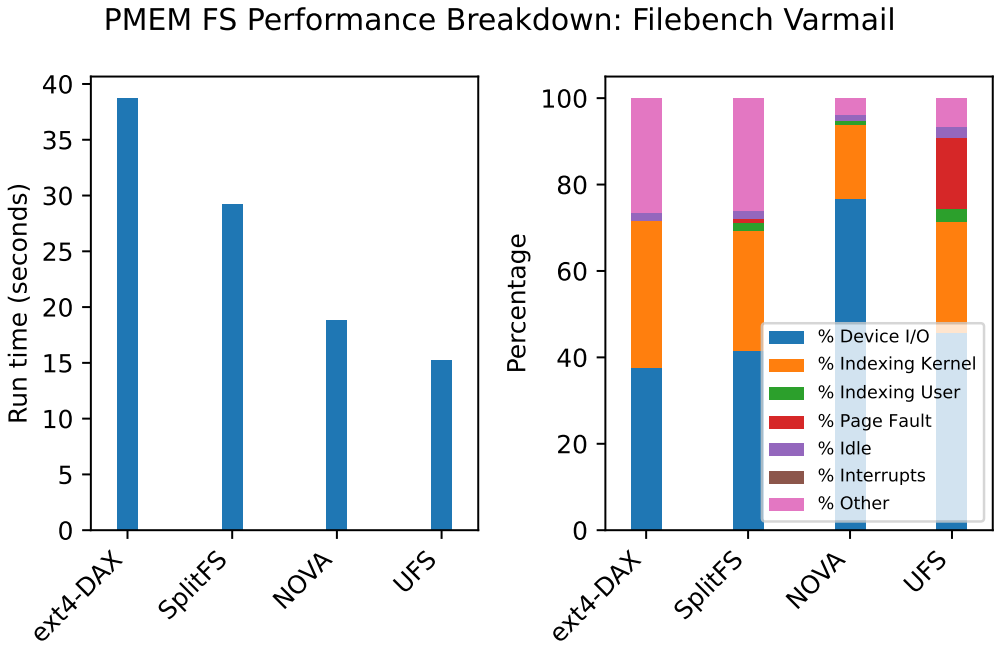


Figure 4.1: Performance breakdown four PMEM file systems, FIO settings: 20000 files (file size: 0.1 MiB), 256 byte append, 50%/50% read/write ratio.

Figure 4.1 displays the results. In terms of run time, UFS achieves performs the best while ext4-DAX performs the worst. It is also evident that the software overhead within the software storage stack accounts for 24% to 68% of the total run time, depending on the file system used. The remaining overhead is attributed to actual I/O operations involving Persistent Memory and page fault handling. I/O operations are particularly important when considering performance at the microarchitectural level since they represent the CPU time spent communicating with the PMEM device at the level of CPU load/store instructions (55). Additionally, we suspect that page faults might be relevant, as they occur at the microarchitectural level within the CPU's Memory Management Unit (MMU).

Towards a Design of *pmemanalyze*. In order to answer the question of how to design a tool that can be used to assess the microarchitectural performance of PMEM file systems

(sub-question RQ3), it is important to consider how Persistent Memory integrates into the computer hierarchy. We already touched on this topic in the background chapter (chapter 2); however, we have not discussed how relevant components (e.g., the CPU and its caches, the PMEM internals, and the MMU) of the hardware architecture can impact performance in the storage software stack.

We conduct a literature study to determine these performance-related idiosyncrasies (sub-question RQ1). The findings of this study help us to define the appropriate performance metrics that are included in the analysis tool *pmemanalyze*. The following section discusses the methodology of the literature study and presents its findings.

4.2 Literature Study: The Identification of Prevalent Micro-Architecture Performance Issues and Metrics

In this section, we present the results of a short but complete literature study that aims to answer sub-question RQ1: *What are the micro-architectural performance-related idiosyncrasies of Intel Optane DCPMM?*

Literature Study Design. To ensure that the findings of a literature study are precise and complete, researchers should use appropriate literature selection methodologies. Well-known selection techniques are a *Systematic Literature Review* (SLR) (92) and *snowballing* (93). A Systematic Literature Review allows for a very comprehensive and structured study; however, we believe that these methodologies are more applicable in studies with a large search space. On the other hand, snowballing allows a systematic search in a constrained search space in which this search space is expanded by exploring the references of relevant papers (backward snowballing) and papers that refer to these relevant papers (forward snowballing). In this study, we use the snowballing methodology, as we have already systemically explored the research field of Persistent Memory in our literature study on file systems (11). This allows for a search space whose boundaries are already clearly defined through concise inclusion and exclusion criteria.

Title	Year
Persistent Memory I/O Primitives (6)	2019
An Empirical Guide to the Behavior and Use of Scalable Persistent Memory (10)	2020

Table 4.1: Seed papers literature study

The search space is defined by specifying the seed papers (see Table 4.1), the search keywords (see Table 4.2), and the paper inclusion/exclusion criteria, as denoted in the enumeration below:

- I.1: The work targets a real Persistent Memory device (no emulation): Intel Optane DCPMM;
- I.2: The work discusses the micro-architectural performance properties of an Intel Optane DCPMM;
- I.3: The work generalizes its findings, meaning that they are applicable to more than one application or file system;
- I.4: *Optional.* The work addresses the micro-architectural performance impact of accessing remote/NUMA Intel Optane DCPMM;
- E.1: The work was published before 2017; the year the first Intel Optane DCPMM devices emerged;

Keyword	**Accepted**	**Rejected**
Seed Paper Forward Snowballing	6	> 50 [1]
Intel Optane DCPMM Micro-architectural Performance	3	> 25
Persistent Memory Low-level Performance	0	9
Persistent Memory Cache Interference	2	7

Table 4.2: Exploratory keywords and paper accepted/rejected Count

The choice of Intel Optane DCPMM as the primary target device is supported by two key factors. First, it has the highest adoption rate in industry (48). Second, our earlier literature survey on Persistent Memory File Systems (11) revealed that most of the literature focuses on assessing and improving the performance of Intel Optane DCPMM, expanding the amount of (potentially) relevant literature to the fullest extent possible.

The following conferences were included in the literature search: ACM (SIGARCH, SIGOPS, SPAA), ASPLOS, EuroSys, IEEE, ISCA, ODSI, MDPI, SC, SOSP, USENIX (FAST), and VLDB. Furthermore, we use the ACM, Arxiv, and Google Scholar search engines to find additional work.

Overview of Findings. We noticed a significant overlap in the findings of all papers included in this literature study. Therefore, we grouped and summarized the contributions of these papers in a concise way, as shown in Table 4.3.

In the following subsections, we will elaborate on these findings individually, together with fitting performance metrics that are incorporated into the *pmemanalyze* tool. We will begin by discussing the findings that specifically relate to a very specific part of the micro-architecture, e.g., Optane staging queues, and then gradually proceed to the higher-level findings, i.e., NUMA interconnects between CPU packages. Throughout the discussion, we frequently refer to Figure 4.2, which visually illustrates the placement of each finding within the microarchitecture, identified by its unique ID as listed in Table 4.3.

Category	Findings		Literature
The Effects of Access Patterns on Latency and Throughput	**ID**	**Finding**	(6, 9, 10, 72, 73, 77, 94, 95)
	1	Mismatch in Access Granularity Degrades Write Bandwidth	
	2	CPU Prefetching Mispredictions causes Cache Pollution.	
	3	Poor CPU Prefetching Performance	
Optane's Internal Buffering	**ID**	**Finding**	(9, 10, 73)
	4	Read Amplification due to Optane's Read Buffer	
	5	Contention at Optane's RMW Buffer	
Implications of NUMA Accesses	**ID**	**Finding**	(9, 72, 76, 95, 96, 97)
	6	Slow Cross-Socket Interconnect Degrades Throughput	

Table 4.3: Overview of findings literature study

4.2.1 The Effects of Access Patterns on Latency and Throughput

All studies that examine the impact of Intel Optane DCPMM access patterns (6, 9, 10, 72, 73, 94, 95) agree with the observation that random access patterns, both reads and writes,

[1] All citations listed in the seed papers bibliography that were not accepted can be considered rejected, hence the high count.

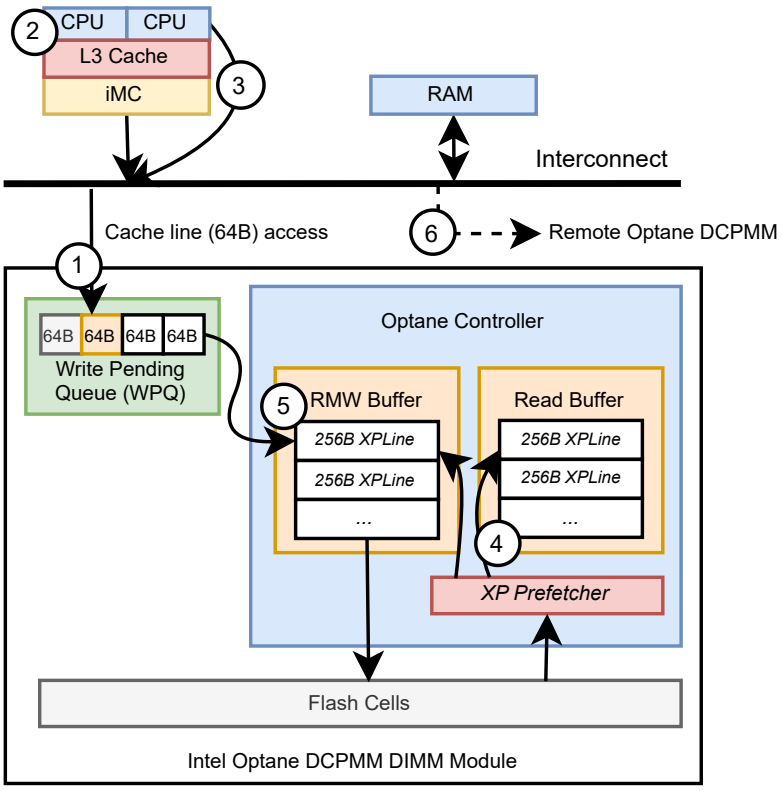


Figure 4.2: Graphical overview micro-architecture, depicting findings literature study Table 4.3

have a detrimental effect on bandwidth compared to a more sequential access pattern. In the following paragraphs, we will provide a comprehensive explanation of why this is the case, presenting relevant concepts. Additionally, we will establish performance metrics that accurately quantify the impact of these concepts. These metrics will be integrated into the *pmemanalyze* tool.

Concept 1: Mismatch in Access Granularity Degrades Write Bandwidth. Several studies (9, 10, 72) show that there is a discrepancy in access granularity between the CPU and Intel Optane DCPMM. In modern systems, CPUs handle data at cache line granularity, typically 64 bytes. However, as shown in Figure 4.2, Intel Optane DCPMM handles data at a larger granularity of 256 bytes, called an *XPLine*. This difference in granularity raises a performance issue. When a CPU sends a 64 byte (cache line-sized) write request to PMEM, it can lead to 4× the requested amount of bytes being written, also known as write amplification. Optane DCPMM employs a prefetcher called the *XP Prefetcher* to account for this performance loss. This prefetcher can coalesce four adjacent

64 byte requests into a single *XPLine* access, which can then be written to flash. However, in the case of random address patterns where the accesses are not linear, the *XP Prefetcher* cannot effectively merge requests. This results in the aforementioned write amplification.

Concept 2: CPU Prefetching Mispredictions cause Cache Pollution. The second finding relates to prefetching at the CPU. In this context, prefetching is the process of proactively loading data from a slow memory (e.g., PMEM) into a faster memory (e.g., the Low-Level Cache) before it is actually needed (98). In the case of random workloads, where PMEM accesses lack a clear linear (also known as *strided*) access pattern and the prefetcher's accuracy decreases, the number of CPU prefetching mispredictions increases. This, in turn, leads to unnecessary Optane operations, which pollute the caches, thus wasting valuable device resources.

In terms of access latency, Xiang et al. (9) show that, depending on the level of randomness of an access pattern, up to 50% more CPU cycles are spent per Optane DCPMM read access compared to a fully sequential read pattern due to this issue of cache pollution. In the next finding, we will show that random writes are less affected *if* they bypass the cache, as this effectively disables the prefetching mechanism.

Concept 3: Cache Bypass Improves Write Bandwidth. Two studies (9, 10) concluded that explicitly flushing the cache after a PMEM store improves the write bandwidth. This finding may initially seem contradictory, considering that caching is commonly known to decrease access latency, in particular when accessing DRAM (98). However, the behavior can be explained as follows. When caching is enabled, the data is eventually evicted from the cache and written back to the device. As cache eviction is not guaranteed to be FIFO, an initially sequential PMEM access pattern might effectively be transformed into a random one. Such a random access pattern is again susceptible to the aforementioned prefetching issue (**Concept 2**).

In conclusion, it is evident that random reads suffer from poor CPU prefetching performance, while random writes experience write amplification due to a mismatch in access granularity between the CPU and Optane. Now, we will define the appropriate performance metrics to detect and quantify the impact of these factors accurately.

