Karlsruher Institut für Technologie

Efficient and Correct Persistent Memory File Systems

Zur Erlangung des akademischen Grades eines

Doktors der Ingenieurwissenschaften

von der KIT-Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Lukas Werling

Tag der mündlichen Prüfung: 12.12.2025

Erster Referent:	Prof. Dr. Frank Bellosa
Zweiter Referent:	Prof. Dr. Christian Dietrich

In memory of my grandma

Emma Westermann

1929 – 2026

Abstract

Persistent memory (PM) is a novel storage technology that enables byte-granular direct access from the CPU with low latency. Compared to traditional asynchronous block storage, this access paradigm allows file systems to offer stronger persistence guarantees at lower latency. However, it also introduces new challenges for performance, efficiency, and correctness. PM's small atomic write size requires careful use of PM primitives to prevent data loss in the event of a crash. Overloading PM with parallel accesses results in expensive CPU stalls.

This thesis investigates the efficiency and correctness of PM file systems. First, we introduce efficiency metrics that quantify CPU time and energy cost per unit of storage access. We show that many existing PM file systems perform poorly under parallel load. To address PM overload, we design mitigation mechanisms that integrate with existing file systems and a monitoring technique to attribute direct-access PM traffic to processes.

Second, to improve correctness, we present **Suvi**, an approach to black-box crash consistency testing for PM file systems. **Suvi** traces a file system's PM and NVMe SSD accesses in a virtual machine and replays the trace with an accurate simulation of x86 store-order semantics. **Suvi** generates crash images using two heuristics to avoid combinatorial explosion, and then automatically analyzes the crash images to detect atomicity bugs.

Together, these contributions provide measurement tools, mitigation strategies, and testing infrastructure to make PM file systems more efficient and more reliable.

Acknowledgements

I would like to express my gratitude to Prof. Dr. Frank Bellosa for his continuous support and helpful advice. He granted me the freedom to pursue my ideas and provided valuable networking opportunities at conferences.

I am grateful to Prof. Dr. Christian Dietrich for kindly agreeing to act as the secondary reviewer and for his constructive comments whenever I presented my work at conferences and workshops.

Many thanks to my colleagues Daniel Habicht, Peter Maucher, and Thorsten Gröninger for their thoughtful feedback on both the thesis and its presentation. Their suggestions helped me improve the clarity and fix errors in technical descriptions. I also thank Johannes Bechberger for his careful proofreading.

Finally, I owe special thanks to Christian Schwarz. His master's thesis provided the initial motivation to pursue PM file system efficiency and correctness. Christian also provided regular insight into storage challenges from an industry perspective.

Contents

Abstract **5**

Acknowledgements **7**

1 Introduction **13**

1.1 Motivation for Persistent Memory 13
1.2 Challenges for PM File Systems 14
1.3 Measuring and Improving PM Efficiency 15
1.4 PM File System Crash Consistency 16
1.5 Contributions 16
1.6 Student Theses and Publications 17
1.7 Structure 19

2 Background – Persistent Memory and File Systems **21**

2.1 Optane Persistent Memory 21
2.1.1 Regions 22
2.1.2 Namespaces 23
2.1.3 Performance Counters 23
2.2 Linux File System Support for PM 25
2.2.1 Accessing PM from the Kernel 25
2.2.2 Block Translation Table (BTT) 25
2.2.3 Ext4 and XFS without BTT 26
2.2.4 DAX Support 27
2.3 PM File Systems 27
2.3.1 PMFS, WineFS, and OdinFS 28
2.3.2 Aerie 29
2.3.3 NOVA, NOVA-Fortis, and Ziggurat 30
2.3.4 Strata and Assise 31
2.3.5 SPMFS 31
2.3.6 Trio 32
2.3.7 P2CACHE 33
2.3.8 SlotFS 33
2.4 Crash Consistency 34
2.4.1 Crash Consistency for Persistent Memory 35
2.4.2 Crash Consistency for NVMe 37

3 Motivation – Designing Data Structures for PM **41**

3.1 PM Ring Buffer Overview 41

3.1.1 PM Organization and Runtime Data ... 42
3.1.2 Recovery ... 43
3.1.3 Write Process ... 43
3.2 Implementation and Evaluation Setup ... 46
3.3 Memory Access Instructions ... 47
3.4 Parallel Accesses ... 49
3.4.1 Non-Interleaved Optane PM ... 49
3.5 Alignment and Access Size ... 50
3.6 NUMA ... 51
3.7 Discussion ... 52
3.8 PM Ring Buffer for File Systems ... 53
3.8.1 ZIL-PMEM: PM Write Cache for ZFS ... 53
3.8.2 DPWC: Write Cache for Block Devices ... 53

4 PM File System Efficiency ... 55
4.1 Metrics for File System Efficiency ... 56
4.1.1 Energy Efficiency ... 57
4.1.2 CPU Efficiency ... 58
4.2 Analyzing File System Efficiency ... 59
4.2.1 Measurement Setup ... 59
4.2.2 File System Selection ... 61
4.2.3 FIO Benchmark Setup ... 62
4.2.4 Power Measurements ... 63
4.3 Evaluation ... 66
4.3.1 ext4 and NOVA ... 66
4.3.2 OdinFS, ZIL-PMEM, and DPWC ... 69
4.4 Discussion ... 72

5 PM File System Overload Mitigation ... 73
5.1 Design and Implementation ... 73
5.1.1 Semaphore ... 74
5.1.2 Workqueue ... 75
5.1.3 DMA ... 75
5.2 File System Integration ... 76
5.3 Evaluation ... 76
5.3.1 Semaphore ... 76
5.3.2 Workqueue ... 77
5.3.3 DMA ... 80
5.4 Discussion ... 80

6 Userspace PM Access Accounting ... 81
6.1 Requirements ... 82
6.2 Accounting with Performance Counters ... 82
6.2.1 Read Accesses ... 82
6.2.2 Write Accesses ... 83
6.3 Approach: Sampling Memory Instructions ... 83
6.3.1 Processor Event Based Sampling (PEBS) ... 84
6.3.2 Bandwidth Estimation ... 84

6.3.3 Limitations ... 85
6.4 Implementation ... 86
6.5 Scheduling ... 87
6.6 Evaluation ... 88
6.6.1 Accuracy ... 88
6.6.2 Overhead and Latency ... 89
6.6.3 Scheduling ... 91
6.7 Related Work ... 92
6.8 Discussion ... 92

7 Crash Consistency Testing ... 95
7.1 Failure Points and Crash Images ... 95
7.2 Types of Crash Consistency Bugs ... 96
7.2.1 Logic Bugs ... 96
7.2.2 Missing Flush ... 97
7.2.3 Ordering Bug ... 97
7.2.4 Performance Bugs ... 98
7.3 Crash Consistency Testing Pipeline ... 98
7.4 Tracing Approaches ... 98
7.4.1 Binary Translation ... 99
7.4.2 Compiler Instrumentation ... 99
7.4.3 Manual Annotation ... 100
7.4.4 Function Tracing ... 100
7.5 Crash Consistency Testing Tools ... 100
7.5.1 Yat ... 101
7.5.2 PMTest ... 102
7.5.3 XFDetector ... 102
7.5.4 Witcher ... 103
7.5.5 Vinter ... 104
7.5.6 Chipmunk ... 104
7.5.7 Mumak ... 105

8 Suvi: Crash Consistency Testing for PM File Systems ... 107
8.1 Tracer ... 108
8.1.1 PM Tracing ... 109
8.1.2 NVMe Tracing ... 110
8.1.3 Hypercalls ... 112
8.1.4 Discussion ... 112
8.2 Crash Image Generator ... 113
8.2.1 Model Goals ... 113
8.2.2 Crash Image Metadata ... 114
8.3 PM Crash Image Model ... 115
8.3.1 Trace Replay ... 115
8.3.2 Failure Points ... 115
8.3.3 Global Store Ordering ... 116
8.3.4 Mixed Non-Temporal and Cached Stores ... 116
8.3.5 Crash Images with Volatile Caches ... 117
8.3.6 Crash Images with Persistent Caches (eADR) ... 119

8.3.7 Partial Application of Non-Temporal Stores ... 121

8.4 PM Crash Image Heuristics ... 121

8.4.1 **Suvi-Reads**: Efficient Exploration of Crash States ... 121

8.4.2 **Suvi-NT**: Detecting NT-Dependent Semantic States ... 125

8.4.3 **Suvi-Fast**: Fast Crash Image Generation for Logic Bugs ... 127

8.5 Cross-Media Crash Images ... 128

8.5.1 NVMe Crash Images ... 128

8.5.2 Combined Crash Images ... 130

8.5.3 Discussion ... 131

8.6 Tester ... 131

8.6.1 State Extraction ... 131

8.6.2 State Analysis ... 135

8.7 Trace Analysis ... 137

8.7.1 Trace Debugging ... 137

8.7.2 Trace Heuristics ... 139

9 Suvi: Implementation ... 141

9.1 Tracer ... 141

9.1.1 PANDA ... 141

9.1.2 QEMU ... 142

9.2 Memory Images ... 142

9.2.1 File System Copy-on-Write ... 143

9.2.2 Hash Memoization ... 143

9.2.3 Evaluation ... 144

9.3 Parallelization ... 145

9.3.1 Discussion ... 146

10 File System Testing with Suvi ... 149

10.1 Virtual Machine Setup ... 149

10.2 Test Cases ... 150

10.2.1 Vinter Test Cases ... 150

10.2.2 Automatic Test Case Generation ... 151

10.3 Performance ... 151

10.3.1 Heuristics ... 151

10.3.2 Tracer Implementations ... 152

10.3.3 Parallelization ... 153

10.4 Results ... 153

10.4.1 NOVA and NOVA-Fortis ... 153

10.4.2 PMFS and WineFS ... 155

10.4.3 ZIL-PMEM ... 155

10.5 Persistent Caches ... 155

10.6 Discussion ... 156

11 Conclusion ... 159

11.1 Outlook ... 160

References ... 163

Chapter 1

Introduction

1.1 Motivation for Persistent Memory

Modern systems have an increasing need for high-performance storage. Applications such as key-value stores [101], graph processing [64], and search indices [49, 51] require storage systems that can retrieve and modify data with minimal latency.

Traditional solid-state drives (SSDs) based on NVMe cannot properly fulfill this role. As both the SSDs and interconnects such as PCIe are getting faster, the relative overhead from storage accesses with NVMe increases. Approaches for kernel bypass can decrease this overhead [62, 83], but require exclusive device access for a single application.

Persistent memory (PM) offers a better solution. PM is byte-addressable and accessed directly from the CPU like main memory, but retains its contents while the system is powered off. PM therefore supports access with low latency, even for small access sizes. With regular paging, the operating system can offer direct PM access to userspace applications. In contrast to kernel bypass with NVMe, the operating system retains control over access permissions, allowing sharing of PM between multiple applications.

Intel Optane PM was the first widely-available commercial implementation of PM. Like DRAM, Optane modules come in the DIMM form factor and attach directly to the CPU's memory controller. We base most of the analysis in this thesis on Intel Optane PM.

More recently, support for Compute Express Link (CXL) appeared in CPUs from multiple vendors, including Intel and AMD. With CXL, devices attached via a PCIe link can offer byte-addressable PM. We expect that most of the techniques we introduce for Optane PM in this thesis also apply to CXL-attached PM.

File systems provide structured and controlled access to storage devices. By targeting a common file system API, applications can store data without knowledge of the underlying storage technology. The file system controls access permissions and allows safe shared access to different applications. Although PM can be used without a file system, file systems remain valuable for these reasons.

Most PM file systems offer *direct access* (DAX) to applications, a feature not found in regular file systems. With DAX, the file system provides memory mappings to a file's PM pages. After establishing such a mapping, an application can read and write to PM without further involvement of the operating system, bypassing the kernel.

1.2 Challenges for PM File Systems

PM can offer its low latency thanks to its integration into the CPU's memory hierarchy, allowing synchronous and direct access at byte granularity. In comparison, traditional storage devices are accessed over an asynchronous protocol. The operating system submits requests for data blocks, which the storage device fulfills by copying the data to system memory and then signaling request completion. From blocks to bytes is a change of paradigm that introduces novel challenges in the design and implementation of file systems for performance, efficiency, and correctness.

A correct file system implementation should be *crash-consistent*. In the event of a crash, the file system's data structures should remain consistent to avoid corrupted or lost files. Traditional storage devices support crash-consistent file system design by offering atomic block updates of typically 512 or 4096 bytes. In the event of a crash during a write operation, such a block is either written completely or not at all. With PM, the CPU's memory write path offers a much smaller atomic write size of only 8 bytes. Additionally, PM file systems and applications need to manage volatile state in the write path (e.g., caches and write buffers) by introducing special instructions called *PM primitives*. Correct use of these PM primitives is challenging since they do not have a visible effect on the application data during runtime but are critical for consistency after a crash.

Synchronous access to PM from the CPU is essential for low latency but becomes expensive once the PM is under load and cannot answer requests immediately. On traditional storage devices with asynchronous access, the operating system can schedule other processes during the wait time or put the CPU in a low-power sleep state. This is not possible with PM, as individual PM accesses are not visible to the operating system. Instead, the CPU pipeline stalls during the wait time, wasting CPU time and energy.

This problem is amplified with Optane PM due to its sensitivity to parallel accesses. Because of internal caching structures, its total throughput declines under parallel load, as shown in Figure 1.1. Individual threads then experience a significant

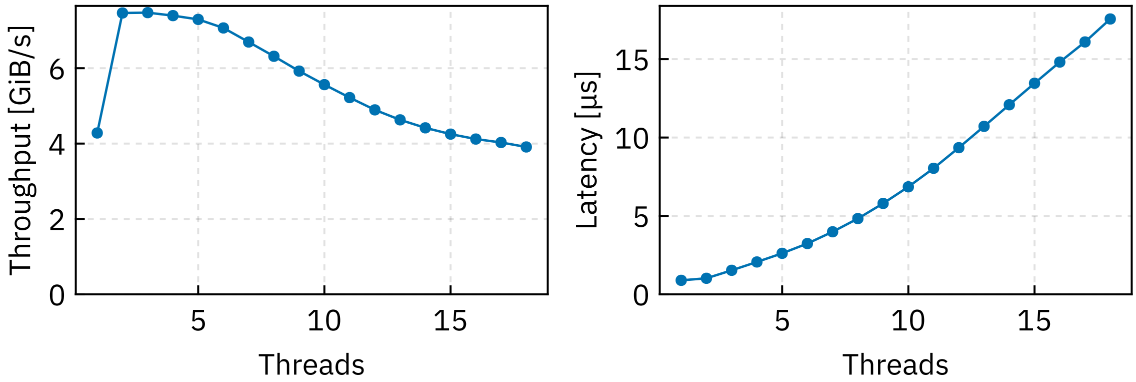


Figure 1.1: Throughput and latency of 4 KiB writes to Optane PM.

increase in access latency and, therefore, CPU stall cycles. To mitigate this problem, PM file systems and applications must limit parallel accesses to PM.

A secondary challenge follows from this requirement. After obtaining a direct PM mapping, applications can access PM without further involvement of the file system. This results in a situation in which neither the file system nor the application can manage parallel PM accesses. The file system does not have knowledge of application activity, and an application cannot know about the activity of other applications.

1.3 Measuring and Improving PM Efficiency

Since file systems act as an abstraction layer between applications and storage, they are responsible for ensuring that the storage is accessed as efficiently as possible for any given workload. For example, file systems for hard disks commonly reduce seeks by arranging related data blocks sequentially. Additionally, operating systems include I/O schedulers that can rearrange asynchronous requests to traditional storage devices. Similar I/O scheduling is not feasible for synchronously accessed PM.

In this thesis, we introduce approaches for measuring and improving the efficiency of PM file systems. Our efficiency metrics assign a specific cost (CPU time or energy) to accessing a certain amount of storage. We design the metrics so that they are independent of storage device throughput and CPU clock speed. We evaluate multiple file systems and find that most PM file systems do not access PM efficiently under a parallel write load.

We propose three mechanisms for improving the efficiency of PM file systems under parallel load. We design these mechanisms to be easily integrable into existing PM file systems. They include two software-based approaches for limiting parallel accesses and another approach based on hardware offloading. We show that these mechanisms are effective for improving PM file system efficiency. In combination with our efficiency metric, we expect future file system designs to incorporate efficient PM access into their designs.

Our efficiency metric and the mechanisms can only cover PM accesses over the file system API. Applications that access PM over DAX mappings cannot be covered. As discussed above, such applications should still avoid overloading PM with excessive parallel accesses. Effective limits require a global view of the volume of PM accesses from all concurrently running applications. Existing hardware and operating system mechanisms cannot provide this information with association to both applications and PM devices. We introduce a monitoring approach based on memory access instruction sampling that provides the required association. We show that our monitoring estimates PM write bandwidth accurately and with low latency.

We make the monitoring data available to applications via shared memory, allowing them to react immediately to overload situations. Using this monitoring data, we implement a policy based on *core specialization* that can limit the number of CPU cores stalling on PM accesses.

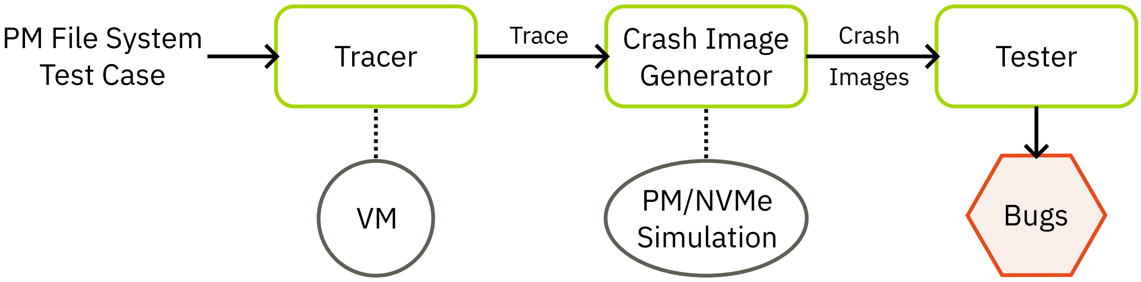


Figure 1.2: **Suvi**'s crash consistency testing pipeline.

1.4 PM File System Crash Consistency

The correct use of PM primitives is critical for PM file systems to ensure data consistency after a crash. In this thesis, we introduce **Suvi**, an approach for black-box crash consistency testing of PM file systems. As a black-box approach, **Suvi** places minimal requirements on the tested software and can analyze any PM file system running in a virtual machine. Rather than employing heuristics for bug detection, **Suvi** uses a record-and-replay approach that finds concrete witnesses for crash consistency bugs.

Suvi implements a crash consistency testing pipeline, shown in Figure 1.2, that traces PM accesses of a test case in a virtual machine. From the trace, it generates *crash images* that represent possible PM contents in the event of a crash. Finally, **Suvi** can automatically determine the crash atomicity of file system operations by analyzing the semantic state contained in the crash images.

Suvi innovates on previous approaches to crash consistency testing in multiple ways. It offers full-system tracing of PM and NVMe accesses using virtual machines with binary translation, allowing analysis of cross-media file systems that use these storage technologies. It includes an advanced PM simulation that models the ordering of x86 store instructions more precisely than other crash consistency testing approaches and supports both volatile and persistent caches. Two heuristics ensure efficient generation of crash images by avoiding a combinatorial explosion when there is a large number of PM stores. By using file system copy-on-write and a memoized hashing scheme, **Suvi** makes the analysis of large PM images feasible. Finally, **Suvi**'s analysis tools allow the automatic detection of crash consistency bugs and help developers identify the causes of such bugs.

1.5 Contributions

Our work presented in this thesis makes the following contributions:

- We introduce novel metrics for file system efficiency. The metrics measure the efficient use of CPU time and energy. We design them to be independent of both storage device throughput and CPU clock speed.
- We evaluate multiple file systems (PM and NVMe) with our metrics. Our results show that most PM file systems were not designed with efficiency in mind.
- We propose and evaluate measures improve the efficiency of PM file systems by mitigating PM overload.

- We propose an approach for accounting userspace PM accesses that can associate throughput with both processes and PM devices. We evaluate the accuracy and overhead of our approach. Using the accounting data, we propose PM overload mitigation based on *core specialization*.
- We introduce **Suvi**, a comprehensive approach for black-box crash consistency testing of file systems based on virtual machines. In particular, **Suvi** encompasses:
    - Support for analyzing cross-media file systems using PM and NVMe.
    - An improved simulation of x86 PM crash consistency semantics. Compared to previous work, it models the interaction of x86 global store order with weakly ordered non-temporal stores more accurately.
    - Simulation of both volatile and persistent caches.
    - Two primary strategies for efficiently generating crash images: an improved heuristic based on post-failure read accesses and a strategy for fast analysis of logic bugs.
    - A secondary heuristic for exploring crash states with weakly ordered non-temporal stores.
    - Novel handling of PM images through file system copy-on-write and memoized hashing, optimized for small modifications to large images.
    - Automated analysis of test results for atomicity, including tools for pinpointing the root cause of bugs.

1.6 Student Theses and Publications

Parts of this thesis are based on previously published information. We supervised a number of student theses that contributed to this thesis:

- In his master's thesis *Low-Latency Synchronous IO For OpenZFS Using Persistent Memory* [111], Christian Schwarz introduced ZIL-PMEM, a component for ZFS that logs synchronous writes on PM. The design and implementation of ZIL-PMEM motivate large parts of this thesis. We discuss ZIL-PMEM in Chapter 3, where we use a reimplementation of its low-level PM data structure to analyze Optane PM performance. Research in crash consistency testing that resulted in Vinter and **Suvi** was motivated by a desire to verify correctness of the implementation of ZIL-PMEM.
- With his master's thesis *Automatic Non-Volatile Memory Crash Consistency Testing for Full Systems* [69], Samuel Kalbfleisch provided the basis for Vinter [68]. We build **Suvi**, as presented in this thesis, on a reimplementation of his Python prototype in the Rust programming language.
- In his bachelor's thesis *Reducing Synchronous Write Latency With a PMEM Write Cache in the Device Mapper Layer* [32], Ilia Bozhinov adapts the PM data structure from ZIL-PMEM for use as a generic block device write cache. We analyze his implementation in Chapter 4.
- Daniel Ritz designed and implemented a crash consistency tester for NVMe in his bachelor's thesis *Crash Consistency Testing for Block Based File Systems on NVMe*

Drives [105]. Parts of his approach, especially for tracing NVMe commands, are part of **Suvi**'s support for NVMe.

- In his master's thesis *GPU4FS: A Graphics Processor-Accelerated File System* [91], Peter Maucher explored implementing a complete file system on a GPU. His analysis of PM accesses from a GPU informed our discussion of memcpy offloading in Chapter 5.
- Thomas Schmidt explored performance counters for accounting PM usage in his master's thesis *Achieving Optimal Throughput for Persistent Memory with Per-Process Accounting* [110]. His thesis included a userspace implementation of our idea for PEBS-based sampling, which failed to show useful results. For this reason, we created an independent kernelspace implementation for Chapter 6.
- In his bachelor's thesis *Crash Consistency Testing for Cross-Media File Systems using Persistent Memory and NVMe* [123], Lucas Wäldele united Vinter and Daniel Ritz's NVMe crash consistency tester. The major effort of this thesis was the implementation of PM tracing on top of modern QEMU since Vinter's PANDA-based tracer does not support NVMe. **Suvi**'s cross-media tracer evolved from his work.
- For his bachelor's thesis *Analyzing Persistent Memory Crash Consistency of WineFS with Vinter* [119], Paul Wedeck designed and implemented multiple small improvements to Vinter that we integrated into **Suvi**. Most prominently, these include support for parallel analysis of multiple tests and parallel state extraction.
- Thomas-Christian Oder integrated Mumak's [46] strategy for crash image generation into Vinter in his bachelor's thesis *Fast Persistent Memory Crash Consistency Analysis based on Virtual Machines* [99]. We adopt this strategy in **Suvi** as described in Chapter 8.

We previously presented parts of this thesis in the publications listed below. At USENIX ATC'22 [68] we introduced Vinter, an approach to PM crash consistency testing that provides the basis for **Suvi**. We later presented extensions to Vinter at FGBS'24 [121] that became part of **Suvi**, including cross-device analysis and faster crash image generation. At DIMES'23 [120], we presented our file system efficiency metric and mechanisms for improving the efficiency of PM file systems.