Performance Metrics. We define multiple performance metrics to quantify whether a trace captured by *pmemtrace* has poor prefetching performance and/or suffers from read or write amplification. The metrics are derived from performance counters available in

Intel's *Performance Monitoring Unit* (PMU), including its *Precise Event-Based Sampling* (PEBS) extension (99). The relevant metrics, along with their units of measurement (e.g. GB/s), goals, and definitions, are presented in Table 4.4.

Table 4.5 presents the performance metrics used to investigate the impact of caching. Metric 7 quantifies the number of retired instructions that bypass the third level of the CPU Low-Level Cache (LLC), facilitating the examination of whether an application is bypassing the cache. Metric 8 and Metric 9 are included on the basis of an observation made by Yi et al. (100). They note that in scenarios where DRAM and PMEM compete for resources, for example, simultaneously reading and writing to the shared memory bus at 100% utilization, the PMEM throughput degrades up to 64%. To assess the interaction between DRAM and PMEM, we track the number of retired load instructions served by DRAM (Metric 8) and the Cycles Per Instruction (CPI, Metric 9) for each load/store machine instruction, which provides an estimate of the access latency for DRAM accesses.

The implementation of these metrics will be discussed later in section 4.3.

Metric	Definition
Metric 1: iMC elapsed clock ticks **Unit**: CPU ticks **Source**: uncore event **Goal**: Used to precise bandwidth measurements (**Concepts 1** and **2**).	`UNC_M_CLOCKTICKS`
Metric 2: PMEM read amplification **Unit**: bytes **Source**: uncore event **Goal**: Investigate potential mismatch in device access granularity (**Concept 4**).	$\frac{64 \times \texttt{UNC\_M\_PMM\_RPQ\_INSERTS}}{\text{OS bytes read}}$
Metric 3: PMEM write amplification **Unit**: bytes **Source**: uncore event **Goal**: Investigate potential mismatch in device access granularity (**Concept 4**).	$\frac{64 \times \texttt{UNC\_M\_PMM\_WPQ\_INSERTS}}{\text{OS bytes written}}$
Metric 4: PMEM write bandwidth **Unit**: GB/s **Source**: uncore event **Goal**: Quantify write performance, high value is desirable (**Concept 3**).	$\frac{(64 \times \texttt{UNC\_M\_PMM\_WPQ\_INSERTS}/(1 \times 10^9))}{\text{sample duration}}$
Metric 5: Device read latency **Unit**: us **Source**: uncore event **Goal**: Differentiate access latencies for different kinds of memory accesses. A low value is desirable. Related to **all concepts**.	$\frac{(1 \times 10^9) \times \frac{\texttt{UNC\_M\_PMM\_RPQ\_INSERTS}}{\texttt{UNC\_M\_RPQ\_OCCUPANCY}}}{\texttt{UNC\_M\_CLOCKTICKS}}$
Metric 6: Device write latency **Unit**: us **Source**: uncore event **Goal**: Differentiate access latencies for different kinds of memory accesses. A low value is desirable. Related to **all concepts**.	$\frac{(1 \times 10^9) \times \frac{\texttt{UNC\_M\_PMM\_WPQ\_INSERTS}}{\texttt{UNC\_M\_WPQ\_OCCUPANCY}}}{\texttt{UNC\_M\_CLOCKTICKS}}$

Table 4.4: Performance metrics related to iMC and PMEM

Metric	Definition
Metric 7: Retired PMEM and DRAM instructions that bypass L3 cache. **Unit**: count **Source**: core event **Goal**: Confirm that non-temporal load and store instruction fully bypass the CPU cache (**Concepts 2** and **3**).	`MEM_LOAD_RETIRED.LOCAL_PMM` `MEM_LOAD_RETIRED.L3_MISS`
Metric 8: Number of memory load instructions that were served by DRAM, thus bypassed the cache **Unit**: count **Source**: offcore event **Goal**: Investigate whether there is correlation/interference between PMEM and DRAM (**Concepts 2** and **3**).	`L3_MISS_LOCAL_DRAM.ANY_SNOOP`
Metric 9: Cycles Per Instruction (CPI). **Unit**: CPU cycles **Source**: TSC unit **Goal**: Latency estimate of PMEM and DRAM accesses. Also used to verify if Metric 5 is reliable.	High Precision Event Timer (HPET) measurement using C++'s `std::chrono::steady_clock` abstraction

Table 4.5: Performance metrics related to caching

4.2.2 Optane's Internal Buffering

In the previous section, we introduced the notion of an *XPLine*, the granularity at which Intel Optane DCPMM handles data internally. Expanding upon this, we elaborate on how Optane processes reads and writes internally and why data placement plays an important role in increasing performance.

Multiple studies (9, 10, 73) confirmed the presence of two distinct 16 kB buffers in Optane DCPMM: a dedicated *read* buffer and a *Read-Modify-Write* (RMW) buffer. The following two sections will detail each buffer's functionality and idiosyncratic performance behavior.

Concept 4: Read Amplification due to Optane's Internal Read Buffer. Optane's internal read buffer contains data that is *exclusive* (not present) in one of the CPU's caches. When the CPU initiates a 64-byte (cache line-sized) read operation of which its data is not contained in the CPU Last Level Cache (LLC), in the case of Intel the L3 cache, the request is sent to the Optane media, which then fetches an entire 256-byte *XPLine* into the 16 kB read buffer (see Figure 4.2). This means that the initial request experiences read

amplification (RA) of 4 [1]. Afterward, a maximum of three adjacent requests to this cache line can be processed directly as they are contained within the same *XPLine*, reducing the RA for these requests to 1. Once the entire *XPLine* has been loaded into the CPU cache, it is removed from the 16 kB read buffer. Therefore, the first subsequent read that hits this *XPLine* will again have an RA of 4, as the entire *XPLine* needs to be recovered from Optane flash cells. Unfortunately, there seems to be no established methodology to prevent this RA from occurring. Xiang et al. (9) highlighted in their study that the read buffer is exclusive to CPU caches, which means that we cannot directly control data placement to improve locality. As a result, the options to prevent this RA from occurring are limited.

Concept 5: Contention at Optane's RMW buffer. When a CPU performs a write operation, the corresponding data is initially stored in the Read-Modify-Write (RMW) buffer, see Figure 4.2. Similarly to the read buffer, the RMW buffer stores data at the granularity of a 256-byte *XPLine*. It operates on the following basis: after receiving a 64 byte write request, Optane's internal controller loads the corresponding (merged) 256-byte *XPLine* into the RMW buffer, provided that it is not already present. Subsequently, the data contained in the write request and the existing data in the *XPLine* are merged. Finally, the write-back of data from the buffer to actual flash memory is done proactively and periodically, depending on the state of the *XPLine*. Specifically, once the entire *XPLine* has been filled with new data, it is selected for write-back. Partially modified *XPLines* are retained in the buffer until they are randomly evicted from the RMW buffer.

In contrast to the read buffer, where we cannot control the placement of data, we have control over data placement in the RMW buffer, since writes bypass the cache and thus are not reordered (**Concept 3**). As a result, Xiang et al. (9) remark that preferably the writes should be merged together to form XPLine-sized writes. This, in turn, reduces contention at the RMW buffer, as these writes can be written to flash memory directly and do not occupy the RMW buffer space for a prolonged period of time.

Performance Metrics. To investigate whether individual (arbitrary-sized) writes are grouped into 256-byte *XPLine*-sized writes and whether these writes exhibit good locality, we introduce a new metric called the Inner-Sample Address Distance (ISAD). The ISAD metric (defined in Table 4.6) quantifies the locality of PMEM operations, penalizing those with poor locality. As we will see later when discussing the implementation of the *pmem-analyze* tool, a captured trace is divided into smaller segments called samples. Each sample

[1] Maximum RA: 256 bytes *XPLine*/64 bytes cache line = 4

is assigned an ISAD value to compare the locality between samples. An ISAD value close to 1.0 indicates that a sample yields poor locality, while a value close to 0.0 suggests good locality.

Metric	**Definition**
Metric 10: Inner-Sample Address Distance **Unit**: fraction within $[0.0, 1.0]$ **Goal**: A penalty is awarded to memory operation are not aligned sequentially. Output is the fraction of instructions that void sequential ordering, i.e. a value close to 1.0 indicates a random access heavy workload, whereas a value close to 0.0 indicates a sequential access pattern (**Concepts 1, 4** and **5**).	$$P = \sum_{n=1}^{s} \begin{cases} 1, & \text{if } \lvert addr^{cur} - (addr^{prev} + op^{size}) \rvert > 0 \\ 0, & \text{otherwise} \end{cases}$$ $$D = \frac{P}{s}$$ where s is the total number of reads, writes, and flushes within a sample, $addr^{cur}$ the address read/written at n, op^{size} the size of the previous operation, and $addr^{cur}$ the address at $n-1$.

Table 4.6: Performance metric to assess data locality

4.2.3 Implications of NUMA Accesses

Although our work is focused on improving performance transparency at the microarchitectural level, we found that the implications of remote (non-local) PMEM access are too significant to be excluded from the study. This is because these implications have a profound effect on performance in a NUMA setting.

Concept 6: Slow Cross-Socket Interconnect Degrades Throughput. Gugnani et al. (72) show that, in particular, high concurrency and large IO workloads suffer from performance degradation when accessing cross-socket PMEM. This degradation is the result of the limited bandwidth of cross-socket interconnects, especially in the case of Intel's *Ultra Path Interconnect* (UPI). UPI serves as a point-to-point interconnect in systems with a shared address space, such as the NUMA architecture (101). The UPI offers multiple data lanes and, on top of this, Intel utilizes a cache *snooping* protocol to distribute data across processors and ensure that data copies remain consistent. This cache-snooping protocol has two implications. First, Caheny et al. (62) show that cache coherency traffic within a NUMA system can consume 25% of the total bandwidth of the lane. Second, an issue that is very specific to PMEM is related to the phenomenon we already described in the discussion of the implications of caching (subsection 4.2.1): the bus snooping protocol essentially

transforms a sequential access pattern into a random one, further decreasing throughput and increasing latencies.

Performance Metrics. Defining appropriate performance metrics to assess remote PMEM is not within the scope of this study and, therefore, is considered future work. However, in our view, the findings presented by Kalia et al. (96) provide a valuable starting point, as they present concise methodologies (particularly in Section 4.3) on how to evaluate the performance of remote PMEM.

4.2.4 Conclusion

In this literature survey, existing literature has been reviewed to answer the following sub-question: *What are the micro-architectural performance-related idiosyncrasies of Intel Optane DCPMM?* We will now formulate the answer to this question:

Compared to a CPU, which accesses data at cache line (64 bytes) granularity, Intel Optane DCPMM stores data at 256-byte granularity, also known as an *XPLine*. This discrepancy makes data reads and writes originating from the CPU suspectable to read and write amplification, especially workloads that exhibit a lot of random accesses (**Concept 1**). Another performance idiosyncrasy is the overhead of CPU prefetching (**Concept 2**). In random workloads, where Optane accesses lack a strided/sequential access pattern, mispredictions increase. Consequently, the PMEM device spends valuable device resources on fetching data that will be disregarded anyhow. In the case of writes, this issue can be circumvented by bypassing the CPU cache (**Concept 3**).

The importance of data placement is demonstrated by **Concept 4** and **5**. We found that Optane DCPMM employs two 16 kB buffers to cache data in flash and to perform the aforementioned request merging. Ideally, reads and writes should be handled by these buffers. Therefore, the working set should be kept small to reduce contention at these buffers.

4.3 Implementation of *pmemanalyze*

Having gained a clear understanding of the micro-architecture performance idiosyncrasies of Intel Optane DCPMM, and defined suitable performance metrics to whether file systems conform to these peculiarities, we now discuss the implementation of *pmemanalyze*. To allow a comprehensive and structured discussion, we begin by describing its simplified control flow, which consists of four components, as shown in Figure 4.3.

In the following sections, we will elaborate on each component's implementation and design choices. As most of the code is written in C++, we divided the implementation into several classes (see Table 4.7), which we refer back to during the discussion. The source code is available through the *PMicroProfile* Git repository, as described in section 7.2.

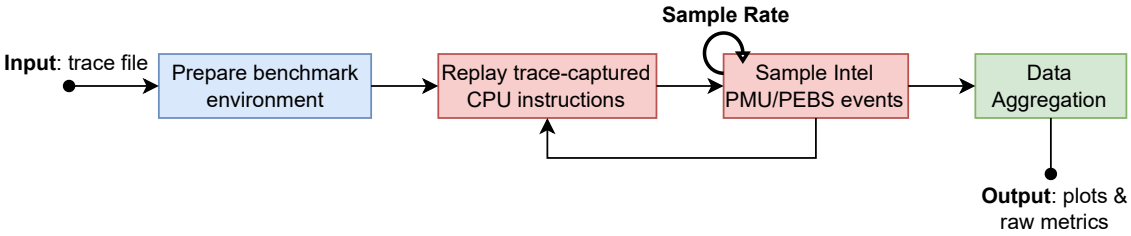


Figure 4.3: Simplified view of *pmemanalyze* control flow

Class	Purpose
`Trace`	Abstract Data Type (ADT) for an ingested trace file
`PMC`	Abstraction for the `perf_event` kernel API
`IO`	Wrapper for all assembly instructions, e.g. `movnti`
`BenchSuite`	Implements actual trace playback
`BenchExport`	Statistics gathering and Data Aggregation

Table 4.7: *pmemanalyze* C++ classes

4.3.1 Preparing a Benchmark Environment

In order to study the micro-architectural performance properties of a *pmemtrace* trace file, it is necessary to replay CPU instructions directly on Intel Optane DCPMM in such a way that other non-microarchitectural factors in the storage stack are excluded from the performance analysis. To achieve this, *pmemanalyze* defines a benchmarking environment where the trace file serves as the input (see Figure 4.3), and the output is an environment where captured instructions are replayed directly from the user space to the Optane DCPMM device without kernel involvement. The rest of this subsection outlines the setup process for this environment.