The remaining two publications are partially informed by the contributions of this thesis. With GPU4FS (FGBS'24 [90]), we take the idea of hardware offloading of PM accesses for efficiency further by moving the entire file system to the GPU. In our work on operating system support for CXL-based hybrid SSDs (DIMES'24 [50]), we evaluate efficiency using metrics from this thesis.

- Samuel Kalbfleisch, **Lukas Werling**, and Frank Bellosa. 2022. Vinter: Automatic Non-Volatile Memory Crash Consistency Testing for Full Systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022. 933–950. Retrieved from https://www.usenix.org/conference/atc22/presentation/werling
- **Lukas Werling**, Yussuf Khalil, Peter Maucher, Thorsten Gröninger, and Frank Bellosa. 2023. Analyzing and Improving CPU and Energy Efficiency of PM File Systems. In *Proceedings of the 1st Workshop on Disruptive Memory Systems*, October 2023. ACM, Koblenz Germany, 31–37. https://doi.org/10.1145/3609308.3625265

- **Lukas Werling**, Thomas-Christian Oder, Lucas Wäldele, Daniel Ritz, and Frank Bellosa. 2024. Improvements in Crash Consistency Testing for Persistent Memory File Systems. In *Tagungsband des FG-BS Frühjahrstreffens 2024*, 2024. Gesellschaft für Informatik e.V., Bochum, Germany. https://doi.org/10.18420/FGBS2024F-01
- Peter Maucher, Lennard Kittner, Nico Rath, Gregor Lucka, **Lukas Werling**, Yussuf Khalil, Thorsten Gröninger, and Frank Bellosa. 2024. Full-Scale File System Acceleration on GPU. In *Tagungsband des FG-BS Frühjahrstreffens 2024*, 2024. https://doi.org/10.18420/FGBS2024F-03
- Daniel Habicht, Yussuf Khalil, **Lukas Werling**, Thorsten Gröninger, and Frank Bellosa. 2024. Fundamental OS Design Considerations for CXL-based Hybrid SSDs. In *Proceedings of the 2nd Workshop on Disruptive Memory Systems (DIMES '24)*, November 2024. Association for Computing Machinery, New York, NY, USA, 51–59. https://doi.org/10.1145/3698783.3699380

1.7 Structure

The remainder of this work is structured into the following chapters:

Chapter 2 Background – Persistent Memory and File Systems (p. 21)

We first take a look at Intel Optane PM and how the Linux kernel supports PM, especially for use in file systems. We then give an overview of PM file systems that we evaluate for efficiency or crash consistency in later chapters.

Chapter 3 Motivation – Designing Data Structures for PM (p. 41)

We motivate our work in performance and crash consistency with a PM data structure for a file system.

Chapter 4 PM File System Efficiency (p. 55)

This chapter introduces our metrics for file system efficiency. We evaluate the metrics on multiple file systems and show that most PM file systems are not efficient under parallel accesses.

Chapter 5 PM File System Overload Mitigation (p. 73)

We then describe approaches for improving PM file system efficiency. We apply the approaches to a PM file system and compare their performance and efficiency.

Chapter 6 Userspace PM Access Accounting (p. 81)

The previous two chapters handled efficiency of PM access from the file system, which leaves out applications that map PM for direct access. In this chapter, we introduce an approach for accounting PM access from userspace applications with association of individual processes and PM devices.

Chapter 7 Crash Consistency Testing (p. 95)

We introduce fundamentals on crash consistency, discuss typical testing approaches, and review previous work.

Chapter 8 **Suvi**: Crash Consistency Testing for PM File Systems (p. 107)

We then present **Suvi**, our approach to crash consistency testing for file systems.

Chapter 9 **Suvi**: Implementation (p. 141)

We discuss details from **Suvi**'s implementation that are necessary for **Suvi**'s performance, including the tracer component, management of memory images, and parallelization.

Chapter 10 File System Testing with **Suvi** (p. 149)

We show how **Suvi** can test file systems in practice. We describe test selection, evaluate performance, and discuss analysis results.

Chapter 11 Conclusion (p. 159)

Finally, we conclude our work and discuss directions for future research.

Chapter 2

Background – Persistent Memory and File Systems

The goal of this thesis is to improve the efficiency and correctness of PM file systems. In this chapter, we introduce the storage stack on which this thesis builds. This includes Optane PM modules, PM support in the Linux kernel, several research PM file systems, and the crash consistency semantics of PM on x86 CPUs and for NVMe SSDs.

2.1 Optane Persistent Memory

Intel Optane Persistent Memory are persistent memory modules in the DIMM form factor.

Intel released three generations of Optane memory, shown in Table 2.1. Each generation was released together with a CPU from Intel's Scalable product line. Optane DIMMs of each generation were available in three sizes (128 GiB, 256 GiB, 512 GiB). Optane 100 and 200 both use the DDR-T interface which has the same physical form factor as DDR4. Optane 300 uses DDR-T2 with the physical form factor of DDR5 DIMMs. First-generation Optane PM was introduced in 2019. Intel discontinued Optane before the release of the third generation in 2022 [89].

In this thesis, we evaluate with 128 GiB Optane 100 DIMMs on Cascade Lake systems (see Section 3.2 and Section 4.2.1 for detailed system overviews). We discuss Optane performance in Chapter 3.

Optane PM has two layers of configuration. The `ipmctl` [2] tool configures *regions* across one or more Optane modules. This tool is specific to Optane PM. On top of regions, the Linux kernel manages PM *namespaces*, which are configured with

Generation	Year	Capacity [GiB]	CPU Generation
100 [8]	2019	128, 256, 512	Cascade Lake SP
200 [9]	2020	128, 256, 512	Ice Lake SP
300 [10]	2023	128, 256, 512	Sapphire Rapids SP

Table 2.1: Overview of Optane PM generations.

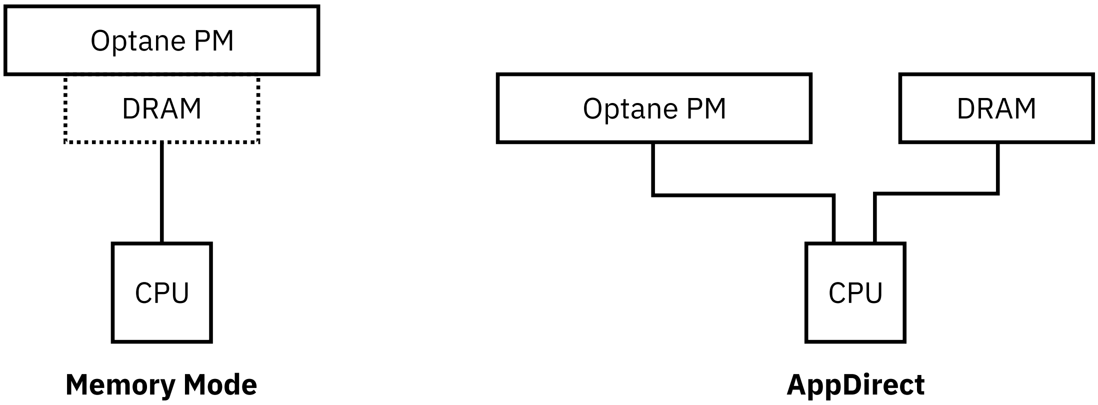


Figure 2.1: Logical view of Optane PM configured in Memory Mode and AppDirect mode.

`ndctl` [3]. Similar to partitions on block devices, namespaces allow separate management and configuration of parts of a PM region. Since namespaces are a concept by the Linux kernel, their use extends to other PM technologies. We discuss regions and namespace in more detail in the following sections.

2.1.1 Regions

A region spans one or more Optane modules connected to one CPU. Regions over multiple Optane modules are *interleaved*. The memory controller distributes accesses to an interleaved region over all modules. The interleaving size is 4 KiB [127]. Accesses over a sufficiently large memory area (e.g., 16 KiB with four interleaved modules) can therefore use the combined throughput of all modules.

There are two modes for regions: Memory Mode and AppDirect mode, pictured in Figure 2.1. In Memory Mode, the configured Optane capacity appears to the system as regular, volatile main memory. The memory controller transparently uses its connected DRAM as a cache to hide high access latencies to Optane. In this mode, Optane is only used for its larger capacity compared to DRAM, not for its persistence. In this thesis, we focus on Optane as persistent memory and do not further discuss Memory Mode.

In AppDirect mode, the Optane region appears separately to the system. The system firmware communicates the PM areas with the e820 memory map [44]. In the memory map, persistent memory areas are marked as type 7. Linux does not use these areas as main memory. Instead, they are handled by the NVDIMM subsystem [11], which manages namespaces as described in the following section.

On Linux, the e820 memory map can be overwritten in the kernel command line with the `memmap` parameter [71]. For example, passing a parameter `memmap=16G!2G` marks 16 GiB of memory starting at offset 2 GiB (physical address 2^{31}) as persistent memory. This is useful for simulating PM on systems lacking real Optane memory. We use this parameter for file system tests in **Suvi** (Chapter 8).

Configuration

Optane regions are configured by the system firmware as part of the boot process. With commands as given below, `ipmctl` can create provisioning goals, which are applied by the firmware after a reboot [2].

```
  SocketID | ISetID             | Type      | Capacity | Free  | HealthState
 ============================================================================
  0x0000   | 0x6e6dc3d053748888 | AppDirect | 504 GiB  | 0 GiB | Healthy
  0x0001   | 0x70efc3d0f3678888 | AppDirect | 504 GiB  | 0 GiB | Healthy
```

Listing 2.1: Shortened output from `ipmctl show -region` with two interleaved regions with four Optane modules.

```
# Configure all PM in AppDirect mode with interleaving.
ipmctl create -goal PersistentMemoryType=AppDirect
# Configure all PM in AppDirect mode with separate regions per module.
ipmctl create -goal PersistentMemoryType=AppDirectNotInterleaved
```

`ipmctl` also includes commands for reading the state of individual modules and the currently active regions. Listing 2.1 shows example output for reading the currently configured regions.

2.1.2 Namespaces

PM namespaces are managed by the Linux kernel and are configured with the `ndctl` tool [3]. A namespace has a size and one of the following modes:

raw The namespace appears as a block device with no extra kernel support for atomicity or direct access. This mode is not commonly used.

sector The namespace appears as a block device, with the kernel providing atomic block (or sector) writes. We describe the underlying mechanism in Section 2.2.2.

fsdax The namespace appears as a block device, with kernel support for direct access (DAX). This mode is required for kernel PM file systems.

devdax The namespace appears as a special character device that allows mapping the underlying PM into userspace processes. This mode is usually used for userspace PM file systems. For higher-level userspace PM software, it is usually preferrable to obtain a DAX mapping from a PM file system, which simplifies access permissions.

For fsdax and devdax namespaces, the kernel needs to allocate memory management metadata for each 4 KiB physical page frame (`struct page` with a size of 64 bytes [122]). A 128 GiB Optane module therefore requires 2 GiB of space for page metadata. The kernel can either allocate this metadata on PM (reducing the size of the namespace), or on regular main memory.

The Linux kernel can also use a devdax namespace as regular main memory. This can be configured with the `daxctl` tool [3]. In contrast to Optane's Memory Mode, PM configured this way appears separately from the system's DRAM as its own NUMA node.

2.1.3 Performance Counters

Performance counters are mechanisms that provide insight into how hardware components operate by counting specific events. There are two primary ways to obtain data on Optane operation: counters on the Optane modules and CPU performance counters with Optane events.

Optane Module Counters

Optane modules provide counters for four events [2].

```
---DimmID=0x0001---
   MediaReads=0x00000000000000000000006ae129ca2c
   MediaWrites=0x00000000000000000000005404670098
   ReadRequests=0x00000000000000000000000b33bafaea
   WriteRequests=0x000000000000000000000044d83a57c5
   TotalMediaReads=0x000000000000000000000ac3809dcff0
   TotalMediaWrites=0x0000000000000000000008b300ca5010
   TotalReadRequests=0x00000000000000000000011b2ece5a75
   TotalWriteRequests=0x00000000000000000000055a55b71f0b
```

Listing 2.2: Example output from `ipmctl show -performance`, showing on-DIMM performance counters.

MediaReads/MediaWrites count the number of 64 byte reads or writes to the media on the module.

ReadRequests/WriteRequests count the number of DDR-T read or write transactions (64 bytes).

All events are counted per DIMM since last boot and as lifetime total. These counters can be read with `ipmctl`. Listing 2.2 shows an example output. We analyze output from these counters in Chapter 3.

CPU Counters for PM

Intel Cascade Lake CPUs (the CPUs we use in this thesis) provide three types of counters that can count events associated with Optane [63, search for "PMM"]. On-core events are counted within a core as part of instruction execution. There are only two on-core PM events that count L3 cache misses from load instructions, which are either fetched from local PM (`MEM_LOAD_RETIRED.LOCAL_PMM`) or from remote PM (`MEM_LOAD_L3_MISS_RETIRED.REMOTE_PMM`).

There are no on-core events for stores to PM. Since x86 CPUs have write-back caches, the CPU core writes only to its L1 cache when executing a store instruction. When a cache line is evicted from the L3 cache and the store reaches PM, there is no direct association with the original store instructions anymore. Similarly, non-temporal stores are handled by write-coalescing buffers.

The second type of counters are off-core response counters [61, Vol. 3B §20.3.8.3.1]. These counters count specific interactions between the on-core L2 cache and the L3 cache or memory. The counted events are selected with a combination of a request type (e.g., data or code read, prefetches) and a supplier (e.g., L3 cache, DRAM, PM). Similar to the on-core counters, the off-core response counters cannot provide information about writes to PM.

Finally, uncore counters provide events from the memory controller [55, 60].[1] The available events include commands issued to PM (e.g., reads and writes) and the state of read and write request queues. From the read and write command counts, the total read and write bandwidth to PM can be calculated: each command transfers 64 bytes.

[1] There are also uncore events for components other than the memory controller, but these are unrelated to PM.

2.2 Linux File System Support for PM

This thesis deals with PM file systems in the Linux kernel. In this section, we therefore introduce the interfaces that Linux provides for PM access from both kernelspace and userspace.

The upstream Linux kernel [12] supports PM for file systems in two ways: a translation layer for traditional block-based file systems and DAX support in some file systems.

2.2.1 Accessing PM from the Kernel

A PM namespace configured as fsdax appears as a block device `/dev/pmemX` in a Linux system.[2] The NVDIMM driver implements the block device access function `pmem_submit_bio`.[3] It supports basic read and write requests, which are fulfilled with a memory copy routine. Writes to PM are performed with non-temporal stores on x86 (via `memcpy_flushcache`). The flags `REQ_PREFLUSH` and `REQ_FUA` trigger a memory fence before or after the operation.

Consequently, file systems using the block I/O layer in Linux (*bio*) do not require modifications to run on top of PM. However, there are two major differences compared to traditional block devices (e.g., those using SCSI or NVMe). First, this thin compatibility layer does not provide any crash consistency guarantees beyond those of the underlying PM. The Block Translation Table, described below, solves this problem by implementing atomicity for larger blocks.

Second, rather than submitting the I/O request to a device and waiting for the result, the compatibility layer performs a synchronous memory copy on the kernel thread that submitted the *bio* request. This has consequences for the performance and energy efficiency of file systems on top of PM, which we discuss in Chapter 5.

File systems that target PM specifically can obtain a pointer to the PM namespace to bypass the *bio* compatibility layer. The kernel provides the function `dax_direct_access` for this purpose. Listing 2.3 shows an example of how to use this function. PM file systems such as NOVA and PMFS (introduced below) obtain a PM pointer once during initialization and then access PM exclusively through that pointer.

2.2.2 Block Translation Table (BTT)

With the Block Translation Table (BTT) [72, 117], the kernel provides atomic block access to traditional file systems. These file systems expect atomicity for full block updates, which usually have a size of 512 bytes or 4 KiB. The BTT provides this block atomicity with an indirection layer that maps logical block addresses (LBA) to PM offsets. Every update to a logical block allocates a new data block on PM. A journal ensures that the mapping from the old to the new data block changes atomically.

The BTT is sufficient for using PM via arbitrary file systems, but it cannot provide direct access (DAX) to PM for userspace applications. For this reason, the Linux file system developers implemented DAX support in two traditional file systems, ext4 and XFS [116].

[2] For example, an fsdax namespace called `namespace1.0` appears as `/dev/pmem1`.

[3] File `drivers/nvdimm/pmem.c` [12]

```
static int get_pm_pointer(
  struct block_device *bdev, /* input: handle to block device */
  void **pm_virt_addr,       /* output: virtual address */
  long *pm_size              /* output: size in bytes */
)
{
  pfn_t pfn;
  struct dax_device *dax_dev;
  int ret;

  /* Obtain handle to a DAX device. */
  dax_dev = fs_dax_get_by_bdev(bdev);
  if (!dax_dev) { return -EINVAL; }

  /* Obtain pointer to DAX area. */
  *pm_size = dax_direct_access(dax_dev,
    0, LONG_MAX / PAGE_SIZE, /* offset and maximum size */
    pm_virt_addr, &pfn       /* output: address and page frame number */
  ) * PAGE_SIZE;             /* return: number of pages */
  if (*pm_size <= 0) { return -EINVAL; }

  return 0; /* success */
}
```

Listing 2.3: Example for how a file system can obtain a pointer to PM from a block device handle in Linux 5.15.[4]

2.2.3 Ext4 and XFS without BTT

Both ext4 and XFS implement journaling to protect metadata (and optionally file data) updates from crashes. Originally, ext4 implemented *physical journaling* with JBD2 [112]. Physical journaling takes place at the level of on-disk data blocks: Before overwriting a protected block, the file system stores a copy of either the old or the new block in the journal. After a crash, the file system recovers by copying the blocks in the journal to their intended disk locations.

In contrast, XFS employs *logical journaling* [112]. The entries in a logical journal describe individual file system operations. A logical journal is therefore more compact than a physical journal, leading to better runtime performance. However, crash recovery is more complex than with a physical journal, since the file system needs to replay the operations in the journal. With FastCommit [112], ext4 now also performs logical journaling for certain file system operations (e.g., file creation and deletion, appending data to a file).

Both XFS and ext4 protect journal entries with checksums. The journals are therefore already sufficiently protected from partially-written blocks after a crash. Since neither file system relies on atomic block updates without journaling for metadata updates, it is safe to use ext4 and XFS on PM without the Block Translation Table.

Previous works have analyzed the behavior of these file systems on top of traditional block devices under crashes. Mohan et al.'s CrashMonkey [93] found no crash consistency bugs in ext4 and XFS when injecting crashes at persistence points (i.e.,

[4] Note that although the parameters to these functions change over time, the overall process remains the same.

`fsync()` or `sync()`). Jaffer et al. evaluated the behavior of ext4 under certain faults, including incompletely written blocks ("shorn writes") [65]. They show that ext4 can recover from such faults, although with partial data loss in some cases. Consequently, ext4 and XFS were already well-equipped for handling crashes before the introduction of PM.

2.2.4 DAX Support

Without the indirection from the Block Translation Table, ext4 and XFS can hand out memory mappings to file data residing on PM to userspace applications. This feature is called DAX (for direct access) in Linux.

If the underlying device supports DAX and no conflicting file system features are enabled (e.g., file system encryption), a flag in the inode controls whether DAX-mapping a file is allowed. Alternatively, the file system mount option `dax=always` overrides the inode flag and enables DAX for all files [73].

If a userspace process requests a shared memory mapping of a file with the DAX flag set, the file system will always map the file's PM pages directly. A process can ensure that a mapping is a DAX mapping by calling `mmap` with the `MAP_SYNC` flag [13]. If the file's DAX flag is not set, `mmap` would then return the `EOPNOTSUPP` error. Since the `MAP_SYNC` flag does not otherwise change the memory mapping, it is not possible to obtain a "traditional" mapping via the page cache for files that support DAX [50].

2.3 PM File Systems

PM's byte-addressability provides new opportunities for file system design. PM file systems no longer need to organize their data in fixed-size blocks as dictated by the underlying block storage device. This allows file systems to offer crash consistency guarantees at a finer level of granularity.

In Table 2.2, we provide an overview of research PM file systems that are relevant for this thesis. We compare the following features:

Kernelspace/Userspace Whether the file system runs completely in kernelspace (K) or has a userspace component (U).

Cross-Media Whether the file system supports storing data on traditional block storage in addition to PM. This property is relevant for **Suvi**, which can test crash consistency properties of cross-media file systems.

PM Bandwidth Control Whether the file system actively limits its bandwidth when writing to PM. We propose mechanisms for PM bandwidth control in this thesis.

Strong Consistency We consider a file system to implement strong consistency if all individual file system operations are immediately and atomically persisted to PM. In contrast, traditional file systems implement delayed persistence and only guarantee that data is retained after a later call to `fsync()` [14]. **Suvi** can automatically test strong consistency properties of file systems.

Artifact Available Whether an artifact with the file system's source code is publicly available. We need access to the source code in order to evaluate crash consistency with **Suvi** or to compare PM bandwidth control mechanisms.

We describe ZIL-PMEM in Chapter 3 and give a more detailed overview of the other file systems in the following sections.

File System	Year	Kernel/User Space	Cross-Media	PM Bandwidth Control	Strong Consistency	Artifact Available
PMFS [41]	2014	K	✗	✗	✓	✓
Aerie [118]	2014	U	✗	✗	✗	⚠
NOVA [125]	2016	K	✗	✗	✓	✓
NOVA-Fortis [126]	2017	K	✗	✗	✓	✓
Strata [79]	2017	U	✓	✗	⚠	✓
Ziggurat [132]	2019	K	✓	✗	(✓)	⚠
SplitFS [67]	2019	U	✗	✗	✓	✓
WineFS [66]	2021	K	✗	✗	✓	✓
ZIL-PMEM [111]	2021	K	✓	✓	✓	✓
SPMFS [128]	2021	U	✗	✓	(✓)	✗
OdinFS [134]	2022	K	✗	✓	✓	✓
Assise [25]	2022	U	✓	✗	✗	✓
Trio [133]	2023	U	✗	✓	⚠	✓
P2CACHE [84]	2023	K	✓	✗	⚠	✓
SlotFS [131]	2023	U	✗	✗	⚠	✓

Table 2.2: Overview of research PM file systems with a comparison of features relevant for this thesis. ⚠ denotes problems with the file system implementation's crash consistency guarantees that are obvious without detailed analysis.

2.3.1 PMFS, WineFS, and OdinFS

PMFS [41] was the first file system designed for persistent memory in current x86 systems and implemented as a Linux kernel file system. Its data layout is optimized for byte-addressable PM. In particular, its allocator manages data in blocks according to the processor's page sizes (4 KiB, 2 MiB, 1 GiB) to support DAX mappings. Metadata is updated in-place with atomic instructions if possible. Otherwise, PMFS implements fine-granular journaling with cache-line-sized log entries (64 bytes). Writes to file data pages are protected with a copy-on-write mechanism.

Since PMFS was designed before hardware with support for PM was available, the authors had to make assumptions about the interaction of atomic instructions with PM [41, §3.2]. Besides 8-byte atomic updates, PMFS also uses `cmpxchg16b` for atomic 16-byte updates, and optionally transactional memory for atomic 64-byte updates. Real hardware ended up guaranteeing atomicity only for 8-byte updates. In our analysis in Chapter 10, we can therefore observe crash consistency bugs in two places where PMFS uses `cmpxchg16b` to update multiple fields in the inode:

- File size (8 bytes) and modification timestamps (2 × 4 bytes)
- Root pointer and height of the B-Tree referencing the file data (both 8 bytes)

PMFS is used as the basis of multiple later research file systems. These file systems inherit PMFS's usage of `cmpxchg16b`. In the following, we introduce two such file systems, WineFS and OdinFS.

WineFS

WineFS [66] improves the performance of memory-mapped files in *aged* file systems. Repeated allocation and deletion over time leads to a fragmented PM area. Since PM is byte-addressable and has a (mostly) uniform access latency, such fragmentation usually does not have an effect on access performance. However, fragmentation is relevant for memory-mapped files. If sufficiently large parts of the data on PM are contiguous and aligned, the file system can employ hugepages when memory mapping the file to a userspace process.

The authors of WineFS show that previous PM file systems suffer from reduced access bandwidth once the file system cannot use hugepages anymore due to fragmentation. They introduce an alignment-aware allocator that avoids fragmentation over time. The authors evaluate the throughput as well as application benchmarks on aged file systems. WineFS outperforms the other PM file systems in these benchmarks.

We analyze WineFS with **Suvi** in Chapter 10 and show that some but not all crash consistency issues of PMFS are fixed in WineFS.

OdinFS

As we show in Section 2.1, Optane PM suffers from low bandwidth with parallel accesses, as well as remote NUMA accesses. OdinFS [134] extends PMFS with a delegation mechanism for accessing PM to mitigate these problems. For each NUMA node, OdinFS runs a fixed number of delegation threads. Every read or write access to PM is delegated to a thread on the same NUMA node as the data. OdinFS therefore both limits the amount of threads that access PM in parallel, and prevents remote NUMA access to PM.[5]

In Chapter 5, we discuss alternative approaches for mitigating performance loss from parallel PM accesses. We show that although OdinFS's delegation threads ensure a constantly high PM bandwidth, they come with a high CPU cost, decreasing the overall efficiency of the file system.

OdinFS additionally stripes data across all available NUMA nodes. It thereby spreads the load evenly across the PM modules and allows higher parallelism, including for large accesses from a single thread.

Since OdinFS inherits all metadata management from PMFS, we expect identical crash consistency behavior. We therefore do not analyze OdinFS separately with **Suvi**.

2.3.2 Aerie

Aerie [118] was the first modern userspace PM file system. It introduces the concept of a user-mode library file system with direct PM access in combination with a

[5] Note that although OdinFS might increase the number of remote DRAM accesses, these do not hurt performance as much as remote PM accesses.

trusted file system service running partially in the kernel. The library file system has direct read-only access to metadata and file data, as well as a write area for new file data. To write a file, the library file system first writes the file data to PM, then requests a metadata update from the trusted file system service. Aerie can delay metadata updates, so file operations with outstanding metadata may be lost in the event of a crash.

Aerie was implemented for x86 Linux systems, but predates support for PM in processors and the Linux kernel. In contrast to PMFS, its artifact [1] was never updated with support for real PM hardware and Linux DAX interfaces. It is therefore not usable for performance comparisons or crash consistency testing.

Concepts from Aerie can be found in most later userspace file systems. We describe some of them in the following sections.

2.3.3 NOVA, NOVA-Fortis, and Ziggurat

NOVA [125] is a log-structured file system for PM. Its primary goals are high performance and strong consistency. NOVA is implemented as a Linux kernel file system.

NOVA's data layout is optimized for concurrent access by maintaining private data structures for each CPU, including inode tables, journals, and free lists. Traditional log-structured file systems maintain a global log containing all file system data, allowing mostly linear write access to the storage media [106]. PM enables efficient random access with small access sizes, allowing NOVA to maintain a private log for each inode. This enables concurrent log operations on different inodes. For operations involving multiple inodes, NOVA uses journaling.

To speed up file access, NOVA maintains volatile runtime data in DRAM, including the file radix tree, the directory entry tree, and free lists. These data structures do not require persistence and are always rebuilt during mounting.

NOVA does not use logging for all inode data. If possible, inode fields are updated directly with 8-byte atomic store instructions (e.g., timestamps and log pointers). File data is updated with copy-on-write, by writing the data to a new data page and updating data pointers through the log.

NOVA-Fortis

NOVA-Fortis [126] extends NOVA with features for protecting the file system from software and hardware faults. It implements consistent whole file system snapshots, including for files that are DAX-mmapped during the snapshot operation.

NOVA-Fortis can handle PM media errors. All file system metadata and file data in NOVA-Fortis is protected with a checksum. Using ECC, the PM hardware can transparently correct certain media errors and detect others. The CPU communicates detected errors via *machine check exceptions* (MCE). In case of an MCE or a checksum mismatch, NOVA-Fortis recovers metadata from a replica and file data using RAID-4 parity. However, it cannot protect file data while it is DAX-mmapped and defers responsibility for data protection to the userspace application.

NOVA and NOVA-Fortis were introduced before Optane PM was available. The authors continued work on the artifact [6], updating it to Linux 5.1 and ensuring compatibility with commercial Optane modules. In this thesis, we use NOVA as a baseline for a PM file system without managed concurrency for write accesses in Chapter 5. We then evaluate our mechanisms for limiting concurrency with NOVA. NOVA and NOVA-Fortis are also part of our crash consistency analysis in Chapter 10.

Ziggurat

Ziggurat [132] extends NOVA with support for cross-device storage. In contrast to other cross-device file systems that use PM as a cache for a disk file system (e.g., ZIL-PMEM or P2CACHE, both described below), Ziggurat prefers to place data in PM.

A *synchronicity predictor* decides whether new file data is placed on PM or a lower tier. The predictor bases its decision on previous `fsync()` calls, the size of the new data, and the size of previous accesses. Data placed in lower tiers is written asynchronously using a page cache in DRAM. Ziggurat therefore does not implement strong consistency for all writes. However, applications can request strong consistency by opening a file with the `O_DIRECT` flag.

Data placement in Ziggurat is not fixed. A migration thread can move hot data from disk to PM and cold data from PM to disk. In particular, background migration is helpful for small writes which are placed in PM at first, then later coalesced into a large, sequential write to disk.

We were unable to analyze the crash consistency of Ziggurat with **Suvi**. Although Ziggurat's source code is published, we found that it is not in a usable state.

2.3.4 Strata and Assise

Strata [79] is the first modern PM-based cross-media file system. A userspace component *LibFS* handles writes by logging directly to a private PM area mapped into the process. The kernel component *KernelFS* then creates a digest of the userspace log. This digest can then be placed either in PM or in lower storage tiers. Strata also supports data migration between storage tiers based on access patterns.

Assise [25] is a distributed file system that builds on Strata. Assise uses Strata's design for node-local storage and extends it with a cache coherence layer *CC-NVM* for replication. Using RDMA, Assise replicates the write log to different nodes, allowing the file system to survive node failures.

LeBlanc et al. attempted to analyze Strata and Assise with Chipmunk, but learned that neither artifact supports crash recovery [81, §4.1]. We therefore do not analyze these file systems with **Suvi**.

2.3.5 SPMFS

SPMFS [128] is a userspace file system featuring PM bandwidth management. It implements most file system operations in its userspace component, with a kernel module handling some "complex" metadata operations as well as free space management. Notably, the authors do not discuss security considerations in case multiple processes share a file system.

SPMFS implements an I/O thread pool to limit parallel write accesses to PM. Applications may not write to PM directly and insert their requests into a queue instead. Based on a predictor, SPMFS then either handles the request synchronously or asynchronously.[6] Synchronous requests are immediately processed by an I/O thread, with the application waiting for completion. Asynchronous requests are inserted into a write cache in DRAM. Similar to the kernel's page cache, SPMFS can serve reads from the write cache and will write back data if `fsync()` is called. SPMFS therefore implements delayed persistence by default, but applications can request strong consistency with the `O_SYNC` flag.

The authors evaluate SPMFS's PM bandwidth management by comparing the parallel synchronous write throughput with and without the I/O thread pool [128, Fig. 6]. They show that the total bandwidth increases with the number of I/O threads and does not decrease as much as the amount of threads increases.

In contrast to our analysis of similar mechanisms in Chapter 5, the authors of SPMFS do not analyze the CPU efficiency of their approach. Since they did not publish any source code, we cannot directly compare SPMFS to our approach.

2.3.6 Trio

Trio [133] is an architecture for secure userspace file systems. Its key feature is an in-kernel access controller that mediates PM access to a userspace library file system. If multiple processes want to access a shared file, the kernel controller ensures exclusive write access. Once a process releases its write access to a file, a verifier checks that the file's metadata is consistent.

This design allows different file system implementations that agree on a core metadata format as enforced by the verifier. Trio's authors implement three library file systems: a POSIX-compatible file system ArckFS, and two customized file systems KVFS and FPFS with a key-value interface and with optimizations for deep directory structures.

To limit parallel accesses to PM, ArckFS implements the delegation mechanism from OdinFS [134]. For bulk data accesses, the library file system sends write requests to in-kernel delegation threads. To reduce overhead, it performs small PM accesses directly from userspace. Since the underlying mechanism is the same as in OdinFS, we do not evaluate ArckFS in Chapter 5.

We attempted to analyze the crash consistency of ArckFS with **Suvi**, but found a number of issues with the published artifact:

- ArckFS requires large amounts of PM and DRAM for statically-allocated metadata Reducing the size of these allocations is not trivial because of implicit alignment requirements.
- We ran into multiple bugs when accessing ArckFS from multiple sequential processes like in shell scripts. Such a scenario does not come up in the authors' evaluation since they only evaluated single long-running benchmark programs.

[6] The predictor appears very similar to the one originally introduced in Ziggurat [132], which we describe above.

- Support for the POSIX API is incomplete. For example, ArckFS requires custom functions for accessing directories instead of `opendir()` and `readdir()`.
- ArckFS always formats the file system on mount. We removed this unconditional formatting, but were unable to successfully remount an initialized (and properly unmounted) file system.

These problems prevented us from successfully running **Suvi**'s test cases, making a crash consistency analysis impossible.

2.3.7 P2CACHE

P2CACHE [84] is a PM-based cache designed to improve the performance of traditional kernel file systems such as ext4. It sits between the VFS layer and the file system and records all file system operations in a persistent write-ahead log. Additionally, it caches all writes in DRAM to serve reads, similar to the page cache.

The authors evaluate P2CACHE on top of ext4, comparing with ext4, XFS, and NOVA. They show that P2CACHE is significantly faster than the traditional Linux file systems ext4 and XFS, both on an NVMe SSD and on PM. It also surpasses NOVA in some benchmarks.

P2CACHE does not implement any PM bandwidth control. The authors show results of a scalability benchmark [84, Fig. 10], but their evaluation system is not set up in a way where Optane contention becomes an issue.[7]

The authors designed P2CACHE to support strong consistency. They developed a custom crash consistency checker to check for crash consistency bugs [84, §4.1]. Unfortunately, these consistency checks were not part of the published artifacts and therefore not reproduced in the artifact evaluation [84, §A.3].

We attempted to evaluate P2CACHE with **Suvi**, but found that the published artifact does not support retaining any data and always formats the PM with an empty file system on mount.

2.3.8 SlotFS

SlotFS [131] is a userspace log-structured file system. SlotFS improves upon prior log-structured file systems such as NOVA by implementing *scattered logging*. Since random access to PM is cheap, there is no need to keep a contiguous log. According to the authors, scattered logging completely eliminates overhead from garbage collection of log entries, as individual freed log entries can immediately be reused.

We attempted to evaluate the crash consistency of SlotFS with **Suvi**, but found a number of issues with the published artifact that prevented an analysis:

- The required Linux kernel is unclear. According to the artifact description [131, §A.2.3], SlotFS requires Linux kernel version 5.1.0 with patches for SplitFS [67] and Hodor [52]. Hodor is a system for isolating components in a userspace process using memory protection keys. SlotFS uses Hodor to isolate its library file system from untrusted processes that access the file system. However, the Hodor patches are only available for kernel version 4.15 or 5.4. We found that the published artifacts do not actually use Hodor in any way, so we attempted evaluation without Hodor.

[7] The evaluation system has four Optane DIMMs attached to a 12-core CPU without hyperthreading.

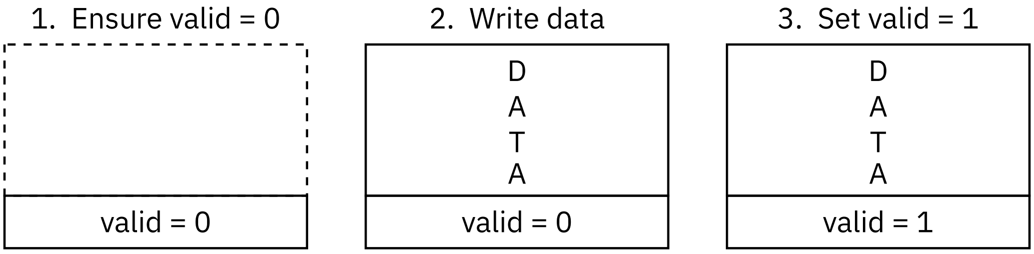


Figure 2.2: Steps when writing a journal entry. For crash consistency to hold, these steps must not be reordered.

- The recovery code is disabled and the SlotFS initialization code always formats the file system. We were able to remove the forced formatting.
- A routine performing cache flushes for a memory range with an optional memory fence was disabled. This routine is used for some metadata updates (e.g., in inodes and directory entries). We re-enabled the routine by removing an early return statement.
- SlotFS expects a PM area with size 48 GiB by default, which is too large for an analysis with crash images. It cannot automatically scale its metadata allocation to smaller PM areas.
- We observed crashes when accessing the file system from multiple consecutive processes, such as in shell scripts. SlotFS sets up a volatile shared memory area that is retained across consecutive processes. Some retained state appears to cause crashes.

Even with multiple fixes to the issues above, we were unable to run basic file system tests on SlotFS without crashes. An analysis of crash consistency properties is therefore not possible.

2.4 Crash Consistency

Crash-consistent software needs to control the order in which its writes reach non-volatile memory. As motivating example, consider a journal as commonly used by file systems or databases to perform complex updates atomically. For this example, each journal entry contains arbitrary data, followed by a valid flag. Writing a journal entry works in three steps, pictured in Figure 2.2:

1. Ensure that the valid flag is not set, for example by zero-initializing the journal memory.
2. Write the data. There is no requirement for atomic writes so that an arbitrary amount of data may be written.
3. Set the valid flag.

A recovery procedure would ignore journal entries that do not have the valid bit set. For a crash-consistent and atomic journal, we therefore must ensure that valid is only set if the data is written completely. In the following sections, we describe the mechanisms available for persistent memory and NVMe to ensure such an ordering. In Chapter 7, we introduce crash consistency testing.

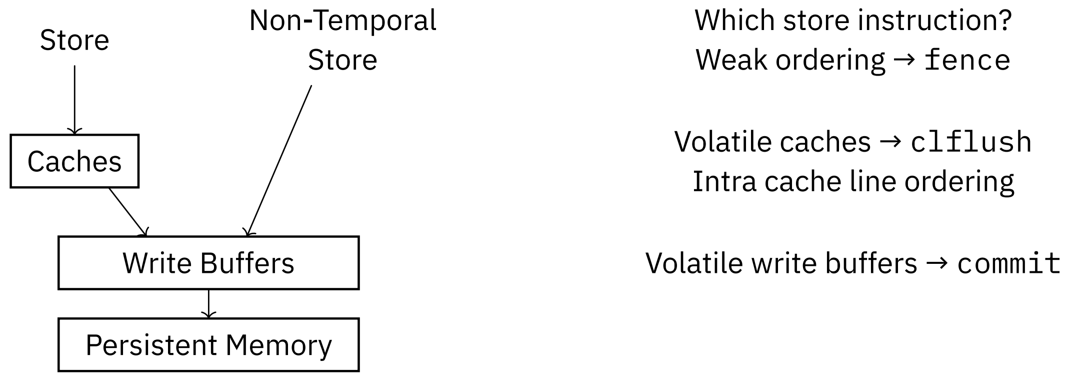


Figure 2.3: Generic write path from store instruction to persistent memory. Depending on ISA, choice of store instruction, and presence of volatile buffers, different persistence primitives are necessary.

2.4.1 Crash Consistency for Persistent Memory

Since persistent memory is part of the CPU's memory hierarchy, its crash consistency semantics are defined by the CPU's instruction set architecture (ISA). The ISA needs appropriate extensions to support PM, as control over when writes reach the underlying memory is not necessary with volatile DRAM.

Figure 2.3 shows a generic memory write path and the persistence primitives that may be necessary to ensure flushing volatile buffers and correct ordering. A normal store instruction writes its data to the CPU's cache hierarchy. The ISA may offer non-temporal store instructions that skip the caches. These instructions may be weakly ordered, meaning that the CPU may reorder them. For example, normal x86 stores are strongly ordered (i.e., they will not be reordered), while non-temporal stores are weakly ordered. For weakly ordered instructions, the programmer needs to insert memory fence instructions to force a certain order.

For stores going to volatile CPU caches, the programmer needs to ensure that modified cache lines reach persistent memory. ISAs offer cache line flush (e.g., x86 `clflush`) or write-back (e.g., x86 `clwb`) instructions for this purpose. Although these instructions generally operate on full cache lines (usually 64 bytes), the CPU microarchitecture might not guarantee atomic transfers of full cache lines to persistent memory. In this case, knowing the ordering of writes within a cache line is important. On systems with persistent caches, these steps are not necessary and cache line flush instructions are optional.[8]

The memory controller finally collects flushed cache lines and non-temporal stores in write buffers. The primary purpose of these buffers is to coalesce writes that are smaller than what the underlying memory expects. If these write buffers are volatile, an additional "commit" instruction is necessary that instructions the memory controller to flush its buffers.

Table 2.3 shows an overview of these properties for x86 with and without eADR. We discuss the properties and the persistence primitives in the following sections.

[8] Even with persistent caches, non-temporal stores may still offer better performance than cached stores since they may reduce the number of requests to PM (see Section 3.3).

property	**x86**	**x86 with eADR**
cached stores	strong	
non-temporal stores	weak	
caches	volatile	persistent
write buffers	persistent	

Table 2.3: Overview of ordering and persistence properties in x86 and ARM. ARM supports processors with either persistent or volatile caches and write buffers.

Crash Consistency Primitives on x86

Normal stores to caches are strongly ordered in x86, so they will always reach the caches in program order [61, Vol. 3A §9.2]. However, volatile caches do not guarantee that stores reach persistent memory in a particular order. Cache line flush (`clflush`, `clflushopt`) or write-back (`clwb`) instructions trigger an immediate flush of stores to a particular cache line to memory. The write-back instruction `clwb` may leave the line in the cache, allowing following reads to fetch the line from cache rather than PM [61, Vol. 2]. Finally, `wbinvd` flushes all cache lines to memory, but may only be used from kernel mode. Since `wbinvd` clears all cache lines, it has a big impact on performance and is thus rarely useful for persistence.

Non-temporal (NT) stores offer an alternative to normal stores in combination with cache flushes. NT stores are weakly ordered on x86 and require use of memory fence instructions for ordering [61, Vol. 3A §9.2]. The primary fence instruction for ordering stores for persistence is `sfence` [61, Vol. 2]. It ensures that all store instructions before `sfence` have completed before any store instructions after `sfence` are executed. `sfence` is the preferred instruction for ordering stores to persistent memory since it does not enforce an ordering of other types of instructions. However, any serializing instruction [61, Vol. 3A §9.3] (e.g., `cpuid`) may be used to enforce ordering as well.

Similar to NT stores, the cache flush instructions `clflushopt` and `clwb` are also weakly ordered [61, Vol. 2].[9] The processor ensures that older writes to the cache line finish first, but does not enforce an ordering with newer writes or other cache flushes. These two instructions are thus also usually used in combination with `sfence`.

Although cache flushes operate on 64 byte cache lines and NT stores may have a size of up to 64 bytes as well, Intel originally only guaranteed atomicity for aligned 8 byte stores [58]. On power failure, a cache line or large NT store might thus tear so that only parts of it end up on persistent memory. Intel informally guaranteed *intra cache line ordering* [109]: The order of 8 byte stores to a single cache line is always preserved. The introduction of the `movdir64b` instruction, which copies 64 bytes from one memory address to another, finally offers atomicity for larger 64 byte writes [61, Vol. 2]. Since this instruction was only introduced recently with Sapphire Rapids (2023), software written for Optane PM does not use it (see Section 2.1).

Originally, Intel added a `pcommit` instruction to x86 for flushing write buffers in the memory controller (called Write Pending Queue) [108]. With the introduction of Asynchronous DRAM Refresh (ADR), the need for this instruction disappeared. In

[9] Weak ordering is preferable for performance if multiple cache lines need to be flushed at once.

the event of a power failure, ADR ensures that the Write Pending Queue is flushed to PM. Since all CPUs with support for persistent memory also had support for ADR, `pcommit` was never supported.

x86 with eADR

With eADR, some Intel CPUs extended the persistence domain to include the caches as well [59]. These CPUs guarantee that they flush any dirty cache lines to persistent memory on power failure, effectively making the cache contents persistent.

Persistent caches simplify the programming model. Cache flush instructions are no longer necessary. Reordering of stores is no longer a concern for strongly ordered cached writes. Memory fences are therefore only needed for weakly ordered non-temporal stores.

Although eADR eliminates an important source of crash consistency issues, programmers still need to be careful. Misplaced memory fences with NT-stores lead to invalid states. Even a program that only uses cached stores might exhibit crash consistency bugs. Developers need to ensure that the application state stays valid with every store. Additionally, compilers might emit stores in an unexpected order.

2.4.2 Crash Consistency for NVMe

The NVMe standard [4, 5] defines an interface for communication between the operating system and a PCIe-attached non-volatile memory device. It is commonly used for modern flash-based SSDs. In the following, we take a look at two NVMe features which are critical for crash consistency. First, the asynchronous command processing that allows arbitrary reordering of write commands. Second, its support for a volatile write cache that requires *Flush* commands. In sum, we end up with a crash consistency model for NVMe that shares similarities with the model for persistent memory.

NVMe Command Processing

NVMe is an asynchronous protocol. The operating system communicates with the NVMe device over ring buffers in main memory. Entries are written to the tail of the ring buffer and consumed from the head. The operating system writes commands into a submission queue and the device writes completions into a completion queue. The complete process for one command is as follows [5, §3.3.1]. NVMe also specifies variants of this protocol, for example to allow polling for completions instead of interrupts.

1. The operating system writes a command into a submission queue and increments the queue’s tail pointer.
2. The operating system notifies the device about the new entry by writing to the *Submission Queue Tail Doorbell*, a memory-mapped device register.
3. The device consumes the command by reading it from the submission queue and incrementing the queue’s head pointer.
4. The device processes the command.
5. The device writes a completion entry to the completion queue and increments the queue’s tail pointer.
6. The device notifies the operating system by triggering an interrupt.

7. The operating system consumes the completion by reading it from the completion queue and incrementing the queue's head pointer.
8. The operating system notifies the device about the processed completion by writing to the *Completion Queue Head Doorbell*.

By enqueuing multiple commands at once, the operating system can reduce overhead from this protocol. Additionally, the SSD can then process the commands in an order that fits its internal organization. NVMe allows arbitrary reordering for most commands, including those that operate on the same data [5, §3.4.1].

NVMe I/O Commands

The NVMe specification defines two main commands, *Read* and *Write*, for accessing data [4, §3.2]. These commands take two main arguments [4, §3.2.4, §3.2.6].

Data Pointer (DPTR) The DPTR field specifies where in physical memory the OS receives data from *Read* or provides data for *Write*. It contains two *physical region page* (PRP) entries. A PRP entry is a 64 bit pointer to either a single data page or (for large transfers) to a page containing more PRP entries [5, §4.1.1].

Starting LBA (SLBA) The logical block address (LBA) that specifies where data is read from or written to.

The data on an NVMe device is organized in fixed-size blocks. The block size is configurable with the *Format NVM* admin command [5, §5.14]. Typical block sizes are 512 bytes and 4096 bytes. A single *Read* or *Write* command can transfer multiple blocks at once, up to a limit given by the NVMe controller [5, Fig. 276]. For example,

Presence of a volatile write cache is indicated with the *Identify Controller* NVMe command. If bit 0 of field `vwc` ("Volatile Write Cache Present") is set, then a volatile write cache is present [5, Fig. 276].

The volatile write cache can be disabled with the *Set Feature* NVMe Command (Dword 11 "Volatile Write Cache Enable (WCE)") [5, §5.27.1.4].

Example nvme-cli [15] commands with shortened output that query presence and status of the volatile write cache:

```
$ nvme id-ctrl /dev/nvme0
NVME Identify Controller:
[...]
mn        : Samsung SSD 980 1TB
[...]
vwc       : 0x7
[2:1] : 0x3   The Flush command supports NSID set to FFFFFFFFh
[0:0] : 0x1   Volatile Write Cache Present
[...]

$ nvme get-feature -H /dev/nvme0
[...]
get-feature:0x06 (Volatile Write Cache), Current value:0x00000001
        Volatile Write Cache Enable (WCE): Enabled
[...]
```

Listing 2.4: NVMe interfaces to check for presence of a volatile write cache.

a Samsung SSD 980 supports transfers of up to 512 blocks of 512 bytes, totalling 256 KiB per command.