Ingesting a Trace File. The first step is to ingest the trace file. *pmemanalyze* accomplishes this by calling the `Trace::parse_trace` method, which decompresses its contents using the Apache Parquet library. Each row of the trace file is then read and converted into

a `TraceEntry` struct. These object instances are then stored together within a C++ vector (`std::vector`). As shown in Table 4.8, the `TraceEntry` struct contains multiple data fields, such as the instruction opcode, instruction name, instruction operation size, CPU affinity, the absolute virtual memory address at which the operation took place, and (if applicable) the payload.

Opcode (hex)	Name	Op. Size	CPU ID	Absolute Address	Write Data
0xC3 0F	movntq	8 bytes	0	0xDEADBEEF	0x40000000
0xE7 0F	movntdq	16 bytes	0	0xFEEDC0DE	0x00001245
0x66 0F AE	clflushopt	64 bytes	*	0xDEADBEEF	-
0x89	mov	8 bytes	0	0xFEEDC0DE	-

Table 4.8: Ingesting a trace: data fields

Establishing a Device DAX (*devdax*) Mapping. In order to replay traces without kernel interference, a Device DAX mapping (*devdax*) needs to be established. This mapping ensures that CPU instructions can be replayed directly without interference or software overhead from the kernel (80). Requesting a *devdax* mapping is done through the `mmap` system call. Listing 4.1 demonstrates how this is done in code. First, a file descriptor is opened using the path of the device (e.g. `/dev/pmem0`). Then, the `mmap` system call is invoked with this file descriptor as an argument, signaling the kernel to insert a Virtual Memory Area (VMA) inside the process address space that points directly to Optane's MMIO region. Additionally, the `madvise` and `mlock` system calls are invoked. Together, these provide a hint to the kernel that memory should not be swapped out. This is important as we envision that the occurrence of page faults could introduce noise in the benchmarking results.

```
1 int fd;
2 void *dax_area;
3
4 fd = open(this->pmem_device_loc.c_str(), O_RDWR); // Example:
    /dev/pmem0
5 dax_area = mmap(NULL, this->mem_size, PROT_READ | PROT_WRITE,
    MAP_SHARED, fd, 0);
6 madvise(dax_area, this->mem_size, MADV_WILLNEED);
7 mlock(dax_area, this->mem_size);
```

Listing 4.1: Allocating a devdax region using mmap, error handling omitted. The `mem_size` variable is set the size of the mapping, by default 28 GiB, which is the same amount of emulated PMEM within the pmemtrace VM (function: `BenchSuite::allocate_pmem_area()`).

The final step in setting up the benchmarking environment is to transform all *absolute addresses* contained in `TraceEntry` objects (see Table 4.8) so that they point to their corresponding locations within the allocated `devdax` region, i.e. *relative addresses*. These relative addresses are obtained in two steps, as illustrated in Listing 4.2.

First, we calculate the offset within the `devdax` region by subtracting the absolute address of the trace entry (`entry.abs_addr`) from the `dev_addr` value. In the context of Listing 4.2, the `dev_addr` value represents the start of the MMIO region where the emulated PMEM device was mapped during tracing. Then, to obtain the corresponding relative address, we add this offset to the starting address of the `devdax`-mapped memory region (i.e., `dax_area`).

```
1 for (TraceEntry &entry : this->trace_file) {
2     entry.addr_offset = entry.abs_addr - static_cast<char*>
          dev_addr;
3     entry.dax_addr = static_cast<char*>(dax_area) +
          entry.addr_offset;
4 }
```

Listing 4.2: Transformation of device absolute addresses into 'daxdax' relative addresses (function: `BenchSuite::run()`).

4.3.2 Replaying Captured CPU Instructions

Having ingested and transformed the trace into a suitable format for performance analysis, our focus now shifts to how *pmemanalyze* implements trace replay.

The process of replaying a trace involves three steps, which are described below.

1. First, we initialize a set of `pthreads`, which size is equal to the number of CPU cores in the system. Each thread's execution is pinned to a CPU core, ensuring that the PMU and PEBS events are applicable only to the workload that was assigned to that particular CPU. Subsequently, each thread is assigned trace entries for which condition $thread_{id} == entry_{cpuid}$ is satisfied. It is important to note that although *pmemtrace* currently only supports single-core tracing, we have decided to include multicore support for replaying traces in order to facilitate future work;

2. Once all threads are initialized, each thread starts executing its workload by invoking the `BenchSuite::replay_trace(...)` function (refer to Listing 4.3). This function iterates over the assigned `TraceEntry` instances and executes the corresponding CPU instructions. This is accomplished by calling the relevant C function from the Intel Intrinsics C library (102), such as `_mm_stream_si64` for storing a 64-bit integer with a non-temporal hint;

3. Furthermore, as illustrated in Figure 4.3, we employ our `PMC` library to capture Intel PMU and PEBS event counters at a user-preselected sampling interval. The implementation details of this library will be covered in the next section.

```
for (const TraceEntry& entry : trace_file) {
    switch (entry.op) {
    case TraceOperation::READ: // omitted ...
    case TraceOperation::WRITE:
        switch (entry.opcode) {
            case 0xA4: // single byte.
                write_mov_8(entry, is_sampling, (*cur_sample));
            case 0xC30F: // MOVNTI - 4/8 bytes
                write_movntq_64(entry, is_sampling, (*cur_sample));
                break;
            case 0xE70F: // MOVNTDQ - 16 bytes
                write_movntq_128(entry, is_sampling,
                    (*cur_sample));
                break;
        }
        break;
    // More switch cases, i.e. CLFLUSH, MFENCE, SFENCE, LFENCE.
    }
}
```

Listing 4.3: Simplified view of instruction replaying (function: `BenchSuite::replay_trace(...)`)

4.3.3 Capturing Intel PMU and PEBS Hardware-Performance Events

As mentioned earlier in the background section on hardware performance counters (section 2.3), Intel-based systems feature a Performance Monitoring Unit (PMU). In this section, we discuss how we employ Linux's `perf_event` kernel API to implement all the performance metrics we designed based on the concepts discussed in the literature study.

Raw access to performance events using `perf` kernel API. The Linux ecosystem provides application developers with tools to profile applications, which the CLI tools *perf* (75) and *eBPF* (103) being the most well-known examples. However, the use of these tools has drawbacks. A drawback of the *perf* CLI tool is that it measures the execution time of an entire program, making it challenging to obtain precise micro-architectural performance measurements for specific sections of the source code. On the other hand, the eBPF profiling framework can annotate particular sections of code for accurate measures, but it lacks direct access to raw performance counters. This is because it still relies on the *perf* CLI tool as an intermediary layer [1].

To overcome this limitation, we developed a small performance profiling library (in user space) called *PMC*. This library utilizes the `perf_event` kernel API (104) to directly read raw Intel PMU core, uncore, and off-core performance counters/events as listed in section 4.3.3. Note that these performance events are essential to implement the performance metrics defined in our literature study.

Performance Event	**Source**
`UNC_M_CLOCKTICKS`	iMC (uncore event)
`UNC_M_PMM_RPQ_INSERTS`	iMC (uncore event)
`UNC_M_PMM_WPQ_INSERTS`	iMC (uncore event)
`UNC_M_PMM_RPQ_OCCUPANCY`	iMC (uncore event)
`UNC_M_PMM_WPQ_OCCUPANCY`	iMC (uncore event)
`L3_MISS_LOCAL_DRAM.ANY_SNOOP`	iMC (offcore event)
`PMM_HIT_LOCAL_PMM.ANY_SNOOP`	iMC (offcore event)
`MEM_LOAD_RETIRED.LOCAL_PMM`	on-core
`MEM_LOAD_RETIRED.L3_MISS`	uncore
`MEM_LOAD_RETIRED.ALL_STORES`	on-core
System Hardware Timer	HPET

Table 4.9: Performance events *pmemanalyze*: on-core, uncore, and off-core

We have included several helper functions in the performance library that facilitate creating, attaching, and removing performance *probes* during program execution. A comprehensive list of all available functions can be found in Appendix C; see chapter 8.

In PMC, we again make the distinction between on-core, off-core, and uncore events:

[1] Linux Mailing List Discussion: initialising/reading raw (hardware) performance counters BPF_PERF_ARRAY, by Andrew Nisbet (04/06/2023), `https://lore.kernel.org/bpf/ZC7ICD%2FvAKdtvopd@krava/T/`

- **On-core and offcore events**: On-core and off-core CPU performance events are created using the `perf_event_open` system call [1]. This system call requires the initialization of a `perf_event_attr` structure, which should be configured according to the respective *event selector* and *umask* constants listed in the Intel Architecture Manual (55). An example of creating a performance event, in this case `MEM_LOAD_RETIRED.L3_MISS`, is provided in Listing 4.4. Note that for off-core events, the `MSR_OFFCORE_RSP` constant should also be set according to the value listed in the aforementioned Intel Architecture Manual.

  Method definition: `bool add_offcore_probe(const unsigned int event_id, const int pid, const unsigned long msr = 0x0)`

```
1  struct perf_event_attr pe;
2  int fd;
3  memset(&pe, 0, sizeof(struct perf_event_attr));
4
5  pe.type = PERF_TYPE_RAW; // Request raw performance counter.
6  pe.size = sizeof(struct perf_event_attr);
7  pe.config = 0xD120; // MEM_LOAD_RETIRED.L3_MISS ->
       EventSel=D1H UMask=20H
8  pe.sample_type = PERF_SAMPLE_IDENTIFIER;
9  pe.disabled = 1; // Disable for now, enable when we actually
       profile some code.
10
11 fd = perf_event_open(&pe, pid, 0, -1, 0); // Perform syscall.
```

Listing 4.4: Creation of `MEM_LOAD_RETIRED.L3_MISS` performance event using `perf_event_open`.

- **Uncore events**: the implementation for uncore events differs slightly from the on-core and off-core events. Recall that uncore events occur at the CPU package level and are not specific to individual cores. Consequently, allocating a single perf event per software thread is inadequate. Instead, for uncore events happening on Integrated Memory Controllers (iMCs) (see section 4.3.3), we need to create multiple perf events, one for each iMC within the CPU package. This is especially important when using Intel Optane DCPMM, as memory requests are handled by not just one iMC, but multiple. The mechanism to acquire a handle for each iMC is provided through the Linux `/sys/bus/event_source/devices/uncore_imc_*` interface and implemented in the `PMC::init()` method. To create an iMC performance probe, one

[1] Documentation: `https://man7.org/linux/man-pages/man2/perf_event_open.2.html`

can use the `add_imc_probe(const unsigned int event_id)` method, which initializes the specified event on each iMC. To obtain the output of an iMC probe, the PMC library reads the raw performance value of each iMC and aggregates those values to obtain the system-wide representative output.

Method definition: `bool add_imc_probe(const unsigned int event_id)`

The final step is to aggregate all the gathered data in a suitable format for export and analysis. This is achieved by the `BenchExport::export_io_stat` function, which exports the data to a comma-separated CSV file. One CSV file contains more than 30 columns, each representing a raw captured counter or another derived metric or heuristic.

4.4 Conclusion

Through a comprehensive literature study on the performance characteristics of Intel Optane Persistent Memory, we formulated performance metrics that served as the foundation for the design of *pmemanalyze* tool. This tool replays *pmemtrace*-captured traces to perform performance analysis. The identified performance metrics are constructed by combining relevant hardware performance counters exposed by Intel's Performance Monitoring Unit (PMU).

5

Experimental Evaluation

In this section, we will evaluate the effectiveness of the *PMicroProfile* framework in identifying performance bottlenecks. We answer this research question by comparing the performance of two file systems: ext4-DAX (49), and SplitFS (19). Specifically, we collect access traces for the Filebench (105, 106) benchmark for both file systems. We will then replay these traces and perform a performance analysis. The results of this evaluation form the answer to sub-question **RQ4** posed in the Introduction.

5.1 Experiment Design

In this section, we will discuss the setup of the experiments. First, we detail the hardware setup. Second, we explain the workflow followed during the experiments.

For tracing, we utilize the same hardware setup described in the tracing-related artifact (section 7.1). Specifically, we use a QEMU virtual machine running Ubuntu 20.24 with Linux kernel version `5-4.232`. QEMU is configured so that it emulates 28 GiB Persistent Memory. For replaying the trace, we used an Intel Xeon Silver 4215 CPU (turbo disabled to avoid indeterminism (107)) with 72 GiB of RAM, and four Intel Optane DCPMM 100 series DIMMs (model NMA1XBD512GQS, capacity 512 GB). The technical specifications can be found on Intel's product page[1], and are summarized in Table 5.1. Each PMEM DIMM is configured as *non-interleaved*, meaning they are treated as four independent storage devices within the operating system. This is to ensure that all accesses hit a single DIMM. The performance evaluation of interleaved Optane DCPMM is considered future work. Finally, we use the GCC compiler version `9.4.0` to compile *pmemanalyze*.