Every data block on the SSD may have an additional metadata area that is transferred separately from the main data [4, §5.8.3]. Metadata is necessary for end-to-end data protection [4, §5.2]. Linux file systems do not directly make use of it. In this thesis, we thus assume SSDs configured without metadata.

A number of additional NVMe commands may modify data on the SSD, such as *Write Zeroes* and *Copy*. However, we found that Linux file systems do not use these commands. We thus limit our analysis to *Read* and *Write*.

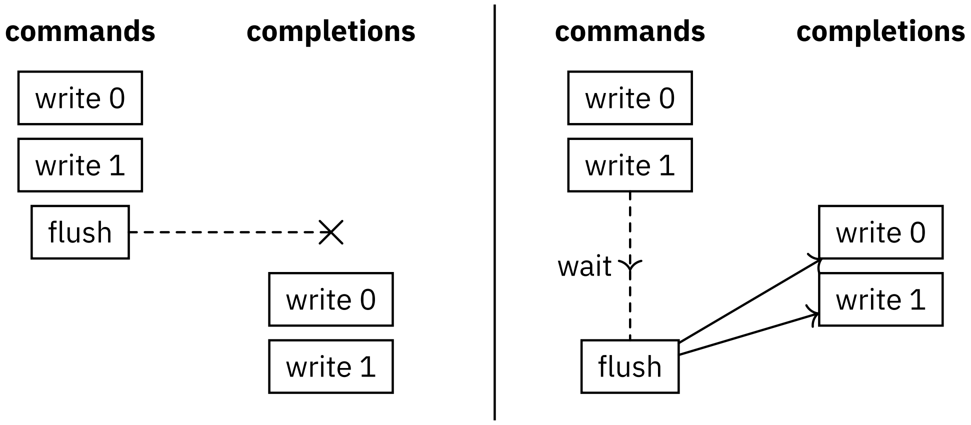
NVMe Volatile Write Cache and Flush Command

NVMe devices can have an optional volatile write cache to hide the high latency of writes to their flash memory. Listing 2.4 shows NVMe management commands to query presence and status of this volatile write cache. In the event of a crash, the contents of the volatile write cache may be lost. For this reason, NVMe provides a *Flush* command that instructs the SSD to write the contents of the cache to its backing memory.

Thus, the operating system needs to issue *Flush* commands after writes and wait for their completion before it can signal that a write has reached non-volatile memory (e.g., as part of an `fsync(2)` system call). Since NVMe does not specify dependencies between commands, special care is required to ensure that the *Flush* command covers all writes as intended.

> The flush applies to all commands [...] completed by the controller prior to the submission of the Flush command.
>
> — NVMe Base Specification, Section 7.1 [5]

Figure 2.4 shows an example of this behavior. If the operating system submits the *Flush* command together with the *Write* commands (a), the *Flush* does not apply to the previous writes. A correct implementation (b) waits for completion entries before submitting the *Flush* command.



(a) without waiting for completions (b) with waiting for completions

Figure 2.4: NVMe *Flush* commands apply to all completed commands at time of submission [5, §7.1]. The OS must wait for writes to complete before submitting a *Flush* command.

Manufacturer	Toshiba	Samsung	Micron	Samsung
Model	XG5	SSD 990[10]	7300 PRO	PM9A3
Type	consumer	consumer	data center	data center
Year	2017	2022	2019	2021
Has volatile write cache?	yes	yes	no	no
Block sizes [byte]	512, 4096	512	512, 4096	512, 4096
Metadata sizes [byte]	0	0	0, 8, 64	0
Max transfer size	256 KiB	256 KiB	16 KiB	2048 KiB
Atomic write size	16 KiB	512 KiB	1 block	512 KiB
Power-fail write size	1 block	1 block	1 block	1 block

Table 2.4: Comparison of NVMe parameters reported by NVMe SSDs.

NVMe Features in SSDs

We now take a look at how the NVMe features discussed here are supported by SSDs in practice. Table 2.4 shows a comparison between four NVMe SSDs. Two NVMe SSDs (Toshiba XG5 and Samsung SSD 990) are consumer models. The other two (Micron 7300 PRO and Samsung PM9A3) are intended for data center applications.

We can see that the consumer SSDs in our set come with a volatile write cache, whereas the data center SSDs do not have one.

The block size on Samsung's consumer SSDs is limited to 512 bytes. The other SSDs also support 4096 byte blocks. In our set of SSDs, only the Micron 7300 PRO supports metadata.

The maximum transfer size per command ranges between 16 KiB and 2 MiB. Samsung SSDs guarantee atomicity of writes up to 512 KiB in size with respect to parallel commands (i.e., a parallel read command will either see all new blocks, or none). However, none of the SSDs support power-fail atomicity for more than one block.

As a consequence, file systems need to support a worst-case feature set that includes a volatile write cache, 512 byte blocks and no power-fail atomicity for more than one block. We assume this device model for **Suvi** in Chapter 7.

[10] Samsung's older models SSD 970 (2018) and 980 (2020) have identical data.

Chapter 3

Motivation – Designing Data Structures for PM

In this chapter, we examine the challenges of designing data structures for PM. As a practical example, we will work with a PM ring buffer designed for use in an in-kernel storage stack. This ring buffer design originated in Christian Schwarz's master's thesis, "Low-Latency Synchronous IO for OpenZFS using Persistent Memory" [111]. It is part of ZIL-PMEM, which caches synchronous file system accesses to hide the latency of the remaining asynchronous ZFS storage stack. We describe how ZIL-PMEM builds on top of the PM ring buffer in Section 3.8.1.

The design of ZIL-PMEM's ring buffer is relevant for this thesis, as it is a simple example of multiple concepts discussed in the following chapters:

- It achieves a write bandwidth close to the raw PM write bandwidth.
- It implements concurrency control, limiting the number of parallel writers to PM.
- It supports strong consistency. Every log entry is immediately persisted.

We start with a full overview of the final ring buffer. In the following sections, we discuss design parameters important for fast and correct PM usage. By comparing these parameters in isolation, we aim to build an intuition for working with PM.

3.1 PM Ring Buffer Overview

Christian Schwarz set the following requirements for the design of the ring buffer [111, §5.1]:

- Minimal overhead
- Scalability on multicore systems
- PM bandwidth management for efficient CPU use
- Detection of media errors with checksums

The ring buffer is used as a cache for asynchronous storage transactions. The asynchronous storage stack handles all read operations through its DRAM page cache. During regular operation, the PM ring buffer is therefore write-only. The recovery process reads from the ring buffer only after a crash.

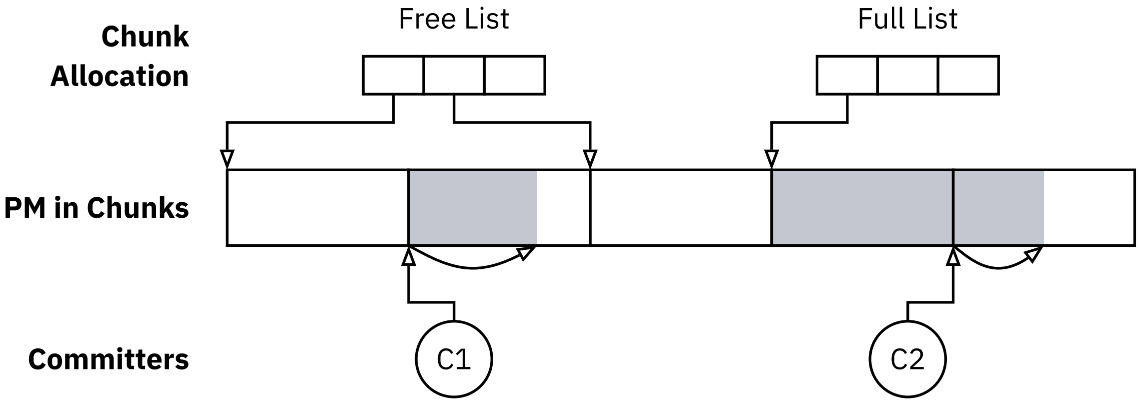


Figure 3.1: Overview of the ring buffer data structures. At runtime, each chunk is either in one of the allocation lists or assigned to a committer. Committers additionally store the offset to the free space within their chunk.

3.1.1 PM Organization and Runtime Data

Chunks

The PM ring buffer must support parallel write access. Because individual entries are small and the raw write time is short, minimizing write contention is important. The PM ring buffer achieves this by granting writers exclusive access to a large chunk of PM.

The PM space is divided into chunks of arbitrary size. There should be at least one chunk for each parallel writer, with additional chunks enabling asynchronous garbage collection. The chunk size determines the maximum entry size, though chunks should be sized to contain multiple entries to amortize the cost of garbage collection. Our implementation creates a configurable number of equally sized chunks spanning the entire PM space. ZIL-PMEM uses a fixed chunk size of 128 MiB, which fits almost 30 000 write entries with 4 KiB data [111, §8.1]. The chunk dimensions must be deterministic or stored externally, because the start and end of a chunk cannot be determined from the data on PM.

A chunk is an allocation unit. As shown in Figure 3.1, each chunk has one of three states:

Free. A chunk in the free list, ready for allocation. A free chunk always starts with an invalid entry header.

Full. A chunk filled with valid entries. It is assigned to the full list. A full chunk is returned to the free list once all its entries have been asynchronously persisted to secondary storage.

Assigned to committer. Every committer owns exactly one chunk. Because a committer has exclusive access to its chunk, it can append new entries without locking.

A mutex protects concurrent access to the chunk allocation lists.

Entry Headers

The data within a chunk is organized in variable-length entries. Every entry starts with a header containing the following fields:

Entry ID. Identifies each entry for testing purposes. Since the ring buffer can write entries to multiple chunks in parallel, some kind of identifier is necessary if the application needs to impose an order between entries.

Body length. Exact size of the body in bytes (excluding padding), used for reading the body and finding the next entry.

Body checksum. Detects media errors within the body.

Header checksum. Indicates valid entry headers. Besides media errors, the header checksum must also detect partially written entries.

ZIL-PMEM includes additional header fields (e.g., for garbage collection) [111, §5.10], which our implementation does not need. Keeping separate header and body checksums allows calculating the more expensive body checksum before holding any locks that might be necessary to fill the header.

The entry header and body are padded with zero bytes to ensure alignment at 256 bytes. We demonstrate the need for such alignment in Section 3.5.

Committers

The committers manage concurrent access to the ring buffer. Each committer owns a single chunk. It keeps track of the offset of the free space within that chunk.

A semaphore and a bitmap control the assignment of committers to writing threads. We describe this process below.

3.1.2 Recovery

The recovery algorithm runs after a crash. It needs to discover all valid entries in the chunks. Note that recovery requires no knowledge about the runtime data (i.e., free list and committers) before the crash. In particular, free chunks always start with a zero entry header.

The recovery algorithm in pseudocode is shown in Figure 3.2.

3.1.3 Write Process

The complete process of writing a new entry to the PM ring buffer is as follows:

1. Prepare the entry header. This includes calculating the body checksum and then the header checksum.
2. Obtain a committer, as described below.

```
for each chunk:
    offset := 0
    loop:
        read header from chunk.start + offset
        if header is zero or header has invalid checksum: break
        check body checksum
        yield entry to application
        offset += len(header) + len(body) rounded up to 256
```

Figure 3.2: Algorithm for the ring buffer recovery procedure in pseudocode.

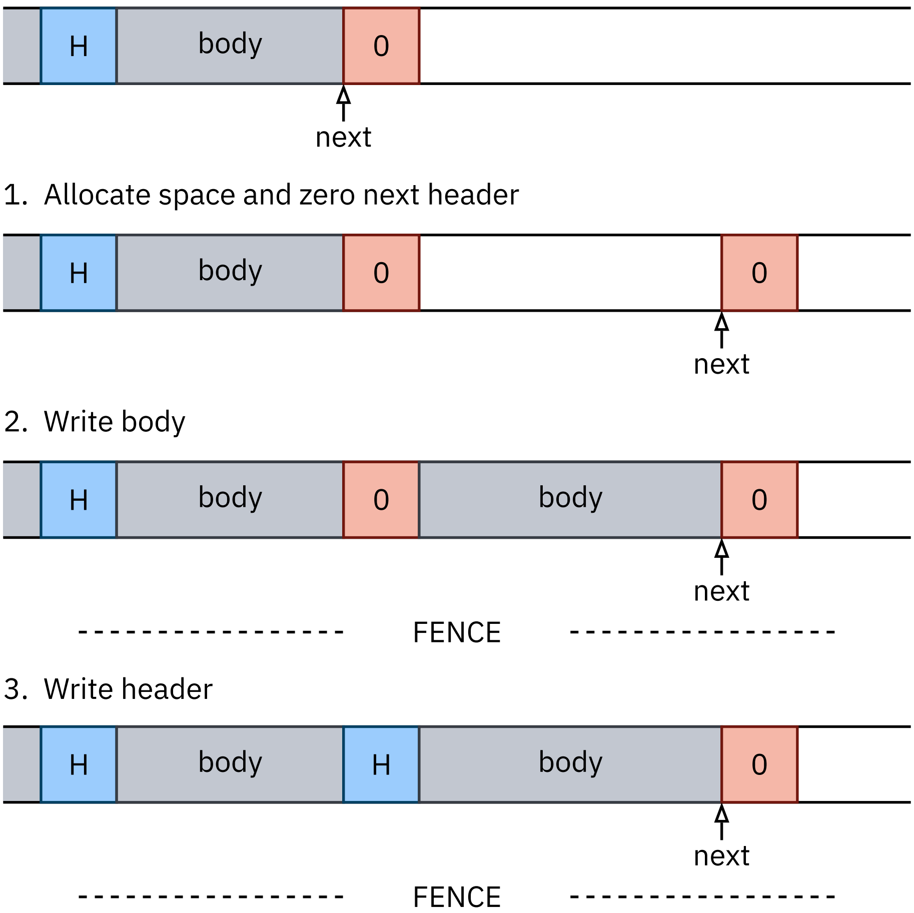


Figure 3.3: Steps for writing a ring buffer entry to PM.

3. Obtain a free chunk if the committer's current chunk does not have enough space for the new entry.
4. Write to PM as shown in Figure 3.3. All write operations use either non-temporal stores or cache flush instructions to ensure that data reaches PM. Fences ensure the ordering of operations for crash consistency, which we discuss below.
    1. Allocate space in the chunk and zero the next header (256-byte aligned).
    2. Write the entry body. Fence.
    3. Write the entry header. Fence.
5. Release the committer.

Committer Selection

The committer mechanism serves two purposes. First, it reduces lock contention in the write path. Second, it limits the number of parallel writes going to PM. Optane PM can handle between two and four committers per module (see Section 3.4).

Access to the committers is protected by a semaphore and a bitmap. The semaphore is initialized with the number of available committers. The bitmap holds one bit for each committer and is initialized to zero.

To obtain a committer, a thread first enters the semaphore. If all committers are taken, the thread will block on the semaphore. The thread then atomically loads the bitmap, toggles the first unset bit, and writes the bitmap back with a compare-andswap operation. If the write-back is successful, the thread has obtained the committer at the index of the toggled bit. Otherwise, it retries by loading the new bitmap.

To release a committer, the thread atomically clears the corresponding bit in the bitmap, then exits the semaphore.

Chunk Allocation

If the committer's current chunk does not have enough space for the new entry, it needs to obtain a new chunk from the free list and return its chunk to the full list. Since committers might access these two allocation lists concurrently, they are protected with a mutex. Contention at that mutex is unlikely as chunk allocation does not happen very often.[11]

Besides the committers, background workers might also touch the allocation lists for garbage collection. In ZIL-PMEM, once all entries in a full chunk have been written to the ZFS file system, garbage collection is triggered [111, §5.12]. Our standalone prototype triggers garbage collection if no free chunks are available.

To clear a full chunk, it is sufficient to overwrite the first entry header with zeroes. The recovery will stop at a zero header (see Section 3.1.2) and the write path does not make assumptions about previous PM contents.

Crash Consistency

The write operation must be atomic. If the system crashes during a write operation, recovery must never pass an incomplete entry to the application. The PM ring buffer achieves atomicity with two memory fences placed before and after writing the entry header (step 3 of Figure 3.3).

If the system crashes before step 3, the header of the new entry is always zero. Recovery will therefore stop at the zero header and will not attempt to read a potentially incomplete body.

If the system crashes after step 3, the final fence ensures that both the new entry header and the body have reached PM. The ring buffer can therefore guarantee to applications that the written entry will not be lost in the event of a crash.

Finally, if the system crashes during step 3, we rely on the header checksum to detect crashes. If the checksum is correct, the fence before step 3 ensures that the body is complete and that the next entry header is zero. Otherwise, recovery will reject the entry and will not process partially-written header fields.

Both fences are necessary for atomicity. If we remove the first fence, the recovery might encounter a valid header with an incomplete body.[12] Additionally, the next entry header might not be zeroed out (step 1). Since chunk garbage collection does not overwrite the complete chunk, recovery might encounter an old entry at that location and yield such an entry to the application.

[11] Christian Schwarz calculates that ZIL-PMEM's 128 MiB chunks require chunk allocation for every 29 127th 4 KiB write entry [111, §8.1].

[12] The recovery will most likely reject the body based on the body checksum. However, with large entries, collisions of basic checksums for error detection, such as CRC-32, become likely.

```
pub trait PMAccess {
    /// Copy src to dst. Pad dst with 0 bytes if it is larger than src.
    /// dst must be a multiple of 64 bytes and aligned at 64 bytes.
    fn memcpy_to_pm(dst: &mut [u8], src: &[u8]);

    /// Store fence for draining PM writes.
    fn sfence() {
        unsafe { _mm_sfence() };
    }
}
```

Listing 3.1: `PMAccess` trait for changing the instructions used for writing to PM.

If we remove the second fence, an entry might be lost in a crash until another entry is written. We could therefore no longer guarantee to applications that their data is immediately persisted.

3.2 Implementation and Evaluation Setup

We implement a standalone version of the PM ring buffer in Rust. The core ring buffer implementation has approximately 300 lines of code. The source code is available at https://github.com/lluchs/pm-ringbuf

Neither the Rust standard library nor the popular reimplementation *parking_lot* provide semaphores. We therefore use semaphores from the C standard library for the committer selection.

For the experiments below, we required different implementations of the PM access functions and needed to vary the entry header alignment. We achieve both using Rust generics. Since Rust performs *monomorphization* of generics at compile time [78], we can switch implementations without runtime overhead or missed optimizations.

All PM access is encapsulated in a Rust trait, `PMAccess`, shown in Listing 3.1. We implement this trait with non-temporal AVX-512 stores and with regular stores plus cache line flushes. The `sfence()` function enables changing the fence instruction from the default `sfence`. The entry header alignment is configured with a *const generic* parameter [35], which sets the size in bytes.

We implement the benchmark using Criterion.rs [53]. Criterion is a library for creating microbenchmarks. It invokes a user-defined benchmark function that performs the operation under test for N iterations. During a short warm-up period, Criterion automatically determines a value for N that ensures a low overhead from the benchmark setup (e.g., measuring runtime). Criterion then performs multiple measurements and calculates statistics such as the mean runtime per operation and the standard deviation.

Since we evaluate parallel workloads, our benchmark functions start $T = 1..18$ threads (based on the number of CPU cores) and waits for them to finish. Each thread writes $\frac{N}{T}$ entries to the ring buffer, then exits. We calculate throughput from the measured write latency and the constant entry size.

Table 3.1 shows the system configuration for the benchmarks in this chapter. We configure the PM attached to the first CPU 4-way interleaved (pc62). In Section 3.4.1, we compare non-interleaved PM (pc62-NI). We pin the benchmark process to the

	pc62	pc62-NI
Motherboard	Supermicro X11DPU	
CPU	2 × Intel Xeon Gold 5220 (18 × 2.2 GHz)	
SMT	Hyperthreading disabled	
DRAM	12 × 32 GB DDR4 2666 MHz	
PM	8 × 128 GB Intel Optane PM 100	
PM Region	2 × 4-way interleaved	8 × non-interleaved
PM Namespace	1 devdax namespace on region 0	
SSD	Toshiba XG5 1 TB attached to CPU 1	

Table 3.1: System configuration for the evaluation in this chapter.

first CPU so that PM accesses are local. For the NUMA experiments in Section 3.6, we run another benchmark pinned to the second CPU (remote PM).

3.3 Memory Access Instructions

PM can be accessed with any CPU instruction that reads from or writes to memory. Especially in file systems, however, most PM writes usually come from a simple memcpy routine that copies file data or metadata blocks in bulk. Our PM ring buffer writes exclusively to PM with memcpy. We therefore want to choose instructions that provide maximum performance for memcpy.

In Figure 3.4, we compare non-temporal store instructions (movnt64) with regular cached stores followed by a cache line flush instruction (clflushopt [13]). We discuss the different write paths with these two instructions in Section 2.4.1. In both cases, we use AVX-512 instructions that write 64 bytes per instruction. We show the throughput of pure memcpy to PM and of ring buffer writes.

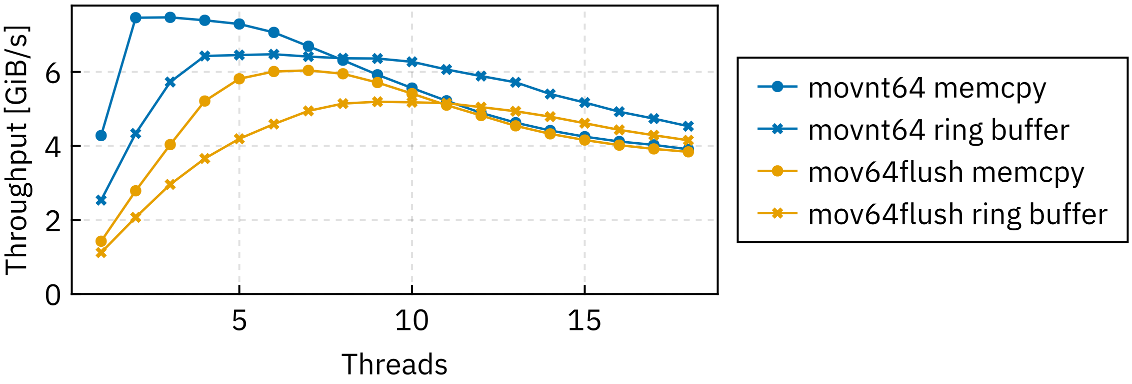


Figure 3.4: pc62 Comparison of writing to PM and the ring buffer with non-temporal store instructions (movnt64) and with regular stores followed by cache flushes (mov64flush). We compare the throughput of a simple memcpy loop and of writing entries to the ring buffer. Parallel writes to the ring buffer are not limited.

[13] x86 also includes the clwb instruction, which writes back a cache line without necessarily evicting it from the cache. On the second-generation Intel Scalable processors used for the benchmarks here, clwb has identical behavior to clflushopt.

Instruction	Media Reads	Media Writes	Read Requests	Write Requests
movnt64 1 thread	17 M	17 M	0.0020 M	17 M
	1.0 GiB	1.0 GiB	120 KiB	1.0 GiB
mov64flush 1 thread	30 M	17 M	12 M	17 M
	1.8 GiB	1.0 GiB	710 MiB	1.0 GiB
movnt64 16 threads	23 M	23 M	0.0050 M	17 M
	1.4 GiB	1.4 GiB	310 KiB	1.0 GiB
mov64flush 16 threads	32 M	21 M	9.2 M	17 M
	1.9 GiB	1.3 GiB	560 MiB	1.0 GiB

Table 3.2: pc62 Results of on-DIMM performance counters for writing 1.0 GiB of data to PM. For each counter, we list the raw value (in 64-byte blocks) and the corresponding number of bytes. We observe read requests with mov64flush and a larger number of media writes with 16 threads.

For memcpy, we observe a large difference in throughput. The implementation with non-temporal instructions writes data at 4.3 GiB/s with a single thread and shows the highest throughput of 7.5 GiB/s with two threads. With regular stores and cache flushes, the single-thread throughput starts at only 1.4 GiB/s. It then ramps up slowly, reaching a peak of 6 GiB/s at six threads.

We can understand this behavior by examining the on-DIMM performance counters. We set up a test program that writes 1.0 GiB of data to PM and records the change in the counters across all interleaved DIMMs. Table 3.2 shows the results. For the non-temporal instructions with one thread, we see 17 million write requests that result in 1.0 GiB of data read and written to the PM media. With cached stores, there are an extra 12 million read requests, causing an additional 0.80 GiB of data to be read from the PM media.

The reason for the additional reads lies in the processor's cache coherence protocol. Before a CPU core may write to a cache line in its private L1 or L2 caches, it needs to load that cache line and invalidate it in all other private caches [95]. On Intel processors, this event is called a *Read For Ownership* (RFO). If the data is not already in the cache hierarchy, it is read from memory.

With both implementations, throughput decreases with more threads after the peak. In our benchmark, we see identical throughput with more than ten threads. In Table 3.2, we also show the PM performance counters at 16 threads. For the same number of write requests, we now see 40% more media reads and writes for movnt64. This indicates that the on-DIMM caching structures can no longer coalesce the incoming parallel write requests, causing expensive read-modify-write operations [124].

The ring buffer does additional work on top of the memcpy operation, including committer selection and multiple memory fences. It also writes extra data for the entry headers, which we do not count toward the throughput here. We therefore expect to see lower throughput than for memcpy, as in Figure 3.4. This extra overhead,

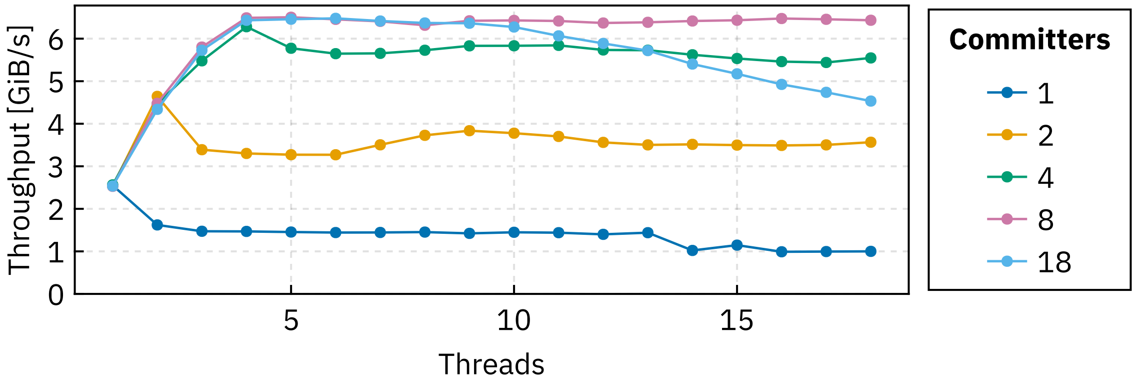


Figure 3.5: pc62 Throughput when writing to the ring buffer with different numbers of committers. The test system has 18 CPU cores, and the PM is interleaved over four modules.

however, also reduces the PM load at higher thread counts. For this reason, the throughput decreases more slowly than for memcpy.

Based on these results, we prefer non-temporal store instructions for PM access when possible. The following benchmarks all use movnt64 instructions.

3.4 Parallel Accesses

In the previous section, we saw that PM write throughput decreases with more threads. To avoid such a decrease, the ring buffer includes a committer mechanism that limits parallel writes to PM (see Section 3.1.3).

In Figure 3.5, we show the write throughput to the ring buffer with different numbers of committers. Since our test system has 18 CPU cores, parallelism to PM is not limited at 18 committers. We can see that the committer mechanism successfully keeps the throughput stable with higher numbers of threads.

However, there is a noticeable overhead once there are more threads than committers. Comparing the results for four and eight committers, we see that both reach maximum throughput at four threads. At five threads, the throughput with four committer slots drops by 11.2% and remains at that level. With eight committers, we do not observe such a drop at nine threads, since even with the extra overhead from committer contention, the available PM bandwidth remains the bottleneck.

We therefore recommend limiting parallelism to the highest number of threads that can sustain the maximum bandwidth. In Chapter 5, we discuss more mechanisms for limiting parallelism. Aside from throughput, we also introduce CPU efficiency as a metric, for which a lower limit is an advantage.

3.4.1 Non-Interleaved Optane PM

The Optane PM in the rest of this chapter is configured as interleaved over four modules. It can therefore handle a higher level of parallelism than a single module can. In Figure 3.6, we repeat the committer benchmark with non-interleaved PM.

We see that for non-interleaved Optane PM, a single thread is sufficient to reach a maximum bandwidth of 1.9 GiB/s for memcpy and 1.8 GiB/s with the ring buffer.

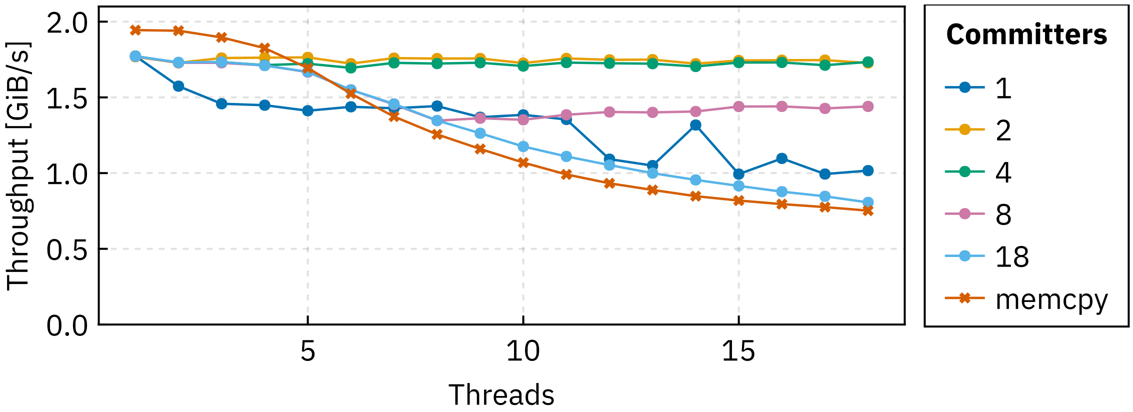


Figure 3.6: pc62-NI Throughput when writing to the ring buffer with different numbers of committers, using non-interleaved Optane PM. We include memcpy throughput for comparison.

With unrestricted parallelism (committers = 18), throughput decreases with more than four threads. Both two and four committers keep the throughput stable. With one committer, contention at the committer selection leads to unstable throughput.

Based on these results, we recommend limiting parallel access to non-interleaved PM to two to four threads.

3.5 Alignment and Access Size

Optane PM has an internal access size of 256 bytes [124]. Writes smaller than 256 bytes require a read-modify-write operation in Optane's caching structures, which increases write latency. Our PM ring buffer avoids such operations by aligning entry headers and bodies at 256 bytes. The header always has a fixed size of 256 bytes, and the body is padded so that the next entry header starts at a 256-byte boundary.

We show the need for such padding by comparing the write throughput with a variant that aligns at 64 bytes (a single cache line) instead of 256 bytes. We expect to see extra latency with 64-byte alignment since the write path has two memory fences that would wait for read-modify-write operations while writing the body (first fence) and the header (second fence). Figure 3.7 shows the results.

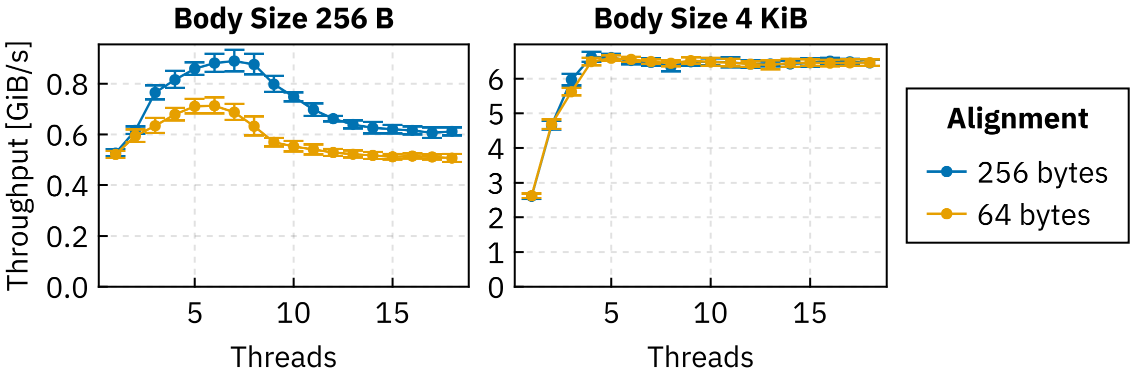


Figure 3.7: pc62 Throughput when writing to the ring buffer (8 committers) with different body sizes and header alignment. Error bars show standard deviation. With 64-byte headers, throughput is 17.4% higher on average for 256-byte bodies.

For a body size of 256 bytes, the total size of the ring buffer entry with 256-byte alignment is 512 bytes. This is 60% larger than the total size of 320 bytes with 64-byte alignment. However, we observe a 17.4% higher throughput with the larger alignment. Consequently, writing more data to PM to avoid read-modify-write operations is advantageous.

Note that the overall throughput with a smaller body decreases, since the relative overhead from the crash-consistent write protocol increases. Additionally, we can observe extra contention at the committer selection once the thread count is higher than the number of committers.

With larger body sizes such as 4 KiB in the right plot in Figure 3.7, the difference in throughput disappears. In this case, the cost of read-modify-write operations while writing the header is overshadowed by the time required to write the body. Since with a larger body the padding size is only a small fraction of the complete entry, there is little benefit in omitting the padding.

3.6 NUMA

On systems with multiple CPUs, each CPU has only a part of the system memory attached to its memory controller. The remaining memory is accessed over an interconnect between the CPUs, which generally results in increased latency. This situation is called Non-Uniform Memory Access (NUMA).

Remote PM accesses are especially expensive. We repeat the previous experiments with the benchmark threads pinned to the second CPU. In Figure 3.8, we compare throughput to local and remote PM as in Section 3.3. In general, the maximum throughput to remote PM is lower than to local PM. Non-temporal stores continue to be faster. With non-temporal stores, we observe a ramp-up to a maximum memcpy throughput of 4.4 GiB at six threads. Remote accesses with regular stores behave more similarly to local accesses, with a smaller increase up to 2.6 GiB at seven threads.

Overloading PM with parallel remote accesses is especially problematic. The throughput drops to a stable level of only 0.33 GiB with non-temporal stores or 0.70 GiB with regular stores. Limiting parallel accesses is therefore important.

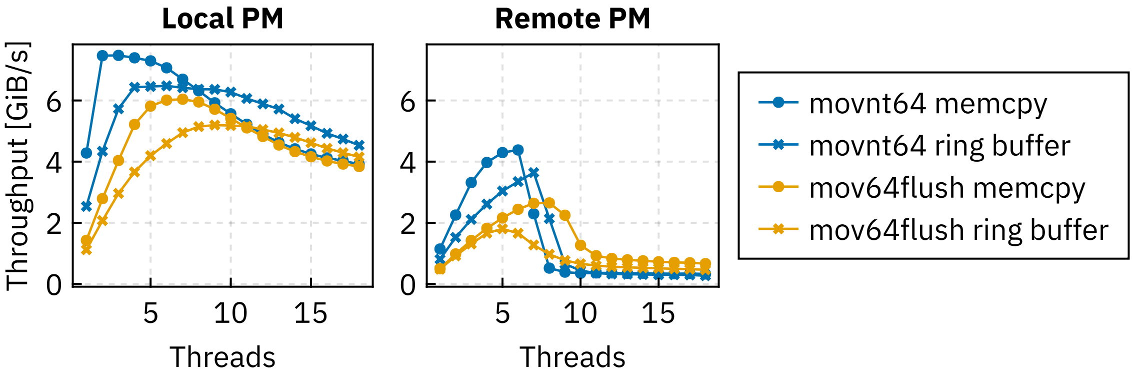


Figure 3.8: pc62 Comparison of writing to local and remote PM with different instructions (see Figure 3.4). The throughput to remote PM is generally lower than to local PM and suffers more from overload.

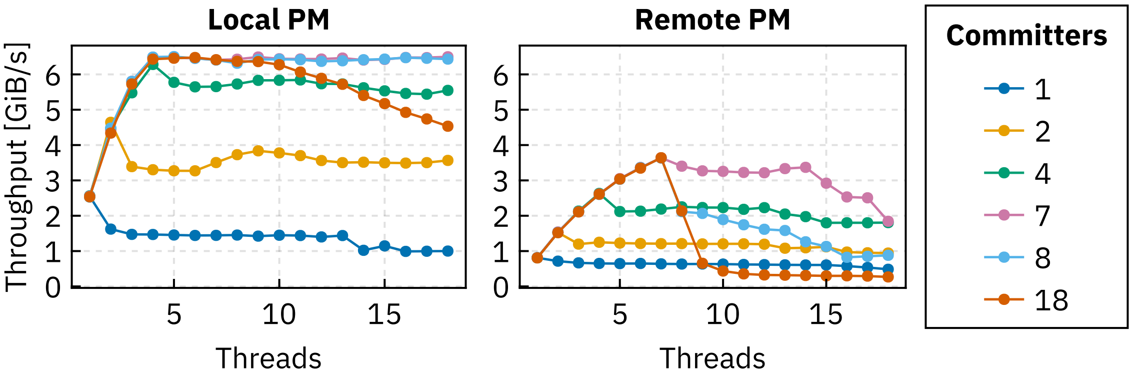


Figure 3.9: pc62 Comparison of committer counts to local and remote PM (see Figure 3.5). The maximum parallelism to remote PM is slightly lower, with the highest stable throughput at seven committers.

In Figure 3.9, we compare the committer mechanism for local and remote PM. Since the maximum throughput to remote PM is achieved with only seven threads, we include seven committers here as well. From the results, we can see that a lower committer count is beneficial for remote PM accesses. Without a limit (18 committers), the throughput after nine threads is lower than with a single committer. At seven committers, the throughput is not as stable as with local PM and decreases to the same level as four committers at 18 threads.

Note that support for NUMA systems was not among the goals of the ring buffer. The ring buffer could be extended to support NUMA by allocating chunks and committers separately for each NUMA node. Entries would then always be placed on local PM. We leave such an extension as future work.

In summary, it is best to avoid remote PM accesses, as they incur significant penalties in latency and throughput. If they cannot be avoided (for example, when an application's data is distributed across all PM regions), then aggressive limits on parallel accesses are necessary to ensure high performance.

3.7 Discussion

From the results in this chapter, we can infer the following recommendations for PM applications. These recommendations match the best practices given by Yang et al. [127].

Use non-temporal stores. Non-temporal stores avoid unnecessary reads from PM, resulting in a higher throughput per thread.

Prefer writes aligned to 256 byte blocks. Writes smaller than 256 bytes cause internal read-modify-write operations and decrease throughput.

Limit parallel PM accesses. At high levels of concurrency, overall throughput decreases as more threads access PM in parallel.

Avoid remote NUMA accesses to PM. Remote PM accesses result in high latency and low throughput compared to local accesses.

3.8 PM Ring Buffer for File Systems

The PM ring buffer described in this chapter was integrated into two storage systems as part of student theses. Christian Schwarz designed and implemented ZIL-PMEM [111], a write cache for the ZFS file system. Ilia Bozhinov integrated the ring buffer into a block device write cache via the Linux Device Mapper framework [32].

In this section, we briefly examine these two works to demonstrate the integration of the ring buffer into larger storage systems. We compare the throughput and efficiency of both systems in Chapter 4.

3.8.1 ZIL-PMEM: PM Write Cache for ZFS

ZFS [31] is a file system known for its advanced features, including volume management, data integrity checks with checksums, compression, encryption, snapshots, and replication. It is a file system for block devices with no built-in support for PM.

ZFS collects modifications in transactions. Multiple transactions form a transaction group that is asynchronously persisted to storage. To avoid high completion latency for synchronous file system writes (e.g., files opened with `O_SYNC` or calls to `fsync()`), ZFS implements a logical log called the *ZFS Intent Log* (ZIL). A synchronous file system operation can return immediately once all relevant changes are recorded in the ZIL and does not need to wait until the transaction group is persisted.

ZIL-PMEM [111] is a ZIL implementation specifically for PM. The PM ring buffer described in this chapter provides the physical storage format for ZIL-PMEM. Due to the committer mechanism, entries in the ring buffer are randomly distributed across multiple chunks at runtime. Consequently, ZIL-PMEM requires additional header fields in each entry to reconstruct the ZIL's logical order during recovery. The additional header fields include the transaction group, a generation number, and a generation-scoped ID. The generation numbers encode dependencies among log entries. In combination with the generation-scoped ID, they determine the replay order of the log after a crash.

For garbage collection, ZIL-PMEM tracks the most recent transaction group of each full chunk. Once a transaction group is persisted, all full chunks that do not have any items belonging to future transaction groups are cleared and returned to the free list.

In his evaluation, Christian Schwarz demonstrates up to 8× speedups of ZIL-PMEM compared to upstream ZFS [111]. He shows that ZIL-PMEM works especially well for small synchronous operations. He also verifies that ZIL-PMEM avoids excessive CPU stalls due to overloaded PM, which we expand in this thesis in Chapter 4.

3.8.2 DPWC: Write Cache for Block Devices

The Device Mapper framework in the Linux kernel [74] allows extending the block layer between file systems and storage devices with new functionality. Linux includes Device Mapper modules that provide, among other functions, software RAID, integrity checks, and encryption.

Similar to ZFS's ZIL, dm-writecache is a module that stores a separate log for modifications on a faster storage medium to speed up `fsync()` requests. Dm-writecache is a module that caches modified blocks on a faster storage medium (SSD or PM) with the goal of improving the performance of `fsync()` requests. Unlike ZFS's ZIL, dm-writecache does not implement a log of modifications. Instead, it stores modified blocks in a red-black tree by their block address. The red-black tree allows dm-writecache to retrieve modified blocks at runtime to service read requests. Updates to the red-black tree are protected with a lock, which prevents parallel writes and limits the maximum throughput.

DPWC [32] is a Device Mapper module that caches modified blocks in the PM ring buffer described in this chapter. Unlike dm-writecache, it stores a log of modified blocks and can handle parallel stores. In contrast to ZIL-PMEM's logical log, dpwc implements a physical log, as Device Mapper modules operate at the block layer.

Since the PM ring buffer is not intended to read entries at runtime (only during recovery), dpwc cannot easily handle read requests for modified blocks. Instead, it assumes that any recently modified blocks are in the page cache and blocks read requests until the corresponding write has completed.

For garbage collection, dpwc does not have information on higher-level transactions, such as ZFS's transaction groups. Instead, dpwc physically organizes the ring buffer chunks into multiple generations. A generation has a header with a generation ID and multiple chunks. Only one generation is active at a time. If a committer cannot allocate a new chunk from the current generation, the generation is closed and all committers acquire new chunks from the next generation. Once all blocks from a closed generation are written back, its chunks are cleared and the generation can be reused.

Similar to ZIL-PMEM, dpwc uses a generation-scoped ID in the entry headers to order entries for replay. After a crash, it finds all non-empty generations, orders them by generation ID, then processes their entries. Since write requests at the block layer are idempotent, dpwc does not need to consider the state of the underlying storage for replay.

In his thesis, Ilia Bozhinov shows that dpwc can reach up to 2× speedup over dm-writecache [32]. However, he also finds that dpwc's coarse-grained garbage collection generations can lead to extended periods of low throughput while all generations are closed.

Chapter 4

PM File System Efficiency

Our analysis in Chapter 3 shows that Optane PM is very sensitive to parallel accesses. When writing to PM in parallel, the overall throughput quickly peaks and then decreases with more writers.

PM file systems face the same challenge indirectly. File systems themselves generally have little parallel activity (e.g., background write-back, defragmentation, garbage collection). However, they handle parallel requests from userspace applications, which results in a parallel PM load. We argue that ensuring efficient access to the underlying storage is the responsibility of the file system rather than the userspace application.

Besides the reduced bandwidth, there is also a hidden cost associated with parallel PM access. With more parallel PM accesses, the CPU is active for longer—more than we would expect from other storage devices such as NVMe SSDs.

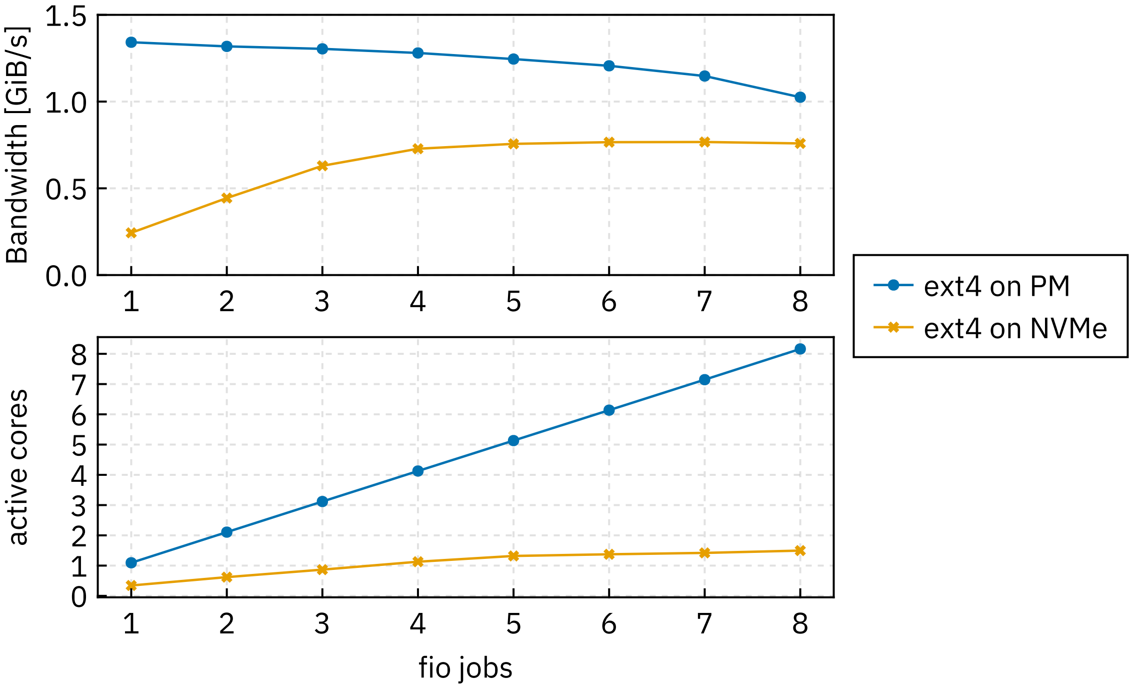


Figure 4.1: pc61 Comparison of throughput and CPU load with parallel writes to an ext4 file system on top of PM and NVMe storage.

Figure 4.1 shows two identical benchmarks writing to ext4 on top of PM and NVMe. We plot the resulting bandwidth and the number of active CPU cores as reported by the Linux scheduler (see Section 4.2.1). The bandwidth behaves similarly, with a small increase at two jobs, after which it stagnates (NVMe) or decreases (PM). However, there is a significant difference in CPU activity. With NVMe, CPU usage rises slowly, with fewer than 1.5 active cores at eight jobs. Writing to PM, on the other hand, generally requires one active core per job. Since these active cores do not achieve higher bandwidth, the additional CPU work is effectively wasted. The CPU cores spend more and more cycles stalling on PM accesses.

The reason for the difference in CPU activity is the different access modes. NVMe is an asynchronous protocol.[14] The CPU is active while the kernel handles NVMe commands and completions. While the SSD transfers the data using DMA, the CPU can sleep or run other tasks. In contrast, the CPU itself performs transfers to and from PM and is active during the entire transfer.

This should be considered unexpected behavior for users of a file system. The file system serves as an abstraction layer between applications and the underlying storage. It provides a common interface so that unmodified applications can operate with different storage media. An application cannot reasonably optimize for a storage requirement such as limited parallelism, since it cannot know the activity of other processes in the system. A PM file system should therefore ensure that it accesses the underlying storage as efficiently as possible.

Efficient use of the CPU has not been a design goal of any of the PM file systems described in Section 2.3. We argue that this is partly because there is no established metric for the efficiency of a file system. Absolute numbers for CPU activity or power consumption do not compare well. As an example, in Figure 4.1, we can see that at 1-2 jobs, ext4 on PM uses roughly double the number of CPU cores as ext4 on NVMe. However, ext4 on PM also achieves a significantly higher bandwidth than on NVMe.