[1]Technical Specifications: `https://ark.intel.com/content/www/us/en/ark/products/190348/intel-optane-persistent-memory-100-series-128gb-module.html`

Sequential Bandwidth (for all R/W ratios)	1 GB/s
Random Bandwidth (for all R/W ratios)	1 GB/s
Sequential Latency - Read (up to)	1 ns
Random Latency - Read (up to)	1 ns

Table 5.1: Technical specifications of Intel Optane DCPMM 100 series provided by Intel. Bandwidth is measured at the host. It is unspecified how latency is expressed. We assume it is the best-case latency and measured at the device, excluding the overhead of the Integrated Memory Controller and caches.

The experimental workflow, depicted in Figure 5.1, consists of three components: a benchmarking environment, data gathering, and data processing.

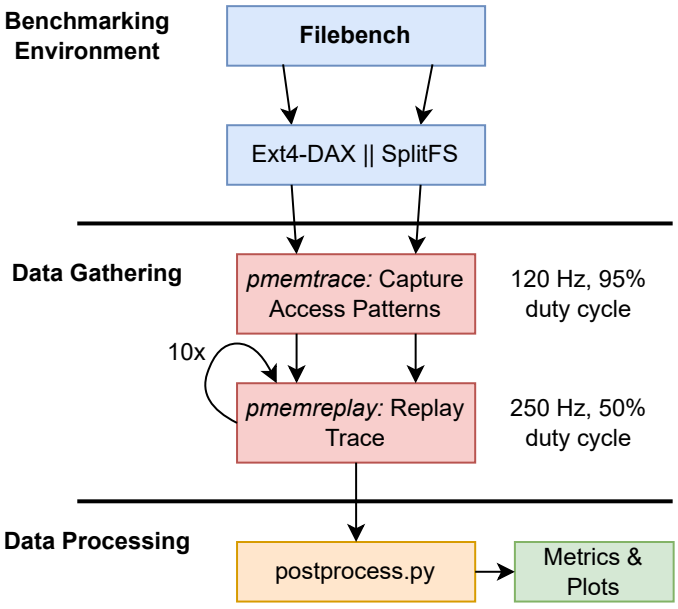


Figure 5.1: Experimental workflow

In the benchmarking environment, we use pre-selected Filebench workloads (specifically, an adapted version of *varmail* that performs small-grained read and writes, relevant artifact in section 7.2) along with the Ext4-DAX and SplitFS file systems.

To collect access traces, we use a sampling-based data collection strategy with a sampling interval of 120 hertz and a duty cycle of 95%. After collecting the traces, they are replayed and performance data is collected at a 250 hertz frequency (50% duty cycle). We have selected this frequency to minimize the overhead associated with performance sampling, while still obtaining a significant amount of data.

It is important to note that we use the same sampling settings for both file systems to ensure unbiased comparisons. Additionally, each trace is replayed at least 10 times. This is done for two reasons. First, the sizes of traces (after decompression) quickly exceed multiple gigabytes and, by design, must be preloaded into the main memory, which is sparse. Second, replaying traces multiple times enables us to verify whether performance behavior remains constant or varies over time, which in turn helps to improve the accuracy and validity of our research.

Finally, the collected (raw) performance data is processed using a postprocessing script (`postprocess.py`). This script calculates the predefined metrics discussed earlier and generates plot images.

5.2 Findings

In the following sections, we will present the results of the experiments. We will categorize findings into three areas: "Workload Characterization", "Data Bandwidth and Access Latencies", and "Cache Interaction".

5.2.1 Workload Characterization

Before discussing specific performance details, we first compared the access patterns of both file systems using an adapted version of the *varmail* benchmark. Specifically, to stress Optane's byte-addressable capabilities, we modified the Filebench varmail configuration file to use the smallest I/O size supported by Filebench, 1 kB.

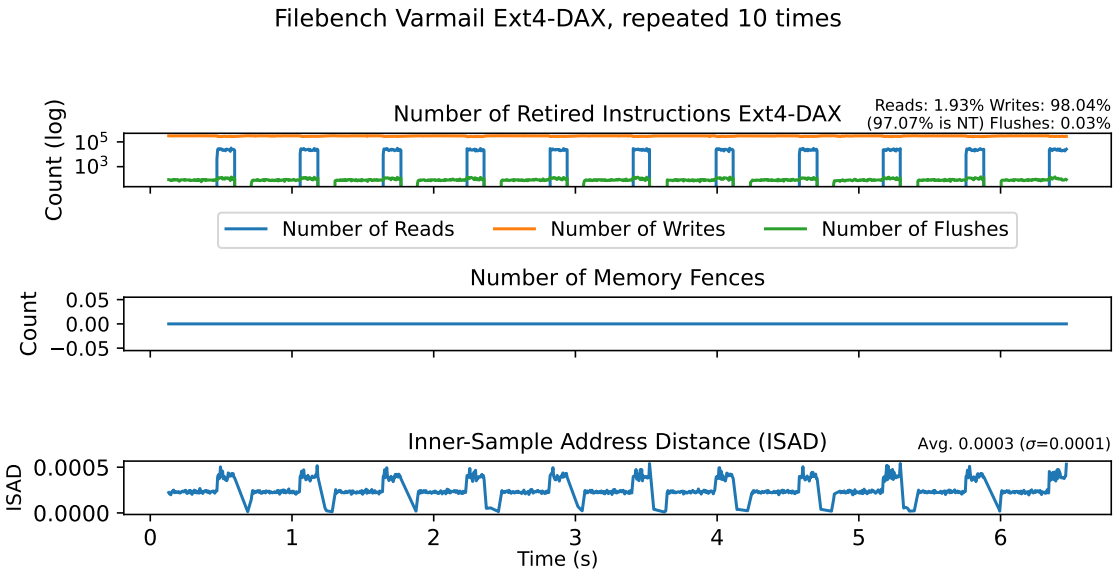
Figure 5.2 displays the access patterns for both file systems obtained by *pmemanalyze*. The first graph displays the number of executed (retired) CPU instructions over time, the second graph the number of memory fences, and the third graph the ISAD locality metric (Metric 10). As the execution time of a single trace file replay is very short (approximately 0.5 seconds), we replay each captured trace at least ten times to gather more run-time statistics. Based on this figure, we can make two observations:

1. **Usage of non-temporal (NT) write instructions**: Ext4-DAX and SplitFS use non-temporal move instructions to bypass the cache. This is evident from the fraction of writes classified as non-temporal, 80.43% for Ext4-DAX and 99.86% for SplitFS (as shown in the top right of the figure). However, there is a difference between SplitFS and Ext4-DAX in terms of memory fence usage. In the case of Ext4-DAX, there are no memory fences during the entire execution, indicating that the ordering

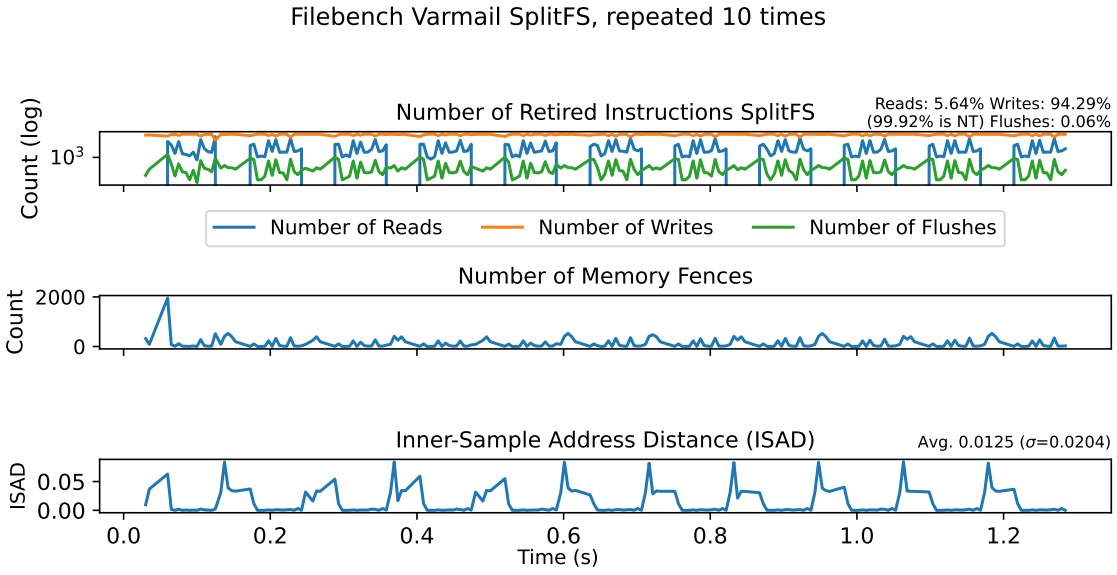
of instructions based on the issue time is not enforced. Although the lack of fence instructions may offer performance benefits, it can cause inconsistencies in the data if the system crashes (see Background subsection 2.1.2).

2. **Poor data locality SplitFS**: Based on the Inner-Sample Address Distance (ISAD) metric (defined in Metric 10), we can derive that, for this specific workload, SplitFS (average ISAD: 0.1028) exhibits a lower data locality compared to Ext4-DAX (average ISAD: 0.0004). Although we do not have enough evidence to prove this, we hypothesize that this difference may be attributed to how Ext4-DAX groups data into 4 KiB blocks. This grouping results in a more sequential access pattern at the PMEM device at the cost of additional read and write amplification at the *file system level* (not at the microarchitectural level). Another observation is related to mixed read-write access patterns. In these cases, the data locality degrades, as spikes in ISAD values indicate.

In summary, both file systems use non-temporal machine instructions to bypass the cache. However, as indicated by the absence of memory fences, Ext4-DAX does not enforce the ordering of instructions. Furthermore, SplitFS exhibits a lower data locality than Ext4-DAX for this workload.



(a)



(b)

Figure 5.2: Ext4-DAX (a) and SplitFS (b) Varmail access patterns. The figure reports the number of executed/retired CPU instructions over time, the number of memory fences (`mfence`, `sfence`, and `lfence`), and the ISAD metric for quantifying data locality over time (value close to 1 implies poor locality).

5.2.2 Data Bandwidth and Access Latencies

In this section, we discuss the microarchitectural performance of both file systems. We do this using four metrics, read and write amplification (Metric 2 and Metric 3), bandwidth (Metric 4), and instruction latency (Metric 5). The results are presented in Figure 5.4. Please note that for the SplitFS file system, we increased the number of replay iterations to 50 to increase run time and thus collect more data. Additionally, note that the axes of the subfigures scale differently.

Once again, we present our findings in an enumeration:

1. **Bandwidth SplitFS approaches Optane's peak bandwidth**: SplitFS achieves a combined read/write bandwidth of 0.959 GB/s, while Ext4-DAX achieves 0.945 GB/s. To validate how close these values are to Optane's peak bandwidth, we measure the raw peak bandwidth on a live system. Initially, we attempted to use the *Memory Latency Checker* (MLC) tool (108) provided by Intel, which is designed for measuring the latency and bandwidth of DIMM devices like Optane Persistent Memory. Unfortunately, the MLC tool proved unreliable, reporting an impossible bandwidth of 1.2 TB/sec. Interestingly, this bug is also reported in an article published by Intel on testing Optane's peak bandwidth [1]. Consequently, to evaluate Optane's bandwidth, we developed a small benchmarking program [2]. This program measures the peak read and write bandwidth by performing a simple 8 GB *memcpy* operation to/from a DAX-mapped region. Using this benchmark, we achieve a read bandwidth of 1.03 GB/s and a write bandwidth of 0.94 GB/s, matching the advertised speed of 1 GB/s specified for Optane (Table 5.1). Therefore, we can conclude that both SplitFS and Ext4-DAX are very close to the advertised peak bandwidth of Optane.

2. **Device access latencies**: To collect access latency data, we use the hardware High-Precision Event Timer (HPET) to sample latencies every $1,000$ read and write operations. Based on the results presented in Figure 5.4, we draw two main findings. First, Ext4-DAX exhibits a relatively high average write latency of 26.21 nanoseconds, whereas SplitFS achieves a lower write latency of 21.41 nanoseconds. Both file systems display similar read latencies.

[1] How to Test the Performance of Intel® Optane™ Persistent Memory: `https://www.intel.com/content/www/us/en/support/articles/000055898/memory-and-storage/intel-optane-persistent-memory.html`

[2] Available in Git repository: `/peak_bw`

Second, SplitFS shows outliers in instruction read latency, reaching up to 500 nanoseconds. This behavior aligns with a phenomenon described by Yang et al. (10) in their study. They suspect that this effect is due to wear leveling on the device. However, in their case, the latency jump is more significant, that is, at the microsecond level. Additionally, if this theory were accurate, both SplitFS and Ext4-DAX should experience a similar increase in latency, which is not the case. To increase the validity of our research, we conducted longer (two-minute) runs, of which the results are shown in Appendix D. Once again, we observe latency patterns where only SplitFS exhibits distinct latency peaks. Consequently, this latency peculiarity is SplitFS-specific.