In this chapter, we propose measuring CPU time and energy per transferred GiB as metrics for file system efficiency. We then compare this metric for several file systems on PM and NVMe. We show that the CPU power consumption dominates the energy efficiency for all file systems. The easy-to-measure CPU efficiency metric is therefore a useful stand-in if full power measurents are not feasible. In the following chapter, we then propose mechanisms to improve the efficiency of PM file systems.

We published parts of the work in this and the following chapter in “Analyzing and Improving CPU and Energy Efficiency of PM File Systems” at DIMES’23 [120].

4.1 Metrics for File System Efficiency

We want to establish metrics that allow comparison of the efficiency of file systems. We cover two separate goals, energy efficiency and CPU efficiency. Although this thesis focuses on PM file systems, we ensure that the metric is compatible with all types of underlying storage systems.

[14] We describe the NVMe protocol in more detail in Section 2.4.2.

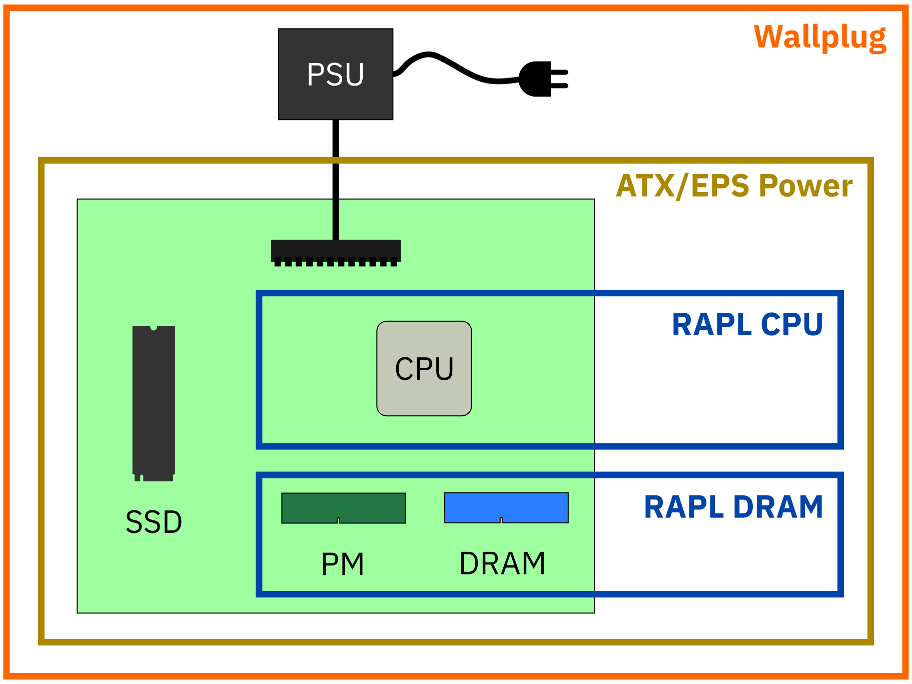


Figure 4.2: Overview of commonly-available energy measurement domains for ATX systems. Wallplug and ATX/EPS measurements require extra hardware, whereas the RAPL domains are provided by CPU performance counters.

4.1.1 Energy Efficiency

Energy efficiency covers the energy consumption of all devices in the system. For power-constrained devices, efficient use of the available energy is important. It improves the battery life of devices such as smartphones and laptops. For server systems, which are the main target for Optane PM, better energy efficiency allows reduced cooling and lowers the electricity bill.

However, measuring and comparing energy consumption comes with a set of challenges. First, it is highly dependent on all hardware components. Comparisons across different hardware configurations are usually not possible. Second, even measurements on the same hardware may not be reproducible. Environmental factors such as the ambient temperature can have an effect on the power consumption as fans need to spin faster.

Finally, most energy measurements require extra specialized hardware. Figure 4.2 shows commonly-available energy measurement domains. Without extra hardware, modern Intel server platforms provide separate energy counters for the CPU and memory DIMMs [57]. These counters do not include peripheral hardware such as SSDs. They are therefore only useful for measurements of pure PM file systems.

The next larger domain are measurements at the power connectors between PSU and the motherboard. There is off-the-shelf measurement hardware that can measure at standard ATX power connectors, providing separate measurements per connector and voltage [94]. At this level, the CPU, memory, and all peripherals are included. However, it is generally not possible to attribute measurements to specific devices since power distribution happens within the motherboard.

Measuring at the motherboard connectors is preferable to wallplug measurements, the largest domain. Wallplug measurements include the power supply unit (PSU) and are therefore highly dependent on the PSU efficiency. We compare data from the measurement domains from Figure 4.2 in Section 4.2.4.

The use of external measurement hardware also leads to a synchronization problem. Since the measurement hardware runs independently from the system under test, it is generally not possible to identify precisely where in the measurement data a benchmark starts and ends.

Metric

With the challenges above in mind, we propose measuring an *energy cost* for a given file system benchmark with the methodology below. This metric is exclusively intended for comparisons of different file systems on a single system and therefore can offer some flexibility in the measurement method. The benchmark can excercise an arbitrary file system operation that reads or writes from the storage medium, but is required to put a constant load on the system.

1. Choose a measurement domain that includes all devices used by the file system.
2. Measure the average power consumption during idle P_{idle}.
3. Measure the average power consumption during benchmark $P_{\text{benchmark}}$. Record the amount of data read or written M and the runtime of the benchmark t.

From the measured data, calculate the *energy over idle* E as follows:

$$E = (P_{\text{benchmark}} - P_{\text{idle}}) \cdot t$$

We calculate E via the average power values rather than direct energy measurements to avoid synchronization problems. Since we assume a constant load from the benchmark, we can cut away the beginning and the end of the measurement values before calculating the average power.

With the energy over idle E and the amount of transferred data M, we propose calculating the *energy cost* as follows:

$$\text{Energy cost} = \frac{E}{M} \qquad \text{Unit: } \frac{\text{J}}{\text{GiB}}$$

The energy cost describes how much energy the storage stack, including the benchmark program, file system, and storage hardware, requires for transferring a given amount of data.

The energy cost can also be calculated from the average bandwidth $r = \frac{M}{t}$:

$$\text{Energy cost} = \frac{E}{M} = \frac{(P_{\text{benchmark}} - P_{\text{idle}}) \cdot t}{r \cdot t} = \frac{P_{\text{benchmark}} - P_{\text{idle}}}{r}$$

4.1.2 CPU Efficiency

CPU efficiency is closely related to energy efficiency, as the CPU is often the largest power consumer during a file system access (see our analysis in Section 4.2.4). Efficient use of the CPU therefore typically also implies efficient energy use. Measuring CPU utilization does not present the challenges associated with measuring energy discussed above. This makes CPU efficiency an attractive alternative when energy measurements are not feasible.

In addition to its connection to energy efficiency, CPU efficiency is important in its own right. A file system that uses less CPU time leaves more CPU time available for other applications.

We propose using CPU time as the basis for measuring CPU efficiency. CPU time is a metric maintained by the operating system scheduler. The total CPU time is the sum of the number of seconds each CPU core is active during a given time period. For example, if an eight-core CPU is fully utilized for ten seconds, we would calculate a CPU time of 80 seconds.

Using CPU time makes the metric more dependent on the speed of the underlying storage device, rather than the CPU. This enhances the stability of the metric in the presence of dynamic frequency scaling and improves comparability across different systems. In contrast, using CPU cycles as a basis would always report worse CPU efficiency on a higher-frequency CPU for PM, since a memory stall of the same wall clock duration would account for more CPU cycles.

Metric

We propose the following method for measuring the *CPU cost* of a file system. An arbitrary benchmark reading or writing to the file system is the basis for our measurements. Unlike for our energy efficiency metric, the benchmark is not required to put a constant load on the system, since there is no synchronization problem with reading CPU utilization. For our analysis below, we only use benchmarks with constant load so that we can measure both metrics simultaneously.

1. Measure CPU time in an idle system over a timespan close to the benchmark runtime. Verify that it is low.
2. Measure the total CPU time T of the full system while the benchmark is running. Record the amount of data M that the benchmark reads or writes.

We then calculate the *CPU cost* as follows:

$$\text{CPU cost} = \frac{T}{M} \qquad\qquad \text{Unit: } \frac{\mathrm{s}}{\mathrm{GiB}}$$

Note that we do not filter the CPU time measurements by process. File systems often have worker threads that do not run in the context of a user process. Instead, we require that the test system has no background activity during the benchmark.

4.2 Analyzing File System Efficiency

4.2.1 Measurement Setup

Table 4.1 shows the system configuration. Our evaluation system *pc61* has two CPU sockets, each equipped with an eight-core CPU, 64 GiB of DRAM, and 128 GiB of Intel Optane PM.

For our benchmarks, we configure a non-interleaved region with the PM attached to the first CPU. We create two namespaces of types fsdax and devdax in this region

	pc61
Motherboard	Supermicro X11DPi-NT
PSU	Supermicro PWS-502-PQ (80 PLUS Bronze)
CPU	2 × Intel Xeon Silver 4215 (8 × 2.5 GHz)
SMT	Hyperthreading disabled
DRAM	8 × 16 GiB DDR4 2666 MHz
PM	2 × 128 GiB Intel Optane PM 100
PM Region	2 × non-interleaved
PM Namespaces	1 fsdax and 1 devdax namespace on region 0
SSD	3 × Micron 7300 PRO 1 TB attached to CPU 0

Table 4.1: System configuration for the evaluation in this chapter.

(see Section 2.1). The in-kernel PM file systems use the fsdax namespace and the userspace benchmarks use the devdax namespace.

We plug the three benchmark SSDs into a PCIe 3.0 x16 slot configured with x4x4x4x4 bifurcation via an adapter. These SSDs are exclusively used for benchmarks. The system is installed on a separate M.2 SSD. The PCIe slot with the SSDs is also connected to the first CPU. For all benchmarks, the first CPU is therefore the local NUMA node and the second CPU the remote NUMA node.

CPU Load Measurements

We obtain data on CPU load from the Linux kernel by reading `/proc/stat` [16]. This file contains counters measuring how much time each CPU spends in certain states.

For our analysis in this chapter, we consider the CPU active in the following states:

user Time spent in user mode. With our benchmark setup, this is almost exclusively time spent by the fio benchmark.

sys Time spent in kernel mode. This includes most time spent in the file system.

irq Time spent in an interrupt handler. This counter is relevant for file systems on NVMe SSDs, which signal completion using interrupts.

softirq Time spent in software interrupt handlers.

The remaining counters are irrelevant for our setup. Our benchmark does not create low-priority tasks, so *nice* is always zero. *steal* and *guest* count time in virtualization environments. Finally, *iowait* counts time a task waits for I/O complete and does not indicate CPU activity.

Power Measurements

We measure the power consumption of our system in two ways: at the wallplug and at the motherboard connectors.

For the wallplug measurements, we used a Rittal *PDU managed* [103]. We connect to the PDU over HTTP to receive measurement data. The device gives us roughly one measurement per second.

We attached a Powenetics V2 [94] measurement device between the PSU connectors and the motherboard. Powenetics measures at a much higher resolution than the

PDU, giving us about 1000 readings per second. Powenetics communicates over USB, presenting a serial port. Since reading data from Powenetics causes a noticeable CPU load of about 5%, we provide the measurements over TCP from a separate computer with a custom driver[15]. The driver downsamples the readings by taking the average of voltage and current, yielding a measurement every 100 ms.

Powenetics provides separate measurements for the three voltage levels of the 24-pin ATX connector and the two eight-pin EPS 12 V connectors [56] in our test system. It measures voltage U and current I, from which we calculate power as $P = U \cdot I$.

Performance Counter Measurements

Our test system has Intel Cascade Lake processors. As part of the Running Average Power Limit (RAPL) system, these processors provide two energy counters: one for the CPU package and one for the memory DIMMs [57]. Intel's documentation refers to the memory counters as DRAM counters, but they include energy used by Optane DIMMs.

The processor provides access to the energy counters via model-specific registers (MSR). For our benchmarks, we access the counters with the sysfs interface provided by the Linux powercap framework at `/sys/class/powercap` [75]. We read the energy counters every 100 ms (same as Powenetics) and calculate power as $P = \frac{\Delta E}{\Delta t}$.

On modern Intel processors, the energy counters are based on actual measurements rather than estimates [37]. Alt et al. have evaluated their accuracy with Optane and DRAM DIMMs using riser cards [24]. They found that the energy counters consistently report a higher power consumption than measured at the DIMMs. They measured an offset of 20% for idle DRAM that decreases to 10% under load. For Optane, they measured an absolute offset of around 2-3 W for one DIMM. These measurements were performed on a newer system (Ice Lake-SP with Optane 200). We still expect a similar accuracy on our system.

4.2.2 File System Selection

For our evaluation, we choose file systems in two categories based on whether they implement some form of PM bandwidth control (see Section 2.3). We limit our our analysis to kernel file systems.

We select the following file systems without bandwidth control:

ext4 as an upstream Linux file system that supports both PM and traditional block devices. We evaluate ext4 both on PM and on an NVMe SSD.

NOVA as a file system specifically designed for PM. [125]

We also include direct userspace access to PM in this category as a low-overhead baseline (**devdax**).

In the second category with bandwidth control, we select the following file systems:

ZIL-PMEM and DPWC showing the efficiency of our PM ring buffer design (Chapter 3). We configure DPWC as a write cache for one NVMe SSD with an

[15] https://github.com/KIT-OSGroup/powenetics-v2

Option	**Value**
`ioengine`	*sync* or *dev-dax*
`rw`	*randwrite*
`sync`	1
`size`	104857600 (100 MiB)
`blocksize`	16384 (16 KiB)
`time_based`	1
`runtime`	30 s
`ramp_time`	2 s
`numjobs`	1 to 8
`numa_cpu_nodes`	0 (local) or 1 (remote)

Table 4.2: The fio configuration for our benchmarks.

ext4 file system on top. For ZIL-PMEM, we provide three NVMe SSDs as backing storage.

OdinFS with its delegation mechanism for PM writes. [134]

We do not include Trio [133] since it is a userspace file system that is challenging to run (see Section 2.3.6). We expect Trio to perform similarly to OdinFS since it inherits the delegation mechanism from OdinFS.

4.2.3 FIO Benchmark Setup

We use fio [27] version 3.40 as the benchmark program. Table 4.2 shows an overview over the configuration. With these options, fio is set up to do 16 KiB writes to random locations of a 100 MiB file per job. The files are opened with `O_SYNC`, meaning that all writes are immediately persisted to storage [17]. For file system benchmarks, we use the *sync* ioengine, which uses normal *write(2)* system calls. The devdax baseline (see above) uses the *dev-dax* ioengine, which maps PM directly in userspace.

The benchmark is time-based and runs for 30 seconds after a two-second warmup time. The warmup time ensures that we measure a steady load during the 30-second benchmark time, without effects from initial page faults and cache misses. We set up all external measurements (Section 4.2.1) after the warmup time. In the plots, we show the mean bandwidth with standard deviation as reported by fio.

For every storage stack (i.e., a file system on PM or NVMe), we repeat the benchmark for 1 to 8 parallel jobs pinned to either the local or the remote NUMA node. We therefore end up with 16 runs per storage stack.

For the *idle* baseline, we do not run fio and just perform the external measurements over 30 seconds.

Discussion

Our metric is agnostic regarding the underlying file system benchmark. We chose these parameters for the analysis in this thesis because we are primarily interested in overhead from parallel PM write accesses. In particular, we chose a relatively

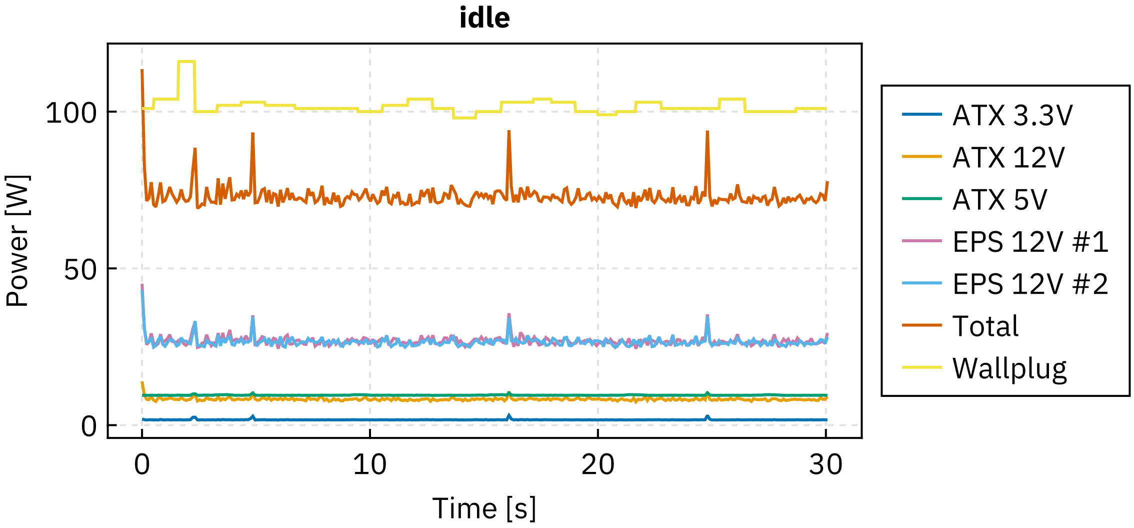


Figure 4.3: pc61 Power measurements of the idle system from the PSU-to-motherboard connectors (24-pin ATX and two EPS 12 V) and the wallplug.

large access size of 128 KiB to reduce overhead from system calls and file system processing other than the storage writes.

The parameters for file size and runtime are arbitrary and do not have a direct effect on the benchmark results. As fio performs synchronous writes, neither the OS page cache nor CPU caches are involved, even for small file sizes. The runtime needs to be large enough for the external measurements (Section 4.2.1). Other than that, we expect to measure a steady state that does not change over time.

4.2.4 Power Measurements

Our metric for energy efficiency requires choosing an appropriate energy measurement method (see Section 4.1.1). In this section, we analyze the available power domains (wallplug, Powenetics, RAPL) for our evaluation platform. We choose power measurements from Powenetics for the energy efficiency metric in this thesis.

Wallplug and Powenetics

Figure 4.3 shows a power measurement of the idle system over 30 seconds. Powenetics provides separate measurements for the three voltage levels of the 24-pin ATX connector and the two eight-pin EPS 12 V connectors [56]. It measures voltage U and current I, from which we calculate power as $P = U \cdot I$. We then sum all power values to obtain the total power.

The power measurements start just before the benchmark and end after the benchmark finishes. In some benchmarks, the beginning of the measurements shows influence from the benchmark preparations and the end captures a momentarily idle system. To calculate an average power value, we therefore remove the first and last seconds of measurements. On our test system, we measure an average idle power of 100 W at the wallplug and 73 W at the motherboard connectors.

The power difference between wallplug and PSU output is due losses within the power supply. According to the data sheet, the power supply in our test system has an average efficiency of 84% [115]. We measure a slightly lower efficiency of 72%

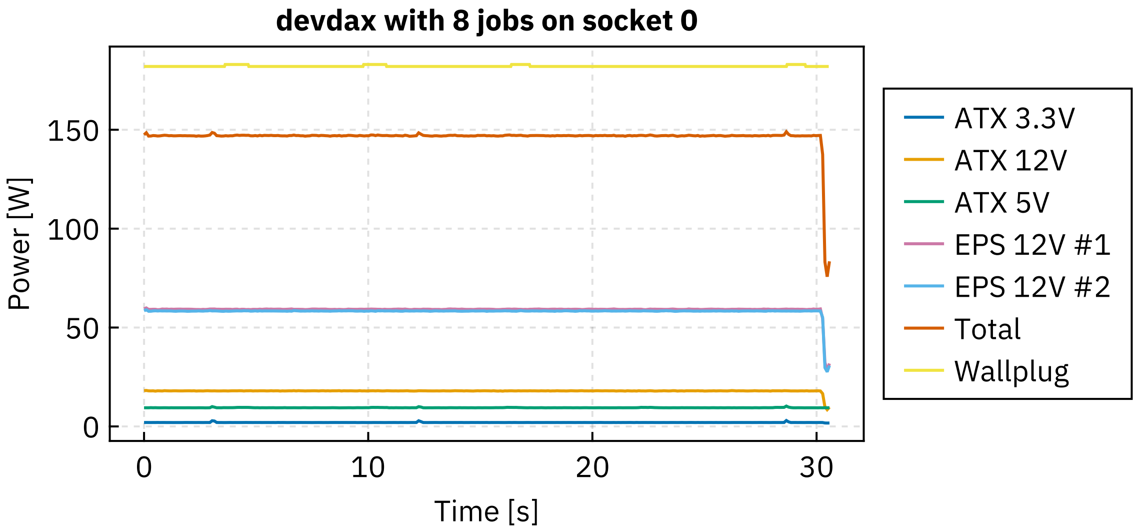


Figure 4.4: pc61 Power measurements of the system under load. The benchmark is writing to PM in userspace with eight parallel jobs on the first CPU.

for the idle system and 81% under single-CPU load (see below). Power supplies with higher ratings reach better efficiency. For example, with an 80 PLUS Gold rating, a PSU must have an efficiency better than 90% at 20% load [34]. We therefore prefer measurements from Powenetics to remove influence from PSU efficiency on the results.

Our measurement of the idle system shows frequent changes in the 12 V connections. Even on the idle system, there is a small amount of background load from the benchmark runner recording sensors. This load causes the CPU to change its power states, which we can observe at the 12 V connectors supplying the CPU.

With a constant load on the system during a benchmark, the power measurements are more stable. We show an example of power measurements during a benchmark in Figure 4.4. Under this particular load, we measure an average wallplug power of 180 W and 150 W at the motherboard connectors.

To obtain the power consumption of the benchmark, we subtract the average idle power from the average power during the benchmark. For the example here, we therefore calculate 150 W - 73 W = 74 W.

Note that even though there are two EPS 12 V connectors supplying the two CPUs, we cannot measure each CPU individually. Even with only one CPU under load, Powenetics reads identical current on both EPS connectors. This indicates that the lines from the 12 V connectors are connected on the motherboard before the voltage regulators of the two CPUs. We therefore have to rely on performance counters for information about fine-grained power distribution.

Performance Counters

In Figure 4.5, we show the energy counter measurements in comparison with the Powenetics and wallplug measurements discussed above. At idle, the energy counters indicate a total power consumption of 39 W, which is 53% of the power measured with Powenetics. Under single-socket load, this measurement rises to 110 W, or 74% of Powenetics. The absolute offset stays at around 38 W, which is

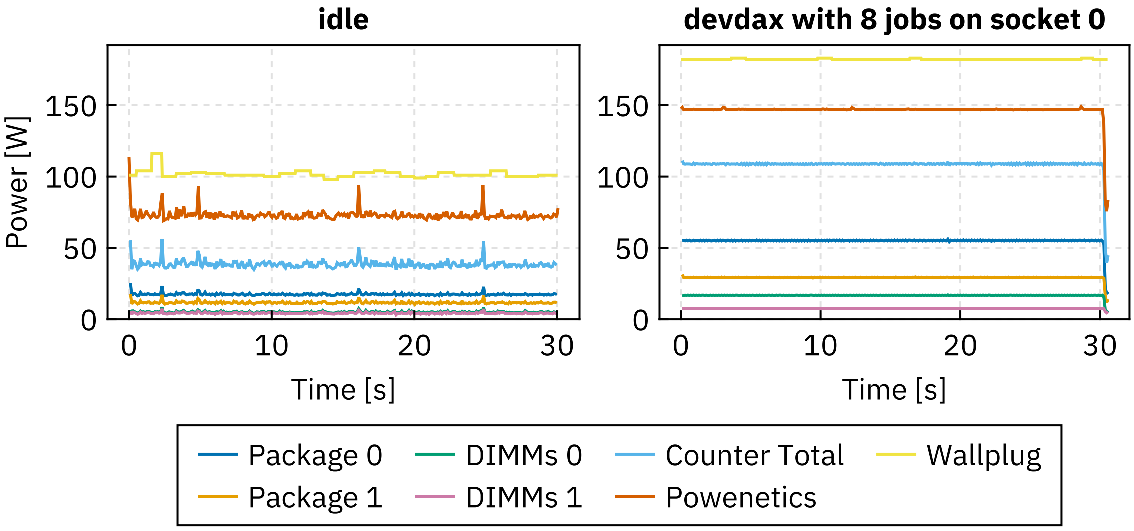


Figure 4.5: pc61 Power measurements with the energy counters, in comparison with external measurements from Powenetics (total) and at the wallplug.

likely power used by components not directly involved in the benchmark (e.g., SSDs and NICs).

Looking at the power increase from idle to the benchmark in Figure 4.5, we observe +70 W with the energy counters and +74 W with Powenetics. The devdax benchmark (writing to PM from userspace) therefore almost exclusively uses extra power for CPU and memory.

As a comparison, we show power used by ext4 on PM and on NVMe in Figure 4.6. We can see that the overall power consumption with NVMe is lower. The Powenetics measurements show regular short spikes, likely due to regular file system or SSD activity.[16] In Table 4.3, we show the increase of power over the idle system. We can

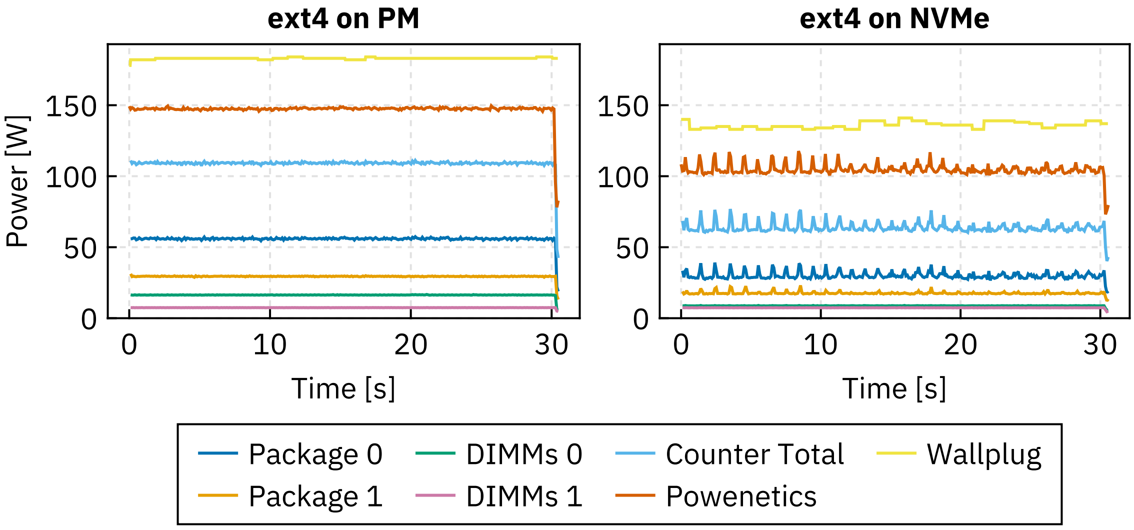


Figure 4.6: pc61 Comparison of power used by ext4 on PM and on NVMe. The benchmark is writing with eight jobs.

[16] The energy counter measurements are not frequent enough to capture the spikes, so we cannot use them to attribute the activity to either SSD or CPU. We do not measure more frequently to avoid extra CPU activity.

Measurement	PM		NVMe	
Package 0	38 W	52%	13 W	39%
Package 1	18 W	24%	6.0 W	19%
DIMMs 0	11 W	15%	3.7 W	12%
DIMMs 1	3.1 W	4.2%	3.1 W	9.6%
Total Counters	71 W	95%	25 W	79%
Powenetics	75 W	100%	32 W	100%

Table 4.3: pc61 Increase of power over the idle system measured by energy counters and Powenetics with ext4 writing to PM and to NVMe with eight jobs. The percentages are relative to the increase measured with Powenetics.

see that ext4 on PM has a very similar power consumption as the userspace devdax benchmark shown above. The increase of the power measured with the energy counters is 95% of the increase measured with Powenetics. The majority of this increase comes from the CPU packages (75%).

With ext4 on NVMe, we expect to see extra power consumption by the NVMe SSD, which is not measured by the energy counters. Indeed, we see that the counter total increase is only 79% of Powenetics. Even with NVMe, the power consumption of the CPU packages make up the majority of the increase at 58%. This underlines the importance of measuring and optimizing CPU time when file system efficiency is a goal.

4.3 Evaluation

We evaluate our metrics on a selection of kernel PM file systems (see Section 2.3). We analyze file systems with and without management of parallel accesses separately.

4.3.1 ext4 and NOVA

In Figure 4.7 (page 68), we show results for ext4 on PM and NOVA as examples of file systems that do not limit parallel accesses to PM. Other file systems without such limits, such as PMFS and WineFS, have very similar behavior and results. We therefore do not show data from these file systems. As comparison, we include *ext4 on NVMe* for a traditional block-based storage system, and *devdax* for userspace PM access.

Local Access

We first discuss access from the local NUMA node (left column). It is immediately apparent that the lower overhead of userspace access translates to higher bandwidth and better efficiency: devdax bandwidth is on average 8.3% higher than NOVA bandwidth, with 8.6% lower energy cost and 6.1% lower CPU cost.

Comparing NOVA and ext4, we see a slightly higher bandwidth for NOVA (0.18% on average). Although the absolute power consumption for NOVA in consistently higher than for ext4, the energy cost is equal. This highlights the value of our metrics: NOVA’s lower overhead and higher bandwidth leads to a higher power consumption, but does not imply a lower efficiency than ext4.

The the bandwidth decreases as more jobs access the file systems. The CPU cost therefore increases proportionally faster than the number of jobs: For a 8× increase in jobs from one to eight, the CPU cost of NOVA increases 9.1×. Similarly, while the power consumption of NOVA from one to eight jobs increases by 13%, the energy cost increases by 58%.

Remote Access

For accesses from the remote NUMA node (right column in Figure 4.7), we observe extreme drops in bandwidth with more than three jobs, as discussed before in Section 3.4. With userspace access (devdax), the bandwidth at lower job counts starts higher compared to the file systems, but then decreases faster with higher job counts. At 6 jobs, we see a devdax throughput of only 77 MiB/s. The lower overhead of DAX access therefore also leads to a more extreme overload situation compared to access from the file system.

The power graph for the remote NUMA node shows a notable difference to local accesses: Between four and six jobs, the overall power consumption rises more slowly. This indicates that the extra CPU cores the additional jobs are running on consume less power than for local accesses. Since the additional jobs cause major CPU stalls across all cores, it is likely that the CPU can save power by disabling execution units. As the bandwidth stabilizes with more than 6 jobs, the overall power consumption rises again as more cores are activated.

Due to the extreme drop in bandwidth, both energy and CPU cost rise quickly. It is therefore important to avoid this situation for both performance and efficiency reasons. In Chapter 5, we introduce approaches to this end designed for addition on top of existing PM file systems like NOVA.

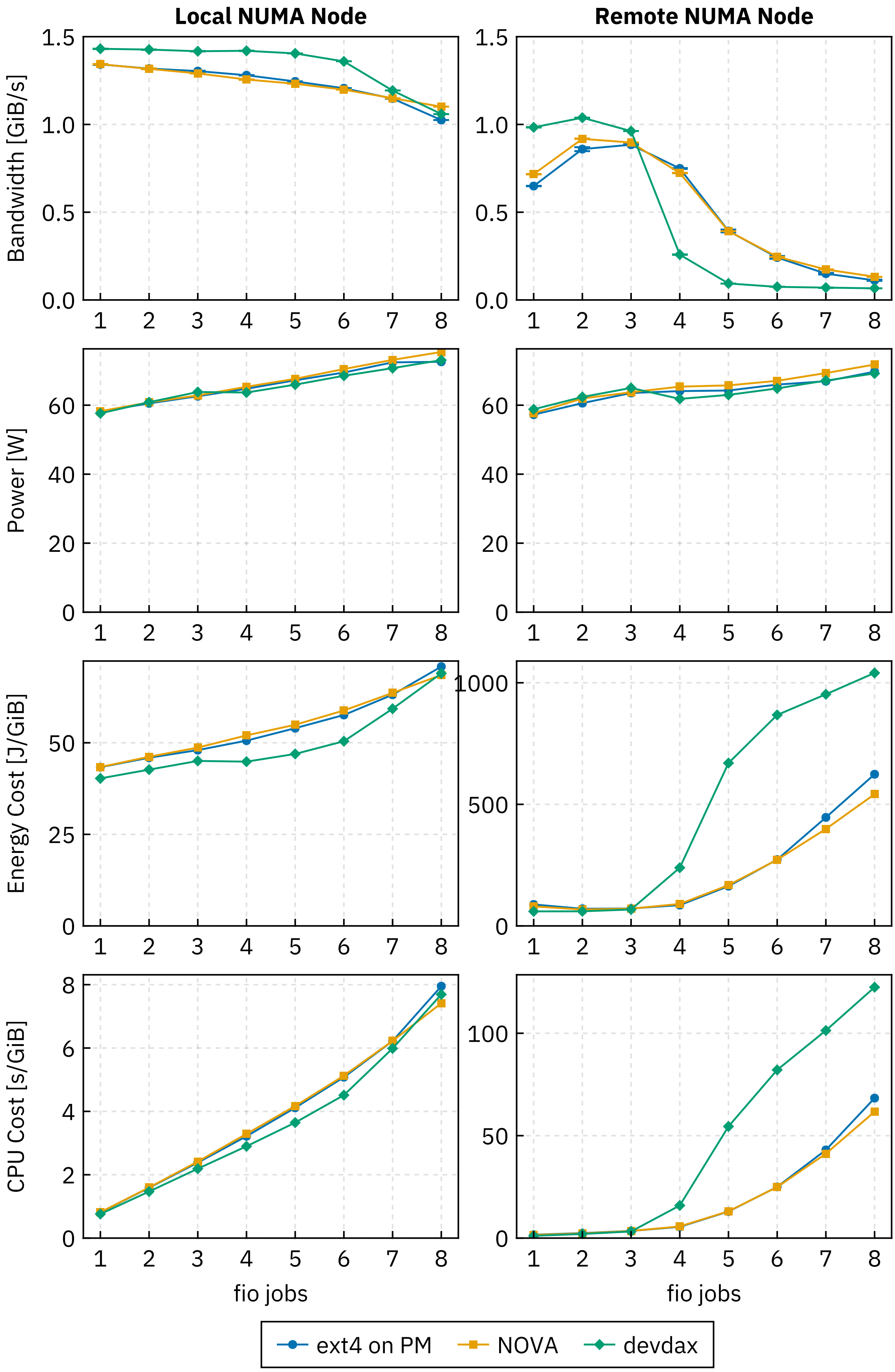


Figure 4.7: pc61 Comparison of efficiency of 16 KiB synchronous writes. This figure shows PM file systems that do not limit parallel accesses. As comparison, *devdax* shows efficiency of direct access from userspace.

4.3.2 OdinFS, ZIL-PMEM, and DPWC

In Figure 4.8 (page 71), we show results for file systems that limit parallel accesses to PM: OdinFS, ZIL-PMEM, and DPWC. We set an identical limit of three parallel writers for these systems.

OdinFS [134] has a delegation mechanism for PM accesses (see Section 2.3.1). By delegating to a limited number of threads, parallel accesses are limited. Additionally, the delegation mechanism avoids remote NUMA accesses by pinning the delegation threads to PM-local CPUs. We set up OdinFS without striping across NUMA nodes, since we want to evaluate only its ability to limit parallel PM accesses.

ZIL-PMEM and DPWC build on the PM ring buffer we describe in Chapter 3. We describe these file systems in more detail in Section 3.8. They limit parallel accesses with a committer mechanism. Every write access to the ring buffer requires acquiring a committer, which are limited (see Section 3.1.3). In contrast to OdinFS, the committers only function as a lock and do not mitigate expensive remote NUMA accesses. Both ZIL-PMEM and DPWC are cross-media storage systems. We set up ZFS with ZIL-PMEM backed by three NVMe SSDs.[17] DPWC is a device mapper module that works in the Linux block layer. We set it up with one NVMe SSD as origin device and with an ext4 file system on top.

As comparison for the cross-media file systems, we also include data from ext4 on one NVMe SSD.

Local Access

Again, we start by analyzing access from the local NUMA node (left column in Figure 4.8). The different bandwidth levels are explained by the different backing storage devices. DPWC is backed by PM and one NVMe SSD. At high levels of parallelism, its bandwidth is close to ext4 directly on NVMe. At fewer jobs, it can sustain this bandwidth since it provides the NVMe SSD with asynchronous requests at a higher queue depth than is possible for plain ext4. ZIL-PMEM reaches a bandwidth closer to the maximum PM bandwidth since the combined bandwidth of its three NVMe SSDs exceeds the PM bandwidth. Finally, OdinFS reaches the highest bandwidth as a PM-only file system.

Comparing ZIL-PMEM and OdinFS between three and seven jobs, we can see that ZIL-PMEM's bandwidth is close (4.6% lower on average). However, it requires almost a quarter less CPU time per GiB. This translates to a 4.2% lower overall power consumption, even though ZIL-PMEM also keeps three NVMe SSDs active. We discuss OdinFS's high CPU cost in comparison with our mitigations in Section 5.3.

Comparing the two PM ring buffer implementations ZIL-PMEM and DPWC, we can see higher CPU and energy costs from DPWC at lower bandwidth. The primary difference is where the PM ring buffer is integrated into the storage system. For DPWC, a write request goes through the full processing in ext4 before reaching DPWC as individual block I/O requests. ZIL-PMEM, in constrast, short-circuits ZFS's regular processing for synchronous accesses like in our benchmark. The processing for the NVMe write can then happen in the background, and gains efficiency by batching.

[17] This setup matches the setup in Christian Schwarz's thesis [111].

Note that we see high CPU and energy costs for all file systems at low job counts compared to NOVA and ext4 on PM (Figure 4.7). For ZIL-PMEM and DPWC, this is likely due to extra work for the NVMe access. However, even OdinFS requires 40 % more energy per GiB than NOVA at one job. This shows that countermeasures against expensive parallel accesses can have a significant cost and motivates a need for monitoring efficiency.

Looking at ext4 on NVMe, we note that while its CPU cost is relatively stable between 1.4 s and 1.9 s, it is 69% higher than ext4 on PM at one job. As long as PM is not overloadad, the CPU time required to manage NVMe commands and completions for a certain amount of data therefore exceeds the CPU time required to copy the data to PM. The overall power consumption of ext4 on NVMe shows a counter-intuitive behavior: As more jobs write to the file system, the power consumption decreases. A possible explanation for this behavior is that with a larger volume of NVMe commands, the completions can be batched and therefore processed more efficiently. Since this thesis focusses on PM file systems, we do not analyze this anomaly in more detail. The two cross-media file systems that also access NVMe SSDs do not exhibit any drops in power consumption for local PM accesses.

Remote Access

For file system access from the remote NUMA node (right column in Figure 4.8), we can see that OdinFS's delegation works as intended: The bandwidth with two or more jobs is identical to local accesses.

The committer mechanism in the PM ring buffer successfully keeps the bandwidth stable, but at a lower level compared to local accesses. We can see that ZIL-PMEM suffers more from remote accesses than DPWC, confirming that the NVMe bandwidth available to DPWC is the limiting factor for local accesses.

The CPU and energy costs for remote accesses are very similar to those with local accesses, with some minor shifts especially for ZIL-PMEM due to lower bandwidth. For DPWC, a drop in overall power appears with more than five jobs, which likely mirrors the behavior seen with ext4 on NVMe.

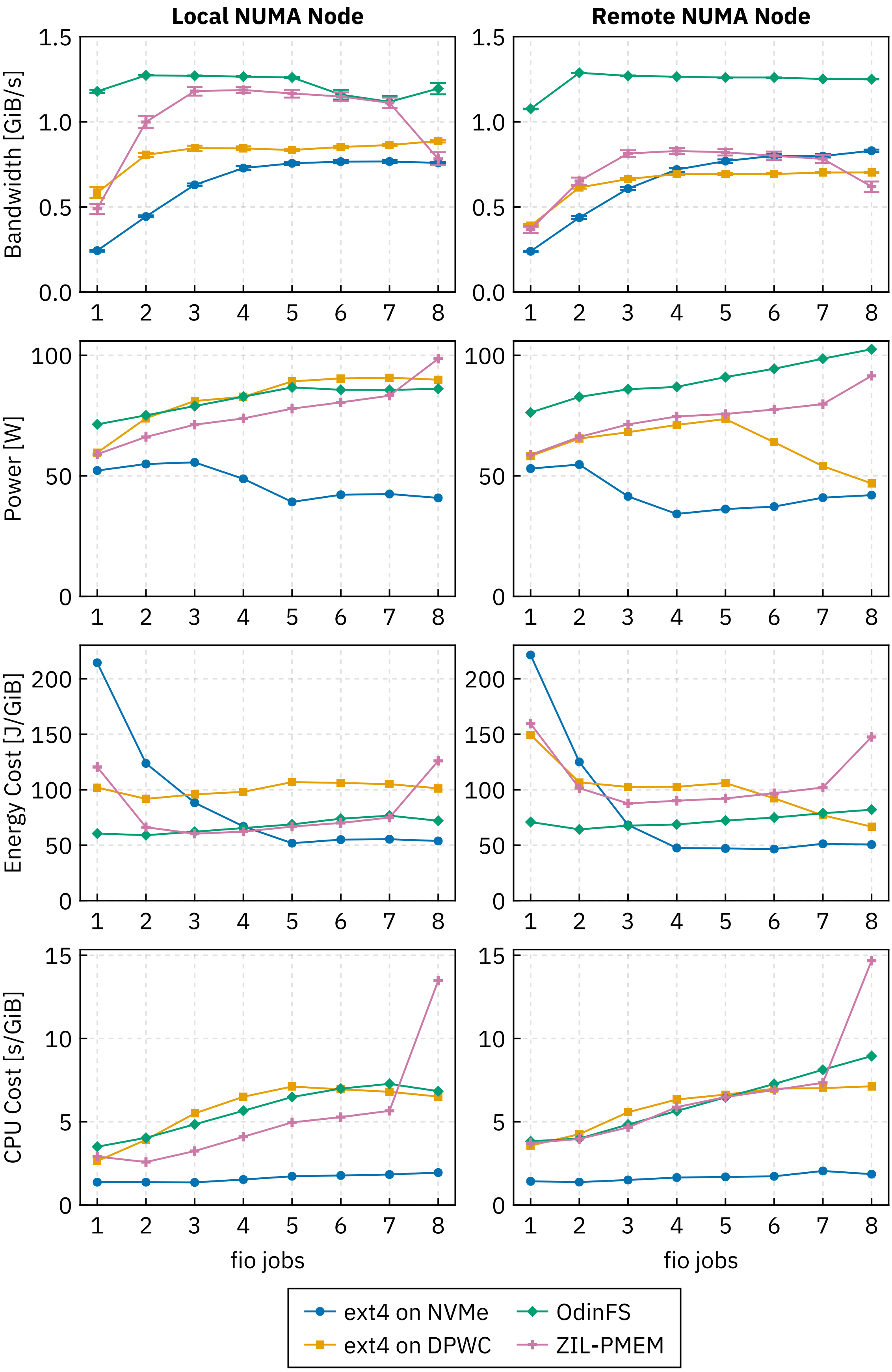


Figure 4.8: pc61 Comparison of efficiency of 16 KiB synchronous writes. This figure shows PM file systems with mechanisms that limit parallel accesses, and ext4 on NVMe as comparison.

4.4 Discussion

We introduced metrics for energy and CPU efficiency and methodologies for measuring them. Our analysis of existing PM file systems has shown that efficiency is usually not a primary concern. Even for file systems that limit parallel accesses, the CPU and energy efficiency results were often counterintuitive. We expect that our metrics will provide a basis for future energy-efficient file system design.

We highlight the importance of bandwidth-independent metrics. The file systems in our test set exhibited a wide range of write bandwidths between 0.5 and 1.2 GiB/s. At higher write bandwidths, the rate of file system calls, and therefore CPU utilization, always increases, which makes direct comparison based on CPU utilization or power draw difficult. For this reason, we designed our metrics based on the amount of written data. In this way, we always analyze the same number of file system calls, independent of bandwidth.

A major challenge for improving the energy efficiency of file systems is the limited set of tools available for power measurement. We observed counterintuitive power measurements for some benchmarks, where total power consumption decreases at higher load. However, neither built-in performance counters nor external probes can provide fine-grained measurements that would allow attribution of such power savings to specific hardware components. Future server platforms should support efficiency improvement efforts by including more power-measurement domains.

Chapter 5

PM File System Overload Mitigation

In the previous chapter, we defined metrics for energy and CPU efficiency of file systems. Our analysis demonstrated that PM file systems commonly do not limit parallel accesses to PM. When under a parallel load, these file systems overload the PM DIMMs, which leads to excessive CPU stalls and declining total bandwidth. These PM file systems therefore waste energy and CPU resources.

In this chapter, we propose three mechanisms to mitigate such behavior. Rather than designing a PM data structure from the ground around efficient PM access like our PM ring buffer (Chapter 3), we aim for mechanisms that are simple to add to existing PM file systems. The key insight is that PM file systems generally use a simple memcpy routine for copying file data to PM. This routine contributes most PM write bandwidth.

We integrate our mechanisms into the NOVA file system [125] and compare their performance and efficiency with OdinFS [134] and ZIL-PMEM [111], both of which already limit PM parallelism. We show that effective parallelism limits are possible without a major redesign of the file system write path.

5.1 Design and Implementation

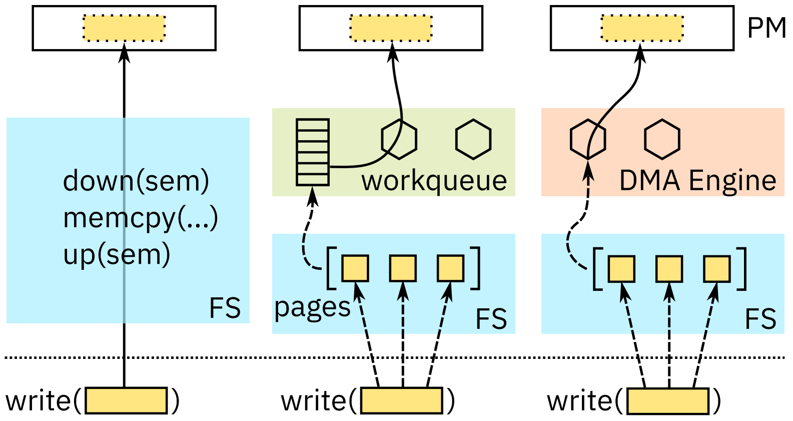
We design our PM overload mitigation as a replacement for a memcpy routine. To facilitate integration with existing file systems, its API is a single function with a signature matching the Linux kernel memcpy functions:

```
int ep_write_pmem(void *dst, const void *src, size_t len)
```

It copies `len` bytes from `src` to `dst` using non-temporal instructions. It returns 0 on success or an error code otherwise. All three mechanisms share this function, with a kernel module option selecting one at runtime.

The central idea for all mechanisms is to make synchronous access asynchronous. Rather than wasting active CPU time with stalls, calls to `ep_write_pmem` should block in an overload situation and allow the scheduler to run other tasks.

Note that it is not required for the file systemto perform all PM accesses with our memcpy function. To be effective, our function only needs to handle the bandwidth-



(a) Semaphore (b) Workqueue (c) DMA

Figure 5.1: Mechanisms for mitigating PM overload in kernel file systems by replacing memcpy.

intensive copying of user data. The file system may still perform its metadata updates directly, as these generally constitute only a small fraction of the bandwidth to PM.

Figure 5.1 shows an overview of the three mechanisms—Semaphore, Worker, and DMA—which we describe in the following sections.

5.1.1 Semaphore

A semaphore is a synchronization primitive that limits the number of tasks accessing a resource simultaneously. By protecting the memcpy function with a semaphore, only a limited number of tasks can access PM simultaneously.

Semaphores are a very simple mechanism for preventing PM overload. Listing 5.1 shows the full implementation of our semaphore-based PM overload mitigation for the Linux kernel.

In order to improve CPU utilization, the semaphore must be blocking: If a task cannot enter the semaphore, it must release the CPU to other tasks. The semaphores in the Linux kernel[18] fulfill this property. They are implemented with a spinlock protecting a count and a wait list. Entering the semaphore with `down()` decreases the count, and leaving the semaphore with `up()` increments the count. If a task cannot enter the semaphore (i.e., the count is zero), it adds itself to the wait list and sleeps. A task leaving the semaphore wakes the oldest task in the wait list.

```
static int ep_write_pmem_memcpy_sem(void *dst, const void *src, size_t len)
{
  down(&memcpy_sem); /* enter the semaphore */
  __copy_from_user_inatomic_nocache(dst, src, len); /* memcpy */
  up(&memcpy_sem);   /* exit the semaphore */
  return 0;
}
```

Listing 5.1: Using a semaphore in the Linux kernel to protect memcpy.