Given the lack of correlation between latency spikes and other performance metrics, it is challenging to provide a definitive explanation for this behavior. It is worth considering that this issue may be unrelated to Optane or specific to our system configuration. Further work is necessary to eliminate these factors and narrow down the underlying cause.

3. **Read and write amplification**: Both file systems experience modest read amplification (Ext4-DAX: ≈ 1.47, SplitFS: ≈ 1.72) and minor write amplification (Ext4-DAX: ≈ 1.00, ≈ 1.15). According to our analysis, random reads contribute to increased read amplification, whereas the write amplification in SplitFS results from one machine instruction that it executes, `movntps`. Now, we zoom in on both cases.

   Where both file systems perform mostly sequential writes, their reads are more random, especially in the case of SplitFS. This observation is based on the increase in Inner-Sampling Address Distance (ISAD) (see Figure 5.2) in read-heavy sections. This random access pattern makes it susceptible to the problem described in Concept 4 of the literature study, in which the CPU suffers from read amplification due to poor internal buffering performance at the Optane device.

   The issue of write amplification in SplitFS can be attributed to the type of machine instructions it executes. Recall that PMEM file systems use non-temporal machine instructions to write through the cache. To accommodate for different-sized writes, the instruction set supports multiple variants. For example, `movntps` can be used to store a 128-bit, 256-bit, or 512-bit number, while the `movnti` instruction is used to store a 32 or 64-bit integer. In the case of SplitFS, it primarily uses the 512-bit `movntps` instruction [1], whereas Ext4-DAX uses the *movntq* instruction. Comparing

[1] `movntps` definition: `https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html#text=movntps&ig_expand=6648,6650,6649`

the write amplification of these instructions (see Figure 5.3), we observe that the `movntps` instruction results in approximately twice the write amplification compared to the `movntq` instruction used by Ext4-DAX.

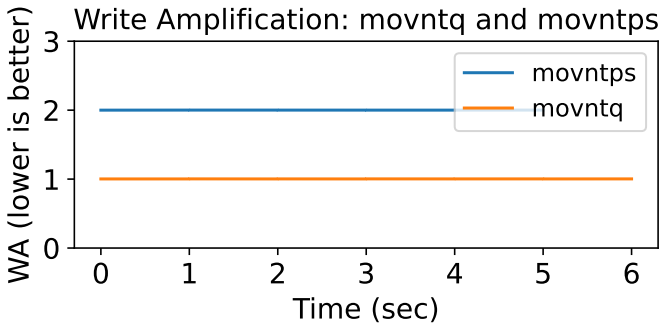
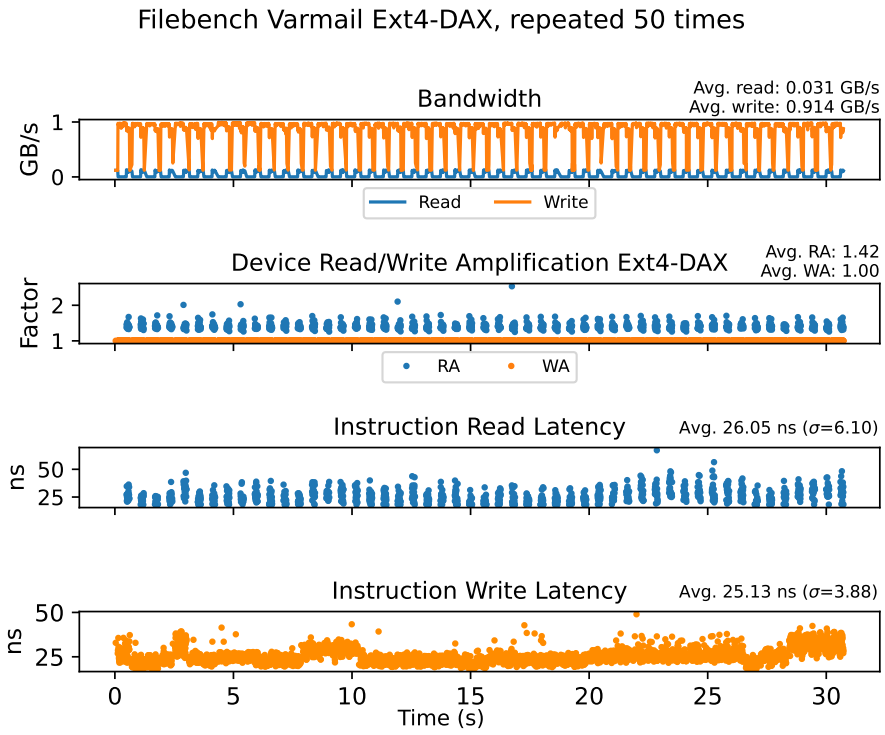
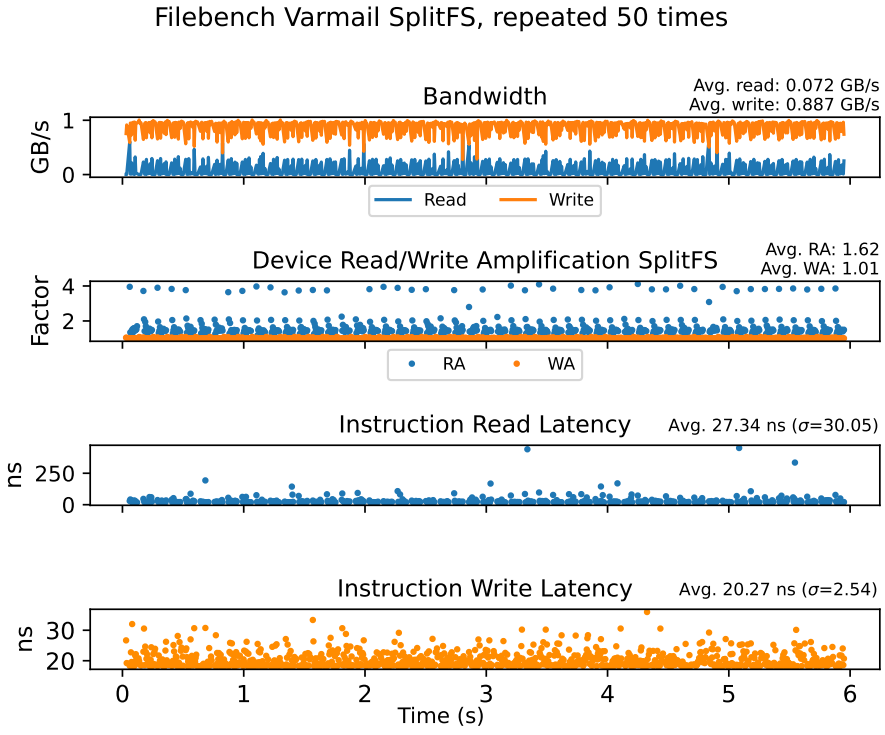


Figure 5.3: Write Amplification (lower is better) `movntq` and `movntps` Machine Instructions

In summary, SplitFS outperforms Ext4-DAX in terms of the bandwidth observed between the CPU and Optane DCPMM. However, SplitFS also exhibits high read and write amplification levels compared to Ext4-DAX, although this does not directly affect the achieved bandwidth. Another unique characteristic of SplitFS is sporadic jumps in access latencies. Unfortunately, we were unable to find the exact causes of these spikes. This could be part of future work.



(a)



(b)

Figure 5.4: Ext4-DAX (a) and SplitFS (b) Varmail Performance Metrics: bandwidth (higher is better), read/write amplification (lower is better), and instruction latency (lower is better)

5.2.3 Cache Interaction

The last finding relates to the interaction between the CPU caches, DRAM, and Intel Optane DCPMM. Figure 5.5 displays three performance metrics: the number of *direct loads* (i.e. loads that bypass the cache) and Last-Level Cache (LLC) misses for both DRAM and Optane. It is evident that the number of direct PMEM loads correlates with the number of PMEM LLC misses. This is expected behavior, as the caches cannot accommodate all PMEM reads, initiating 64 cache line-sized load operations. Additionally, the DRAM LLC misses follow a similar trend. Since the entire trace file is preloaded into DRAM and cannot fit fully in the CPU caches, it leads to DRAM fetches.

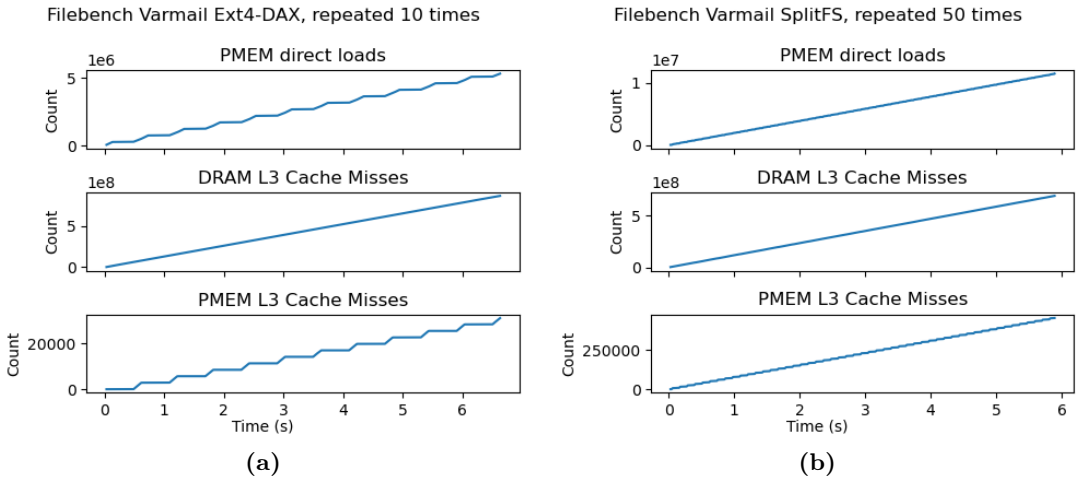


Figure 5.5: Ext4-DAX (a) and SplitFS (b) Varmail: DRAM and Intel Optane DCPMM interaction

5.3 Limitations and Discussion

This section discusses the limitations of *pmemanalyze* and the experimental evaluation associated with it. We will specifically discuss two limitations: the challenges in finding micro-optimizations and the accuracy of the experimental analysis.

Finding Micro-Optimizations. While our experimental evaluation allowed us to identify bottlenecks, we have not yet developed concrete, implemented micro-optimizations for PMEM file systems. There are three reasons for this. First, there is no consensus about the 'best case' performance for Intel Optane DCPMM with respect to measurable factors such as latency and throughput. The measures provided are often overly optimistic or specific to a particular hardware configuration. Consequently, it is challenging to determine whether there is still room for performance improvement. For example, in the case of SplitFS, it achieves (close to) the advertised bandwidth of Intel Optane DCPMM.

Second, although there are quite some performance counters for Intel Optane DCPMM (approximately 60 counters [1]), many of them are redundant and undocumented. For example, the number of direct PMEM loads is inversely proportional to the number of cache misses. We had to extract these findings ourselves through a lot of experimentation. Better documentation would have been beneficial and had speeded up the implementation phase.

Lastly, we prioritized enhancing the precision of both the *pmemtrace* and *pmemanalyze* tools. For example, if the access traces captured by *pmemtrace* were inaccurate, it would compromise the validity of the accompanying performance analysis.

Accuracy of Experimental Analysis. The second limitation concerns the generalizability of the experimental evaluation. Ideally, we would have liked to replay longer traces. However, the approach of replaying captured access trace for performance analysis required storing each machine's instruction for replay. This level of precision demands a significant amount of storage. To illustrate, a 500 MiB trace file translates to approximately 1 GiB in PMEM read/write traffic. Considering that the peak bandwidth of Optane is around 1 GB/s, a 120-second run would require a 60 GiB-sized trace file. Therefore, we believe that future efforts should focus on reducing storage space requirements, e.g. by merging memory adjacent instructions for compression.

[1] `https://perfmon-events.intel.com/cascadelake_server.html` (search term: 'PMM')

5.4 Conclusion and Future Work

In this chapter, we highlighted the importance of considering CPU microarchitectural overhead in performance analysis, as a significant portion of overall performance (38% to 78%) can be attributed to I/O operations occurring at the microarchitectural level (Figure 4.1). To address this, we designed and implemented *pmemanalyze*. This tool replays file system access traces and quantifies the microarchitectural performance overhead using predefined based on a literature study (section 4.2) of Intel Optane DC Persistent Memory microarchitecture idiosyncrasies.

An experimental evaluation of *pmemanalyze* demonstrated its effectiveness in identifying performance bottlenecks in two file systems: Ext4-DAX and SplitFS. In this evaluation, we extracted multiple findings, such that SplitFS achieves a higher bandwidth than Ext4-DAX, although it suffers from sporadic jumps in write access latencies. However, translating these findings into in-code micro-optimizations remains future work.

Other future research directions include investigating the impact of cross-socket Optane traffic and exploring performance effects in multicore environments. Associated research questions include: "What is the impact of cross-socket (NUMA) Optane traffic in PMEM file systems?" or "Can the PMicroProfile framework pinpoint microarchitectural degrading events in multi-core environments?".

6

Conclusion

In this work, we designed and implemented a framework, *PMicroProfile*, to conduct micro-architectural performance analysis of Persistent Memory file systems. Our work addresses an open question of how to evaluate the performance of Persistent Memory file systems close to the hardware. We designed and implemented two tools to achieve this: *pmemtrace* and *pmemanalyze*. The *pmemtrace* tool captures file system access patterns at the bottom of the software stack, while *pmemanalyze* replays these traces to quantify the performance of file systems. We will now answer each of the research questions.

RQ1: What are the performance-related idiosyncrasies of Intel Optane DCPMM at the CPU micro-architectural level? A literature study revealed that a significant idiosyncrasy is a difference in data access granularity between the CPU (64 bytes) and Intel Optane DC Persistent Memory (256 bytes). Due to this difference, read and write requests initiated by the CPU are susceptible to up to 4× read or write amplification (*idiosyncrasy 1*). To mitigate this overhead, it is crucial to group data requests into 256-byte accesses when possible (*idiosyncrasy 2*).

Another idiosyncrasy found in the literature is related to poor CPU prefetching performance. CPUs proactively load data into on-die caches. However, when the Optane device accesses lack a sequential access pattern, resulting in increased mispredictions, the Optane device spends valuable device resources on fetching data that will be disregarded anyhow. This issue can be mitigated for write operations by bypassing the CPU cache (*idiosyncrasy 3*).

Lastly, data placement and locality place an important role. Ideally, the working set should be kept small to minimize contention at Optane's internal buffers (*idiosyncrasy 4*).

RQ2: How to design a tool that can trace the access patterns of the PMEM file system at a microarchitectural level? To capture the PMEM-related accesses patterns at the level of the CPU microarchitecture, we designed and implemented a tracing methodology that logs PMEM-affiliated machine instructions executed in user and kernel space. Specifically, we extended the existing Linux in-kernel tracing infrastructure to generate a Memory Management Unit 'page fault' for every PMEM access, signaling the kernel of all incoming PMEM requests. This tracing methodology successfully captures all PMEM traffic, which is crucial for our analysis. However, it comes at the cost of significant overhead. In the worst-case scenario, run times can increase by approximately 300× (see Table 3.2). To limit this overhead, we have included support for sampling-based data collection. Further work should focus on decreasing this overhead.

RQ3: How to design a tool that quantifies the performance of PMEM file systems using its access patterns? We implemented a separate tool called *pmemanalyze*. This tool replays the captured machine instructions in a *pmemtrace*-captured trace file directly from user space. Meanwhile, we implemented a library called *PMC* that leverages the *perf* event kernel API to gain access to the Intel PMU and PEBS raw hardware performance counters. By combining these raw counter values, we derived metrics that assess whether file systems comply with the recommendations formulated in research question **RQ1**.

RQ4: Can the PMicroProfile framework pinpoint microarchitectural performance degrading events to define new file system micro-optimizations? We used *pmemanalyze* to evaluate the access traces of two file systems, namely Ext4-DAX and SplitFS. This analysis showed that SplitFS is more susceptible to write amplification. Another significant finding is that Ext4-DAX sacrifices stronger crash-consistency guarantees in exchange for more performance. Such findings demonstrate the ability of PMicroProfile to identify microarchitectural events. However, it is important that due to our primary focus on enhancing the accuracy and capabilities of the *pmemtrace* and *pmemanalyze*, implementing new micro-optimizations based on these findings has not been performed yet, thus is part of future work.

References

[1] David Reinsel, John Gantz, and John Rydning. **The Digitization of the World from Edge to Core**. 2018. 1

[2] Alexandru Iosup, Fernando Kuipers, Ana Lucia Varbanescu, Paola Grosso, Animesh Trivedi, Jan Rellermeyer, Lin Wang, Alexandru Uta, and Francesco Regazzoni. **Future Computer Systems and Networking Research in the Netherlands: A Manifesto**, May 2022. arXiv:2206.03259 [cs]. 1, 2

[3] Yuhui Deng. **What is the future of disk drives, death or rebirth?** *ACM Computing Surveys*, **43**(3):23:1–23:27, April 2011. 1

[4] Chris Mellor. **SSDs will crush hard drives in the enterprise, bearing down the full weight of Wright's Law**, January 2021. 1

[5] **IDC FutureScape: Top 10 Predictions for the Future of Digital Infrastructure**, December 2022. 2

[6] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. **Persistent Memory I/O Primitives**. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*, DaMoN'19, pages 1–7, New York, NY, USA, July 2019. Association for Computing Machinery. 2, 23, 27, 53, 55

[7] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. **Basic Performance Measurements of the Intel Optane DC Persistent Memory Module**, August 2019. arXiv:1903.05714 [cs]. 2, 3, 14

[8] Jianyong Zhang, Anand Sivasubramaniam, Qian Wang, Alma Riska, and Erik Riedel. **Storage performance virtualization via throughput and latency control**. *ACM Transactions on Storage*, **2**(3):283–308, August 2006. 2

[9] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. **Characterizing the performance of intel optane persistent memory: a close look at its on-DIMM buffering**. In *Proceedings of the Seventeenth European Conference on Computer Systems*, EuroSys '22, pages 488–505, New York, NY, USA, 2022. Association for Computing Machinery. 2, 23, 24, 27, 55, 56, 57, 60, 61

[10] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. **An Empirical Guide to the Behavior and Use of Scalable Persistent Memory**. pages 169–182, 2020. 2, 3, 23, 24, 27, 53, 55, 56, 57, 60, 77

[11] Wiebe van Breukelen. **Persistent Memory File Systems: A Survey**, January 2023. 2, 3, 4, 7, 10, 16, 17, 19, 53, 54

[12] Takahiro Hirofuchi and Ryousei Takano. **A Prompt Report on the Performance of Intel Optane DC Persistent Memory Module**. *IEICE Transactions on Information and Systems*, **E103.D**(5):1168–1172, May 2020. arXiv:2002.06018 [cs]. 2

[13] Xinyang, Song and Sihang Liu. **Persistent Memory – A New Hope**, September 2022. 3

[14] Tobias Mann. **Why Intel killed its Optane memory business**. 3

[15] Lawrence Benson, Marcel Weisgut, and Tilmann Rabl. **What We Can Learn from Persistent Memory for CXL**. Technical report, Gesellschaft für Informatik e.V., Bonn, 2023. 3

[16] Minseon Ahn, Andrew Chang, Donghun Lee, Jongmin Gim, Jungmin Kim, Jaemin Jung, Oliver Rebholz, Vincent Pham, Krishna Malladi, and Yang Seok Ki. **Enabling CXL Memory Expansion for In-Memory Database Management Systems**. In *Data Management on New Hardware*, pages 1–5, Philadelphia PA USA, June 2022. ACM. 3

[17] David Boles, Daniel Waddington, and David A. Roberts. **CXL-Enabled Enhanced Memory Functions**. *IEEE Micro*, **43**(2):58–65, March 2023. Conference Name: IEEE Micro. 3

[18] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. **Design of ion-implanted MOSFET's with very small physical dimensions**. *IEEE Journal of Solid-State Circuits*, **9**(5):256–268, October 1974. Conference Name: IEEE Journal of Solid-State Circuits. 3

[19] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. **SplitFS: reducing software overhead in file systems for persistent memory**. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 494–508, Huntsville Ontario Canada, October 2019. ACM. 3, 5, 9, 10, 20, 21, 51, 71

[20] Jian Xu and Steven Swanson. **NOVA: A Log-structured File System for Hybrid {Volatile/Non-volatile} Main Memories**. FAST '16, pages 323–338, 2016. 3, 6, 51

[21] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. **Strata: A Cross Media File System**. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 460–477, Shanghai China, October 2017. ACM. 3, 5, 6

[22] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. **Better I/O through byte-addressable, persistent memory**. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles - SOSP '09*, page 133, Big Sky, Montana, USA, 2009. ACM Press. 4, 18

[23] Ruibin Li. **ctFS: Replacing File Indexing with Hardware Memory Translation through Contiguous File Allocation for Persistent Memory**. *20th USENIX Conference on File and Storage Technologies (FAST 22)*, 2022. 4, 5, 6, 15, 18, 22

[24] Youmin Chen. **Kuco: Scalable Persistent Memory File System with Kernel-Userspace Collaboration**. *Proceedings of the 19th USENIX Conference on File and Storage Technologies.*, February 2021. 4, 5, 18, 21

[25] Xiaojian Wu, Sheng Qiu, and A. L. Narasimha Reddy. **SCMFS: A File System for Storage Class Memory and its Extensions**. *ACM Transactions on Storage*, **9**(3):7:1–7:23, August 2013. 4, 6, 12, 14, 15, 18

[26] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. **Aerie: flexible file-system interfaces to storage-class memory**. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1–14, New York, NY, USA, April 2014. Association for Computing Machinery. 5, 15, 20

[27] Finn de Ridder, Animesh Trivedi, and Razavi, Kaveh. **UFS: Perserving Isolation when Unifying File Systems With Virtual Memory**. Unpublished. 5, 6, 22, 51

[28] Chloe Alverti, Vasileios Karakostas, Nikhita Kunati, Georgios Goumas, and Michael Swift. **DaxVM: Stressing the Limits of Memory as a File Interface**. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 369–387, October 2022. 5, 6, 22

[29] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. **PMFS: System software for persistent memory**. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 1–15, New York, NY, USA, April 2014. Association for Computing Machinery. 6, 14

[30] Utku Sirin, Pinar Tözün, Danica Porobic, Ahmad Yasin, and Anastasia Ailamaki. **Micro-architectural analysis of in-memory OLTP: Revisited**. *The VLDB Journal*, **30**(4):641–665, July 2021. 6

[31] Viktor Leis, Alfons Kemper, and Thomas Neumann. **Exploiting hardware transactional memory in main-memory databases**. In *2014 IEEE 30th International Conference on Data Engineering*, pages 580–591, March 2014. ISSN: 2375-026X. 6

[32] Hao Zhang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Meihui Zhang. **In-Memory Big Data Management and Processing: A Survey**. *IEEE Transactions on Knowledge and Data Engineering*, **27**(7):1920–1948, July 2015. Conference Name: IEEE Transactions on Knowledge and Data Engineering. 6

[33] **NVIDIA BlueField Data Processing Units (DPUs)**. 6

[34] Animesh Trivedi and Marco Spaziani Brunella. **CPU-free Computing: A Vision with a Blueprint**. In *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, HOTOS '23, pages 1–14, New York, NY, USA, June 2023. Association for Computing Machinery. 6

[35] Peter-Jan Gootzen, Jonas Pfefferle, Radu Stoica, and Animesh Trivedi. **DPFS: DPU-Powered File System Virtualization**. In *Proceedings of the 16th ACM International Conference on Systems and Storage*, SYSTOR '23, pages 1–7, New York, NY, USA, June 2023. Association for Computing Machinery. 6

[36] Naim A. Kheir. *Systems Modeling and Computer Simulation*. Marcel Dekker, Inc., New York, USA, 2nd edition. 7

[37] Yair Levy and Timothy J. Ellis. **A Systems Approach to Conduct an Effective Literature Review in Support of Information Systems Research**. *Informing Sci J*, **9**:181–212, 2006. 7

[38] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The AtLarge Vision on the Design of Distributed Systems and Ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776. IEEE, 2019. 7, 9

[39] Richard Hamming. *The Art of Doing Science and Engineering: Learning to Learn*. CRC Press, 1997. 7

[40] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. **A Design Science Research Methodology for Information Systems Research**. *Journal of Management Information Systems*, **24**(3):45–77, 2008. Publisher: Taylor & Francis. 7

[41] R Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons Inc., New York, USA, 1991. 7

[42] Gernot Heiser. **Systems Benchmarking Crimes**, 2019. 7

[43] John Ousterhout. **Always measure one level deeper**. *Communications of the ACM*, **61**(7):74–83, 2018. Publisher: ACM. 7

[44] Sonja Bezjak, April Clyburne-Sherin, Philipp Conzett, Pedro Fernandes, Edit Görögh, Kerstin Helbig, Bianca Kramer, Ignasi Labastida, Kyle Niemeyer, Fotis Psomopoulos, Tony Ross-Hellauer, René Schneider, Jon Tennant, Ellen Verbakel, Helene Brinken, and Lambert Heller. **Open Science Training Handbook**, 2018. 7

[45] Mark D. Wilkinson, Michel Dumontier, and Aalbersberg. **The FAIR Guiding Principles for scientific data management and stewardship**. *Nature Scientific Data*, **3**, 2016. 7

[46] Emery D. Berger, Stephen M. Blackburn, Matthias Hauswirth, and Michael W. Hicks. **A Checklist Manifesto for Empirical Evaluation: A Preemptive Strike Against a Replication Crisis in Computer Science**. 2019. 7

[47] Alexandru Uta, Alexandru Custura, Dmitry Duplyakin, Ivo Jimenez, Jan Rellermeyer, Carlos Maltzahn, Robert Ricci, and Alexandru Iosup. **Is Big Data Performance Reproducible in Modern Cloud Networks?** In *NSDI*, 2020. 7

[48] Carol Sliwa. **Adoption of Intel Optane persistent memory picks up in 2020**. *TechTarget Storage*, October 2020. 7, 54

[49] Linux Foundation. **ext4 Data Structures and Algorithms — The Linux Kernel documentation**. 9, 10, 51, 71

[50] Jakob Luttgau, Michael Kuhn, Kira Duwe, Yevhen Alforov, Eugen Betke, Julian Kunkel, and Thomas Ludwig. **Survey of Storage Systems for High-Performance Computing**. *Supercomputing Frontiers and Innovations: an International Journal*, **5**(1):31–58, 2018. 9

[51] Simon D. Smart, Tiago Quintino, and Baudouin Raoult. **A High-Performance Distributed Object-Store for Exascale Numerical Weather Prediction and Climate**. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '19, pages 1–11, New York, NY, USA, June 2019. Association for Computing Machinery. 9

[52] Michèle Weiland and Bernhard Homölle. **Usage Scenarios for Byte-Addressable Persistent Memory in High-Performance and Data Intensive**

Computing. *Journal of Computer Science and Technology*, **36**(1):110–122, January 2021. 9

[53] Yehonatan Fridman, Yaniv Snir, Matan Rusanovsky, Kfir Zvi, Harel Levin, Danny Hendler, Hagit Attiya, and Gal Oren. **Assessing the Use Cases of Persistent Memory in High-Performance Scientific Computing**. pages 11–20. IEEE Computer Society, November 2021. 9

[54] Linux Foundation. **Page Table Management**. 13

[55] Intel. **Intel® 64 and IA-32 Architectures Software Developer's Manual**, April 2022. 13, 14, 16, 22, 39, 52, 69

[56] IAIK. **Paging on Intel x86-64 – IAIK**. 13

[57] Ivy B. Peng, Maya B. Gokhale, and Eric W. Green. **System evaluation of the Intel optane byte-addressable NVM**. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, pages 304–315, New York, NY, USA, September 2019. Association for Computing Machinery. 14

[58] Chao Su and Qingkai Zeng. **Survey of CPU Cache-Based Side-Channel Attacks: Systematic Analysis, Security Models, and Countermeasures**. *Security and Communication Networks*, **2021**:e5559552, June 2021. Publisher: Hindawi. 14

[59] Saarland Informatics Campus. **Cache Latencies**. 14

[60] K. Bhandari, D.R. Chakrabarti, and H.-J Boehm. **Implications of CPU caching on byte-addressable non-volatile memory programming**. January 2012. 14, 15, 16

[61] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. **Loose-Ordering Consistency for persistent memory**. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 216–223, October 2014. ISSN: 1063-6404. 14

[62] Paul Caheny, Lluc Alvarez, Said Derradji, Mateo Valero, Miquel Moretó, and Marc Casas. **Reducing Cache Coherence Traffic with a NUMA-Aware Runtime Approach**. *IEEE Transactions on Parallel and Distributed Systems*, **29**(5):1174–1187, May 2018. Conference Name: IEEE Transactions on Parallel and Distributed Systems. 15, 62

[63] Paul McKenney. **Memory Ordering in Modern Microprocessors, Part I | Linux Journal**. *Linux Journal*, June 2005. 15

[64] Yang Yang, Qiang Cao, Jie Yao, Yuanyuan Dong, and Weikang Kong. **SPMFS: A Scalable Persistent Memory File System on Optane Persistent Memory**. In *50th International Conference on Parallel Processing*, ICPP 2021, pages 1–10, New York, NY, USA, October 2021. Association for Computing Machinery. 15

[65] **NVM Programming Model (Version 1.2)**, June 2017. 15

[66] Intel. **eADR: New Opportunities for Persistent Memory Applications**. 16

[67] Intel. **Persistent Memory Learn More Series Part 2 : Power-Fail Protected Domains**, January 2021. 16

[68] **Persistent Memory Extensions - x86 - WikiChip**, May 2021. 17

[69] Miao Cai. **FlatFS: Flatten Hierarchical File System Namespace on Non-volatile Memories**. *Usenix*, July 2022. 17

[70] Linux Foundation. **Direct Access for files**. 19, 20

[71] Vincent Weaver. **System-wide Performance Counter Measurements: Offcore, Uncore, and Northbridge Performance Events in Modern Processors**. Technical Report UMAINE-VMW-TR-UNCORE-OFFCORE-2015-10, University of Maine, July 2017. 22

[72] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. **Understanding the idiosyncrasies of real persistent memory**. *Proceedings of the VLDB Endowment*, **14**(4):626–639, February 2021. 24, 27, 55, 56, 62

[73] Lawrence Benson, Leon Papke, and Tilmann Rabl. **PerMA-bench: benchmarking persistent memory access**. *Proceedings of the VLDB Endowment*, **15**(11):2463–2476, September 2022. 24, 55, 60

[74] Intel. **Intel VTune Profiler**. 24

[75] The Linux Foundation. **perf**. 24, 68

[76] Jian Xu, Juno Kim, Amirsaman Memaripour, and Steven Swanson. **Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks**. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 427–439, New York, NY, USA, April 2019. Association for Computing Machinery. 27, 55

[77] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. **Performance characterization of a DRAM-NVM hybrid memory architecture for HPC applications using intel optane DC persistent memory modules**. In *Proceedings of the International Symposium on Memory Systems*, MEMSYS '19, pages 288–303, New York, NY, USA, September 2019. Association for Computing Machinery. 27, 55

[78] Richard A. Uhlig and Trevor N. Mudge. **Trace-driven memory simulation: a survey**. *ACM Computing Surveys*, **29**(2):128–170, June 1997. 29

[79] UEFI Forum, Inc. **Advanced Configuration and Power Interface (ACPI) Specification**, January 2021. 31

[80] pmem.io community. **Using QEMU Virtualization**. 31, 65

[81] Srikar Dronamraju. **Uprobe-tracer: Uprobe-based Event Tracing — The Linux Kernel documentation**. 36

[82] The Linux Kernel Contributors. **In-kernel memory-mapped I/O tracing**. 37, 47

[83] Youyou Lu, Jiwu Shu, Long Sun, and Onur Mutlu. **Improving the Performance and Endurance of Persistent Memory with Loose-Ordering Consistency**. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2018. arXiv:1705.03623 [cs]. 41

[84] Chris Lattner and Vikram Adve. **LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation**. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, page 75, USA, March 2004. IEEE Computer Society. 42

[85] Keith D. Cooper and Linda Torczon. **Chapter 3 - Parsers**. In Keith D. Cooper and Linda Torczon, editors, *Engineering a Compiler (Second Edition)*, pages 83–159. Morgan Kaufmann, Boston, January 2012. 42

[86] osdev. **x86-64 System Call Calling Conventions**. 42

[87] Apache. **Apache Parquet**, April 2023. 45

[88] Vogl, Sebastian and Eckert, Claudia. **Using Hardware Performance Events for Instruction-Level Monitoring on the x86 Architecture**. In *Proceedings of EuroSec'12, 5th European Workshop on System Security*. ACM Press, April 2012. 46

[89] Jo Van Bulck, Frank Piessens, and Raoul Strackx. **SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control**. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, SysTEX'17, pages 1–6, New York, NY, USA, October 2017. Association for Computing Machinery. 47

[90] Myoungjae Kim, Hyunmin Yoon, Minkwan Choi, Shakaiba Majeed, and Minsoo Ryu. **Multiprocessor MMIO Tracing via Memory Protection and a Shadow Page Table**. In *Proceedings of the International Conference on Foundations of Computer Science (FCS)*, page 16. The Steering Committee of The World Congress in Computer Science, Computer . . . , 2015. 48

[91] Axboe, Jens. **Flexible I/O Tester**, 2022. GNU GPL v2.0. 52

[92] Kitchenham, Barbara Ann and Charters, Stuart. **Guidelines for performing Systematic Literature Reviews in Software Engineering**. Technical Report EBSE-2007-01, Software Engineering Group Keele University, July 2007. 53

[93] Claes Wohlin. **Guidelines for snowballing in systematic literature studies and a replication in software engineering**. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, EASE '14, pages 1–10, New York, NY, USA, May 2014. Association for Computing Machinery. 53

[94] Kai Wu, Jie Ren, Ivy Peng, and Dong Li. **{ArchTM}: {Architecture-Aware}, High Performance Transaction for Persistent Memory**. pages 141–153, 2021. 55

[95] Guangyu Zhu, Jaehyun Han, Sangjin Lee, and Yongseok Son. **An Empirical Evaluation of NVM-Aware File Systems on Intel Optane DC Persistent Memory Modules**. *MDPI - Electronics*, **10**(16):1977, January 2021. Number: 16 Publisher: Multidisciplinary Digital Publishing Institute. 55

[96] Anuj Kalia, David Andersen, and Michael Kaminsky. **Challenges and solutions for fast remote persistent memory access**. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, pages 105–119, New York, NY, USA, October 2020. Association for Computing Machinery. 55, 63

[97] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. **{ODINFS}: Scaling {PM} Performance with Opportunistic Delegation**. pages 179–193, Carlsbad, CA, 2022. USENIX Association. 55

[98] Saugata Ghose, Tianshi Li, Nastaran Hajinazar, Damla Senol Cali, and Onur Mutlu. **Demystifying Complex Workload-DRAM Interactions: An Experimental Study**. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, **3**(3):60:1–60:50, December 2019. 57

[99] Soramichi Akiyama and Takahiro Hirofuchi. **Quantitative Evaluation of Intel PEBS Overhead for Online System-Noise Analysis**. In *Proceedings of the 7th International Workshop on Runtime and Operating Systems for Supercomputers ROSS 2017*, ROSS '17, pages 1–8, New York, NY, USA, June 2017. Association for Computing Machinery. 58

[100] Jifei Yi, Benchao Dong, Mingkai Dong, Ruizhe Tong, and Haibo Chen. **{MT^2}: Memory Bandwidth Regulation on Hybrid {NVM/DRAM} Platforms**. In *20th USENIX Conference on File and Storage Technologies (FAST 22)*, pages 199–216. USENIX Association, 2022. 58

[101] **An Introduction to the Intel QuickPath Interconnect**. Technical Report 320412-001US, Intel. 62

[102] **Intel® Intrinsics Guide**, May 2023. 67

[103] eBPF community. **eBPF**. 68

[104] Vincent Weaver. **Linux perf event Features and Overhead**. Technical Report, University of Maine, 2013. 68

[105] Tarasov Vasily, Erez Zadok, and Shepler Spencer. **Filebench: A flexible framework for file system benchmarking.**, 2016. 71

[106] Richard McDougall. **FileBench**, 2004. 71

[107] Gabriele Paoloni. **How to Benchmark Code Execution Times on Intel IA-32 and IA-64 Instruction Set Architectures**. Technical Report 324264-001, Intel, September 2010. 71

[108] **Intel® Memory Latency Checker v3.9a**, July 2021. 76

7

Artifacts

7.1 Artifact 1: Configuring VM environment *pmemtrace*

This artifact provides instructions for reproducing the QEMU VM environment, including the *pmemtrace* trace collection tool discussed in chapter 3. Preferably, one should have access to a machine with 64 GB of RAM or more. In this stage, having access to real Persistent Memory hardware is optional since PMEM is emulated by QEMU. Finally, the number of physical cores should be at least eight.

Most of the installation is automated by scripts, so the entire installation process should take approximately 30 minutes.

Artifact Check-list (Meta-information)

- Compilation: GCC, CMake 3.22.1, make
- Hardware: `x86_64` CPU with at least 8 cores, 64 GB of RAM (< 64 GB requires manual configuration)
- How much disk space required?: 30 GB
- How much time is needed to prepare workflow (approximately)?: 30 minutes, mostly depends on kernel build time
- Publicly available?: Yes: `https://github.com/stonet-research/PMicroProfile`
- Code licenses?: Multi-license, Linux kernel version `5-4.232` is GPL-2.0 licensed, SplitFS uses the BSD license. Our work is MIT licensed.

Description

How to access

All necessary code, experiments, and installation scripts are included in a Git repository. This repository can be obtained by running the following command in a shell:

```
$ git clone https://github.com/stonet-research/PMicroProfile
```

Hardware Dependencies

An Intel processor (Skylake architecture or newer) that supports the `clflush`, `clflushopt`, or `clwb` cache bypass instructions. Other processor vendors, such as AMD, that implement both the `x86_64` instruction set and the aforementioned instructions may function as well; however, this has not been verified.

Software Dependencies

A Linux-based operating system that supports KVM. Our installation scripts target Debian-based systems, for example, Ubuntu 22.04.* LTS.

Our Hardware/Software configuration

We use the following system configuration:

- CPU: Intel Core i9 13900K processor;
- RAM: 4× Corsair Vengeance 16 GB 5600 MHz DDR5 memory (64 GB in total);
- SSD: 1× Samsung 980 Pro 1 TB;
- OS: Ubuntu `22.04.2` LTS running the Linux `5.19.0-41-generic` kernel, QEMU version `6.2.0`.

Installation

The installation process consists of two steps: establishing a QEMU virtual machine instance with a custom kernel and afterward installing the *pmemtrace* tracing tool. Most of the steps involve running automated scripts that pre-configure all required dependencies and infrastructure in such a way that it requires minimal effort.

First, set up a new QEMU VM instance:

```
$ git clone https://github.com/stonet-research/PMicroProfile
$ cd PMicroProfile

# Run the automated install script.
$ ./setup-vm.sh
```

A successful installation should result in the following output:

```
✓ Installed package dependencies.
✓ Built kernel.
✓ Configured kernel.
✓ Created QEMU disk image.
✓ Setup Finished! Execute the "vm/run_kvm_iso.sh" command to launch
    Ubuntu Installer. Make sure that you select "QEMU HARDDISK" as
    install drive!
```

You may now proceed to the Ubuntu installation wizard by executing the `vm/run_kvm_iso.sh` script. After finishing the Ubuntu installation, perform the following steps:

1. Remove the installation medium by commenting/removing the line `-cdrom "ubuntu.iso"` in `vm/run_kvm_iso.sh`;
2. Open a terminal inside the VM and execute the following command: `$ cat /proc/mounts | grep " / "`;
3. Copy the mount path of the root file system, for example, `/dev/sda5`, and update the kernel `root` boot parameter in the `vm/run_kvm.sh` file accordingly;
4. *Optional*: Modify the number of cores, the amount of RAM, and the size of the emulated PMEM device by modifying the `vm/run_kvm.sh` file accordingly;
5. Now, boot the VM using the custom kernel by executing the script `vm/run_kvm.sh`;
6. Verify that you are running the custom kernel by running `$ uname -r` within the VM. This command should print `5.4.232`.

Within the VM, spawn a terminal and, again, clone the git repository. Subsequently, run the *pmemtrace* installation script:

```
$ git clone https://github.com/stonet-research/PMicroProfile
$ cd PMicroProfile

# Run the automated pmemtrace install script.
$ ./install-vm.sh
```

The `pmemtrace` executable will be placed inside the `/usr/local/bin/` folder so that it is contained in the user `$PATH`. Verify this by running `sudo pmemtrace --help`.

Experiment Workflow

Mounting PMEM file system. To use a Persistent Memory device, it must first be mounted inside the user space. There are two options available: `devdax` and `fsdax`. The former enables applications to access storage without the involvement of a file system. The latter attaches the device to a file system. To mount the device as `devdax`, one can execute the following command with the repository working directory: `./mount-dexdax.sh`. Alternatively, the `./mount-ext4-dax.sh` script can be used to mount a `ext4` file system with the PMEM device as the backend. Artifact 2 provides a discussion on how to mount other file systems.

Using *pmemtrace*. To enable access tracing when executing a CLI command, use the following syntax: `sudo pmemtrace [OPTIONS] experiment_name [COMMAND]`. When executing the command, *pmemtrace* configures the in-kernel infrastructure to log low-level PMEM accesses and then executes the provided command. As described in Figure 3.5, a sampling-based data collection strategy to decrease tracing overhead. Related command-line arguments are: `--sample-rate` (between 0 and 240 hertz) and `--duty-cycle` ($[0.0, 1.0]$). To capture all events, one can disable sampling by adding the `--disable-sampling` toggle. Example usage:

```
# Mount PMEM device as fsdax
$ cd thesis-research
$ sudo ./mount-ext4.dax

# Example of tracing all accesses random file write (i.e. sampling
    disabled)
# Optionally, one may enable experimental multi-core capturing
    support by setting the --enable-multicore flag.
$ sudo pmemtrace randwrite-all-exp sudo bash -c "head -c 16M
    </dev/urandom >/mnt/pmem_emul/rand_file.txt" --disable-sampling
```

```
# Example of tracing all accesses random file write with sampling
    (60 hertz, 80% duty cycle):
$ sudo pmemtrace randwrite-sampling-exp sudo bash -c "head -c 16M
    </dev/urandom >/mnt/pmem_emul/rand_file.txt" --sample-rate 60
    --duty-cycle 0.8
```

The compressed trace file is saved as `[experiment_name].parquet` in the current working directory. This trace file is now ready to be replayed in *pmemanalyze*. A readable log can also be found in the `/tmp` directory.

Evaluation and Expected Results

Provided in Artifact 2 (section 7.2).

7.2 Artifact 2: Reproducible Experiments

This artifact provides all the information to reproduce the experiment plots and tables throughout this thesis. Together, the entire process should take around two hour.

Artifact Check-list (Meta-information)

- Compilation: GCC, CMake 3.22.1, make
- Data set: randomly generated data by urandom and other existing file system benchmarks, e.g., FIO.
- Run-time environment: real machine (no VM), running Linux 5.* kernel.
- Hardware: Intel Xeon Silver 4215 Server CPU, with 72 GiB of RAM, 4× Intel Optane DCPMM (NMA1XBD512GQS, 512 GiB);
- Metrics: runtime, throughput, latency, Intel PMC and PEBS events.
- Output: mostly plots, some raw csv files;
- Experiments: all experiments are contained in the `experiments/` folder within the Git repository. Each experiment is numbered (e.g., 01) and is contained in separate folders. Each experiment contains a `REPRODUCE.md` file with instructions to reproduce the results.
- How much disk space required?: 10 GiB
- How much time is needed to prepare workflow (approximately)?: 1 hour
- How much time is needed to complete experiments (approximately)?: $1 - 2$ hours

- Publicly available?: Yes: `https://github.com/stonet-research/PMicroProfile`
- Code licenses?: Multi-license, Linux kernel version `5-4.232` is GPL-2.0 licensed, SplitFS uses the BSD license. Our work is MIT licensed.

Description

How to access

All necessary code, experiments, and installation scripts are included in a Git repository. This repository can be obtained by running the following command in a shell:

```
$ git clone https://github.com/stonet-research/PMicroProfile
```

Hardware Dependencies

The system must contain one or more Intel Optane DCPMM DIMM modules. Furthermore, an Intel *Cascadelake-Server* based CPU is mandatory. A virtual machine is not supported since it does not implement (representative) Intel PMC and PEBS performance counters required for performance evaluation.

Software Dependencies

A system running a `5.*` Linux kernel. We used an Ubuntu 20.04.6 LTS installation, running kernel version `5.7.1`.

Installation

The installation process involves setting up the *pmemanalyze* tool. To install this tool, run the following bash commands:

```
$ git clone https://github.com/stonet-research/PMicroProfile
$ cd PMicroProfile

# Run the automated pmemanalyze install script.
$ ./install-real-machine.sh

# Change directory to experiments folder.
cd experiments
```

Experiment Workflow

All experiments are contained in the `experiments` folder. Each experiment includes a `REPRODUCE.md` file with instructions for reproducing the results of that specific experiment.

The following overview can be used to find the respective experiment folder of a figure/table contained in this thesis.

Figure	Folder
Table 3.2	`01-runtime-rand-pmemtrace`
Table 3.3	`01-runtime-rand-pmemtrace`
Figure 3.9	`03-pmemtrace-overhead`
Figure 3.8	`03-pmemtrace-overhead`
Figure 4.1	`02-fs-overhead`
Figure 5.2	`05-ext4-splitfs-varmail`
Figure 5.3	`05-ext4-splitfs-varmail`
Figure 5.4	`05-ext4-splitfs-varmail`
Figure 5.5	`05-ext4-splitfs-varmail`

Evaluation and Expected Results

Can be found inside the `REPRODUCE.md` file within the `experiments` folder.

8

Appendix

Appendix A: Capturing PMEM driver events using eBPF and BCC

```
from bcc import BPF
from bcc.utils import printb

b = BPF(text="""
#include <uapi/linux/ptrace.h>
#include <linux/blk-mq.h>
#include <uapi/linux/virtio_pmem.h>
#include <linux/libnvdimm.h>

void trace_write(struct pt_regs *ctx, void *pmem_addr) {
  bpf_trace_printk("W %p \\n", pmem_addr);
}

void trace_read(struct pt_regs *ctx, void *pmem_addr) {
  bpf_trace_printk("R %p \\n", pmem_addr);
}
""")

if BPF.get_kprobe_functions(b'write_pmem'):
  b.attach_kprobe(event="write_pmem", fn_name="trace_write")
  print("Found write_pmem function!")

if BPF.get_kprobe_functions(b'read_pmem'):
  b.attach_kprobe(event="read_pmem", fn_name="trace_read")
  print("Found read_pmem function!")

```

```
while 1:
  try:
    (task, pid, cpu, flags, ts, msg) = b.trace_fields()
    (op, pmem_addr) = msg.split()

    if op == b'W':
      print("WRITE {}".format(hex(pmem_addr)))
    elif op == b'R':
      print(f"READ {pmem_addr}")
    else:
      print(f"UNKNOWN OP {op}")
  except KeyboardInterrupt:
    exit()
```

Appendix B: QEMU device setup methodology 1

```
qemu-system-x86_64\
  -cpu host \
  -enable-kvm \
  -smp cores=4 \
  -drive file=ubuntu.img.qcow2,format=qcow2 \
  -append "root=/dev/sda5 earlyprintk=serial net.ifnames=0
     nokaslr" \
  -kernel $1/arch/x86/boot/bzImage \
  -machine pc,nvdimm=on \
  -m 4G,slots=2,maxmem=32G \
  -object memory-backend-file,id=mem1,mem-path=./nvdimm0,
            share=on,pmem=on,size=28G,align=2M \
  -device nvdimm,memdev=mem1,id=nv1,label-size=256K \
  -net user,host=10.0.2.10,hostfwd=tcp:127.0.0.1:2222-:22 \
  -net nic,model=e1000 \
        -vga virtio \
  -pidfile vm.pid
```

Listing 8.1: QEMU device Setup

Appendix C: List of methods PMC library

These methods are declared in the `pmemanalyze/include/pmc.hpp` file and defined in the `pmemanalyze/src/pmc.cpp`. Class methods defined as `protected` and `private` have

been omitted.

1. **Initialize Library**: Find all the available integrated memory controllers by listing all the file entries in `/sys/bus/event_source/devices`. The ID of each iMC is stored in a `std::array`.

   Definition: `bool init()`

2. **Create iMC probe**: Initialize an iMC probe using the event ID listed in Intel's Cascade Lake Event Overview: `https://perfmon-events.intel.com/cascadelake_server.html`.

   Definition: `bool add_imc_probe(const unsigned int event_id, const bool is_single)`

3. **Create offcore probe**: Initialize an iMC probe using the event ID listed in Intel's Cascade Lake Event Overview: `https://perfmon-events.intel.com/cascadelake_server.html`. As offcore events can be attributed to a single CPU, the calling process's Process Identifier (PID) should be provided. Additionally, the `msr` variable must be set accordingly.

   Definition: `bool add_offcore_probe(const unsigned int event_id, const int pid, const unsigned long msr)`

4. **Enable all iMC probes**: Enables all iMC probes, used to implement sampling.

   Definition: `void enable_imc_probes()const`

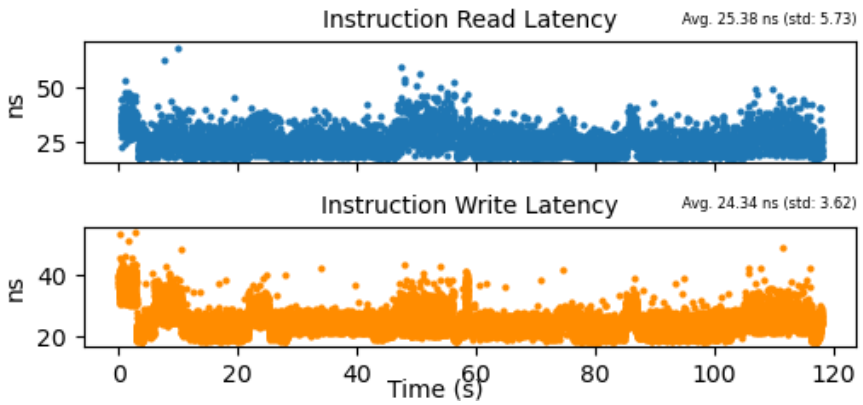
5. **Disable all iMC probes**: Disables all iMC probes, used to implement sampling.

   Definition: `void disable_imc_probes()const`

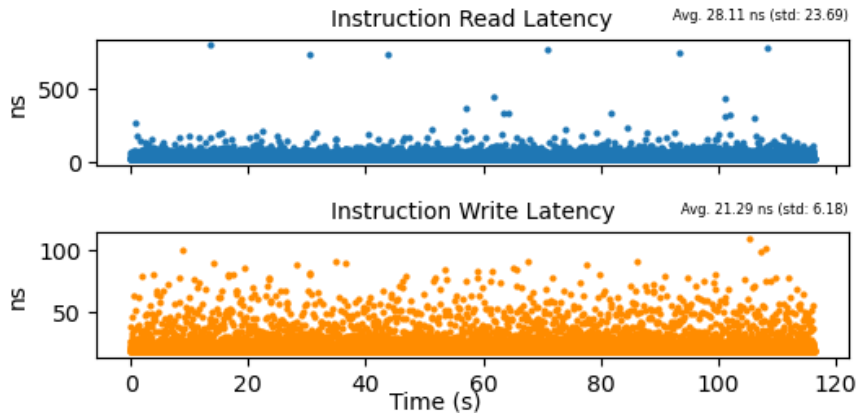
6. **Reset iMC counters**: Reset iMC counter values, used to implement sampling.

   Definition: `void reset_imc_probes()const`

Appendix D: Long-Duration Latencies *varmail* Workload



(a)



(b)

Figure 8.1: Ext4-DAX (a) and SplitFS (b) long-duration latencies