[18] Implemented in `kernel/locking/semaphore.c`

5.1.2 Workqueue

A workqueue is a mechanism for performing concurrent work. It is organized around a synchronized queue of *work items*, which are arbitrary structures defining a unit of work. Any task may enqueue a work item into the queue. Worker threads dequeue items from the workqueue, perform the work, and mark the item as finished.

We propose using a workqueue to perform the memcpy to PM operations asynchronously. By limiting the number of workers, we limit the maximum amount of parallelism to PM. A workqueue can also mitigate expensive remote NUMA accesses to PM by pinning the worker threads to the same node as the PM.

We implement this mechanism based on Linux's workqueue API [54]. The work items capture the three parameters to memcpy (destination and source pointers, and length). The destination pointer always refers to PM mapped in the kernel address space and can be passed directly to the workers. In contrast, the source pointer can refer to the user address space, which is not mapped in the worker threads. Therefore, it is first necessary to obtain kernel mappings to any user pages with `get_user_pages()` [36] and pass those mappings to the worker. The workers then copy the data from each mapping to PM, since the kernel mappings are not necessarily contiguous. Source pointers refering to the kernel address space are passed and handled directly.

A call to `ep_write_pmem()` creates and enqueues a work item, then blocks until the work is completed. The workqueue improves CPU utilization, since other tasks may execute while a memcpy operation is waiting in the queue.

Since creating mappings and work items introduces overhead for each copy operation, we handle copy operations smaller than a 4 KiB page synchronously without the workqueue.

5.1.3 DMA

The workqueue improves CPU efficiency by performing memcpy operations asynchronously with a limited number of workers. However, the workers each still occupy a CPU core for just a memcpy loop. We can further reduce the CPU utilization by offloading the workers to dedicated hardware.

Such hardware is available in modern Intel processors as Intel I/OAT [62]. The I/OAT hardware is connected over PCIe and can perform operations like memcpy or memset with DMA. The Linux kernel provides the *DMA Engine* API for working with I/OAT and similar hardware [76].

Before enqueueing a DMA memcpy operation, we need to make the source and destination memory ranges available to the DMA hardware. As with the workqueue, source pages in userspace must be resolved and pinned first. If the source address is not from userspace (i.e., the file system is copying internal data), we fall back to a direct memcpy because the kernel does not allow DMA to most of its address space. The source and destination pages are then mapped into the DMA hardware's address space.

The I/OAT hardware provides multiple channels that can execute operations in parallel. We limit parallelism to PM by using a limited number of channels.

After mapping all pages, `ep_write_pmem()` obtains a channel (protected with a semaphore) and enqueues memcpy operations for each source and destination page. It then waits for the DMA operations to complete before unmapping the pages and returning.

The DMA mechanism improves CPU efficiency by blocking both when no DMA channel is available, and during the actual memcpy operation. However, there is extra overhead for setting up the DMA transfer. We therefore only use DMA for copies that have a size of at least one page (4 KiB).

5.2 File System Integration

To integrate our mechanisms into a file system, we need to identify and replace the function that copies file data from userspace to PM. As all three approaches share a common API, we only need to perform this replacement once. We describe integration into NOVA, PMFS, and WineFS, which are file systems that do not limit parallel PM accesses.

NOVA and WineFS use a local helper function to copy data to PM. In NOVA, this function is called `memcpy_to_pmem_nocache()`[19], and in PMFS, it is called `memcpy_to_nvmm()`[20]. In both file systems, these helpers call the kernel memcpy function `__copy_from_user_inatomic_nocache()`, whose signature and usage match our `ep_write_pmem()` function. This makes the integration a single-line change in the file system.

PMFS does not provide such a local helper function. Instead, it calls the aforementioned memcpy function directly in multiple places. We therefore need to replace all of these calls.

Since these three file systems exhibit very similar performance in our benchmark, we evaluate only NOVA in the following section.

5.3 Evaluation

We integrate our approaches into the NOVA file system as described above and compare them with OdinFS and ext4 on NVMe. In Figure 5.2 (page 78), we evaluate scalability under parallel access as in Chapter 4.

Since some mitigations have a high cost per access, we additionally compare different access sizes with a fixed number of jobs in Figure 5.3 (page 79). For this benchmark, we set the workqueue to always copy via a worker, even when the access size is smaller than 4 KiB, to show the real overhead of this mechanism. We include devdax to compare userspace access without per-access overhead.

5.3.1 Semaphore

The semaphore is a good solution for local PM access. Throughout our scalability benchmark in Figure 5.2, NOVA/Semaphore shows the highest bandwidth (tied with OdinFS) at the lowest energy and CPU cost.

[19] located in `fs/nova/nova.h`

[20] located in `fs/winefs/xip.c`

However, the semaphore cannot mitigate expensive remote NUMA accesses. On average, we observe a 32% lower bandwidth from the remote NUMA node compared to local access. However, it can still keep a steady bandwidth with increasingly parallel file system accesses.

At small access sizes below 1 KiB (Figure 5.3), we observe a lower bandwidth with the semaphore than with unmodified NOVA. For these access sizes, the threads enter the semaphore more frequently, creating additional overhead from contention.

5.3.2 Workqueue

The workqueue variant is similar in approach to the delegation mechanism in OdinFS. However, there are significant differences in performance and efficiency. For the 16 KiB access size shown in Figure 5.2, NOVA/Workqueue shows equal performance to OdinFS for three and four local jobs. With fewer or more jobs, the bandwidth of NOVA/Workqueue drops. However, NOVA/Workqueue consistently requires less power than OdinFS, making it more efficient when the bandwidth is sufficiently high. It shows a smaller or equal energy cost up to five jobs and a smaller CPU cost up to six jobs.

The reason for this disparity lies in the notification mechanisms used by Linux workqueues and OdinFS. OdinFS uses busy waiting to wait for the delegated PM write to complete. This strategy ensures low latency but wastes CPU time and power. In contrast, Linux workqueues block the calling thread while waiting for a work item, which releases the CPU for other tasks. This strategy reduces CPU time and saves power but increases latency, which is why the bandwidth below three jobs suffers.

The drop in bandwidth with five or more jobs is explained by contention at workqueue submission. At larger access sizes and high job counts (not shown in figures), we observe reduced contention and improved bandwidth.

In the access size comparison in Figure 5.3, NOVA/Workqueue suffers most from access sizes under 4 KiB. For every access, it must resolve at least one 4 KiB page, which significantly increases the overhead for accesses smaller than one page. Therefore, such accesses are performed directly, without indirection via the worker.

For remote NUMA jobs, NOVA/Workqueue reaches higher bandwidth than NOVA/Semaphore, since it performs the PM accesses locally. However, it still suffers from higher latency and contention, as discussed above. Compared to OdinFS, both CPU and energy costs are consistently either better or equal, since OdinFS's busy waiting appears costlier at more than five remote jobs.

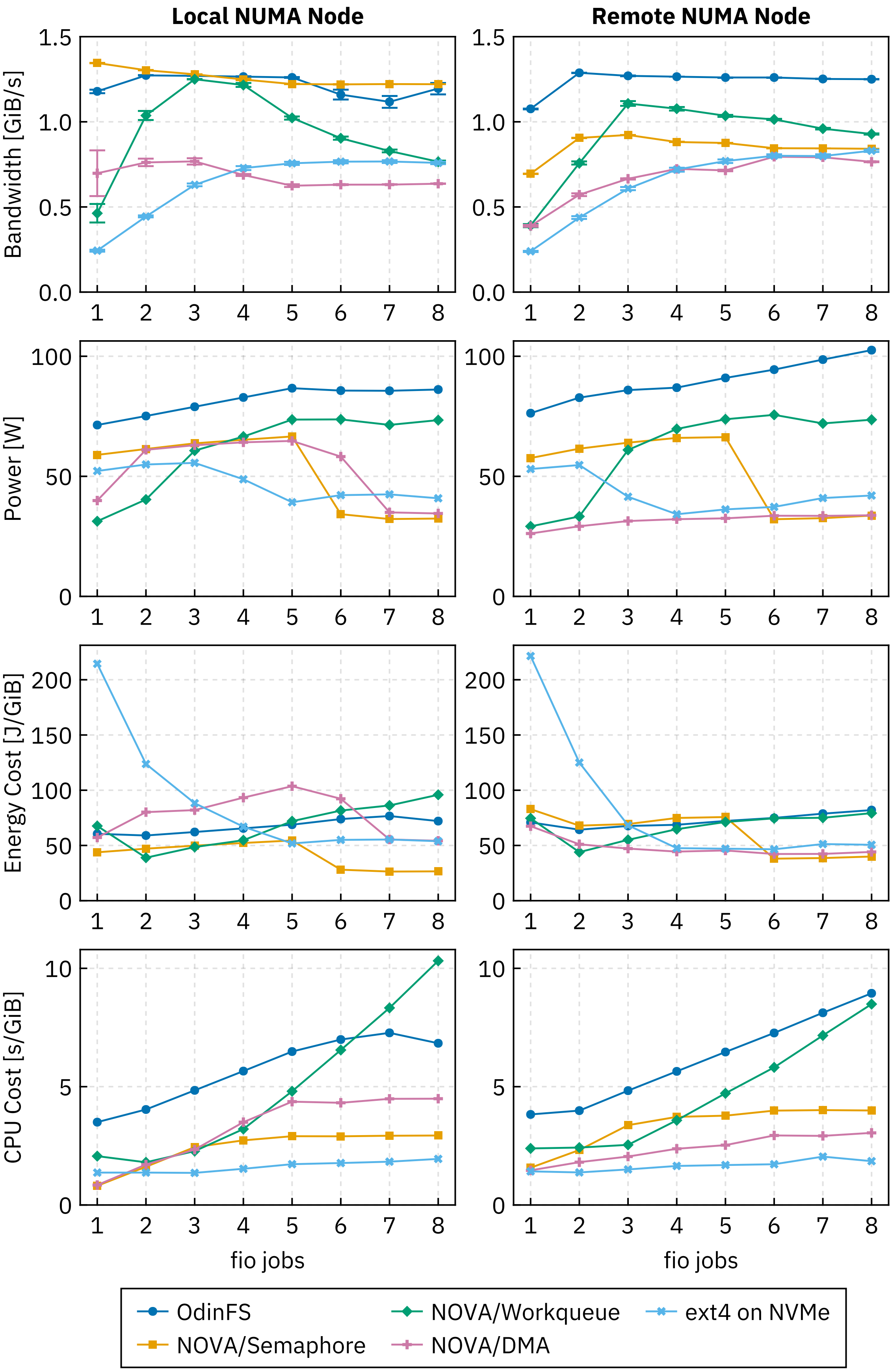


Figure 5.2: pc61 Multicore scalability comparison of our strategies to manage parallel PM accesses, with fio synchronous 16 KiB writes (see Section 4.2.3).

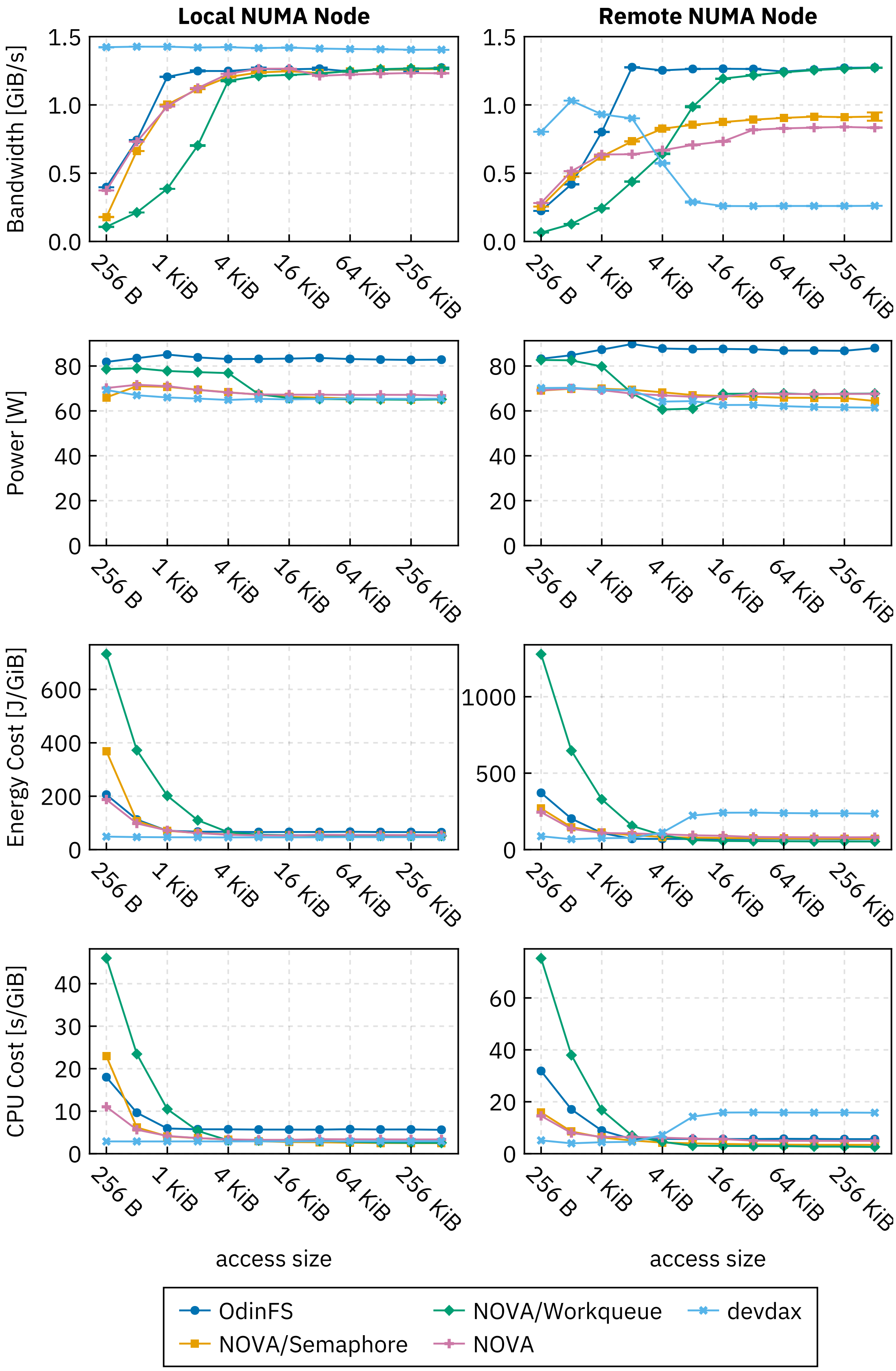


Figure 5.3: pc61 Access size comparison for software-based strategies to manage parallel PM accesses. We configure four fio jobs and power-of-two access sizes.

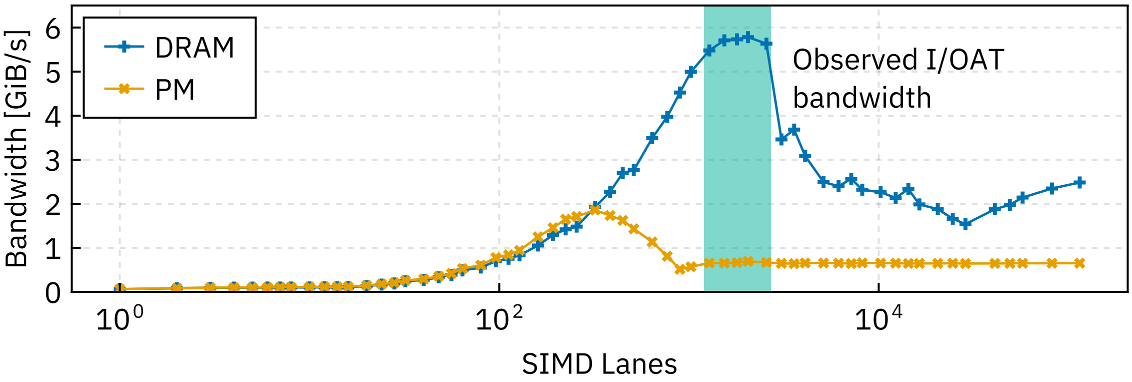


Figure 5.4: Write bandwidth to DRAM and PM from a GPU at increasing levels of parallelism. The observed bandwidth with I/OAT matches the GPU bandwidth.

5.3.3 DMA

DMA offloading with I/OAT cannot reach the expected performance in our benchmarks (Figure 5.2, page 78). The bandwidth of NOVA/DMA peaks at 0.77 GiB/s, around half of the bandwidth of a regular NOVA in this benchmark. We confirm that this behavior is not a problem with our implementation by repeating the same benchmark on DRAM instead of PM. With this setup, we observe a bandwidth exceeding 5 GiB/s, which confirms that our implementation is not the bottleneck.

To better understand DMA transfers to PM, we create a benchmark that writes to PM and DRAM from a GPU. By varying the number of SIMD lanes available to the GPU program, we can control the amount of parallelism of the write accesses. Figure 5.4 shows the results. At the number of SIMD lanes required for maximum bandwidth to DRAM, PM is already overloaded. Maximum bandwidth to PM would therefore require reducing parallel transfers.

These observations from GPU transfers provide a likely explanation for the I/OAT behavior. We assume that the I/OAT hardware is tuned for DRAM transfers and would require adjustment for better performance to PM. The low bandwidth is therefore not a general problem with our approach.

5.4 Discussion

The main goal of the design and implementation of the mitigation mechanisms here is to demonstrate that making a PM file system CPU and energy efficient PM is always viable, even if efficiency was not a requirement of the original design.

Our prototype therefore comes with a number of limitations. There is only a global limit to parallelism (i.e., one semaphore, one set of workers). Since a PM-equipped system can have multiple independent regions (different NUMA nodes or non-interleaved DIMMs), there should be separate limits per region.

Our goal in this thesis is efficiency under load. We therefore focus our implementation and evaluation on throughput under load. While the PM is not overloaded, copying data asynchronously to PM increases latency significantly. For this reason, a full implementation should take the current load to PM into account and should always copy synchronously during light load.

Chapter 6

Userspace PM Access Accounting

In the previous chapters, we have established that parallel PM access can lead to wasted CPU time and energy. In particular, parallel writes to PM lead to reduced bandwidth (Chapter 3), which is not handled well by most PM file systems (Chapter 4). To improve CPU and energy efficiency, we propose mechanisms for kernel PM file systems (Chapter 5).

However, these mechanisms can only manage file system accesses via the system call interface. For DAX mappings, the kernel file system only manages creation of the mapping. The userspace process then reads or writes directly to PM, without further kernel involvement.

DAX mappings prevent effective management of parallel PM accesses. Neither the kernel nor userspace processes have enough information on global PM usage, as shown in Figure 6.1. The kernel can easily track any accesses via its system call interface, but cannot account for accesses to DAX mappings. A userspace process can manage parallel accesses to its own DAX mappings, but lacks information about other processes accessing PM. Consequently, even when all PM programs in a system implement best practices and avoid parallel writes, PM will still be overloaded when multiple PM-accessing processes are scheduled at the same time.

We argue that this means a loss of control for the operating system. In the presence of DAX mappings, the kernel is lacking critical information to schedule the available resources among ready processes. With PM, this is especially important since

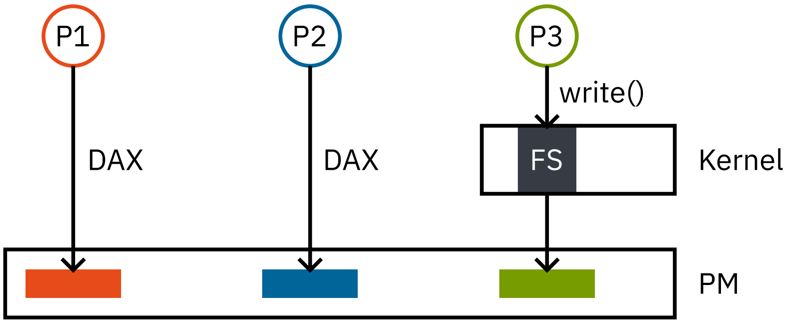


Figure 6.1: In a system with processes accessing PM over DAX mappings, neither the kernel nor the processes can limit parallel PM accesses, since they have no information on PM activity of other processes.

overload leads to reduced performance for the whole system: Processes working with PM experience low throughput, while other processes receive less CPU time as the CPU is busy stalling on PM accesses.

In this chapter, we design and implement a mechanism to restore insight of the kernel into DAX mappings. As the hardware does not provide appropriate performance counters for this purpose, our mechanism works by sampling memory access instructions. We then propose using the accounting information for *core specialization*, which mitigates PM overload from parallel access.

6.1 Requirements

We define the following requirements for an accounting mechanism:

process association It must be possible to account bandwidth to individual processes. Without process association, it would be impossible to identify and throttle processes that overload PM.

device association It must be possible to distinguish bandwidth to different PM devices. A system may have multiple PM regions on separate PM modules. If one PM device is overloaded, there is no need to throttle traffic to other PM devices.

low latency The latency between a PM access and measuring the access must be low. If the latency is too high, effective mitigations are impossible.

low overhead The measurements should not introduce overhead on the measured processes or the system as a whole.

high accuracy The total measured bandwidth should be close to the real bandwidth at the device.

6.2 Accounting with Performance Counters

As described in Section 2.1.3, systems with Optane PM have three types of performance counters with PM events: on-core, off-core response, and uncore. These performance counters cannot satisfy all of the requirements we set above. Table 6.1 shows an overview of the counters and fullfilled requirements. We discuss read and write accesses separately.

6.2.1 Read Accesses

For read accesses, there are two relevant on-core counter events that count L3 cache misses serviced from local or remote PM.[21] Since these events are counted separately for each core, they are always associated with the process currently running on that core. Reading these counter is fast via model-specific registers (MSR). The latency and overhead therefore both depend on the sampling interval. A smaller sampling interval leads to smaller latency, but higher overhead.

There are two problems with these counters. First, they can only distinguish reads to local and remote PM. We therefore cannot associate the counts with individual PM devices. This problem is solved with the PEBS sampling we propose in the following section.

[21] `MEM_LOAD_RETIRED.LOCAL_PMM` and `MEM_LOAD_L3_MISS_RETIRED.REMOTE_PMM`, see Section 2.1.3.

access type	read	write	
counter type	on-core	uncore	Optane module
events	L3 misses serviced from local or remote PM	Number of write commands to PM	Write requests
process assoc.	✓	✗	✗
device assoc.	✗	✗	✓
low latency	✓	✓	✗
low overhead	✓	✓	✓
high accuracy	✗	✓	✓

Table 6.1: Requirements fullfilled by performance counters for accounting read and write accesses to PM.

Second, their accuracy is limited by the prefetcher. The prefetcher may read additional data from PM, reducing the number of counted cache misses. The measured value is therefore a lower bound for the actual read traffic to PM. This problem could be solved by counting prefetch events with off-core response counters (see Section 2.1.3). However, these counters suffer from the same device association problem above, but do not support PEBS.

6.2.2 Write Accesses

For write accesses to PM, we identify two useable counters: uncore counters and counters on the Optane modules.

The uncore counters are located at the memory controller. They therefore measure PM accesses with high accuracy. The same trade-off between latency and overhead as with counters for read accesses applies. Uncore counters do not provide any association with the core causing the access. Thus, we cannot use them for per-process accounting. Just like with the read counters, we also can only distinguish writes to local and remote PM, but not to individual PM modules.

Alternatively, we could use the write request counters on the Optane modules. Since these counters are located on the modules, we obtain per-device counts with high accuracy. However, these counters are not made for high-frequency measurements. On our systems, ipmctl requires approximately 40 ms per Optane module to read the performance counters. Finally, the Optane modules cannot know which process caused a write request, either.

Our PEBS-based sampling provides both process and device association at configurable latency, albeit with reduced accuracy.

6.3 Approach: Sampling Memory Instructions

Regular performance counters cannot provide device and process association, which is necessary for managing parallel write bandwidth to PM. We therefore propose using PEBS to sample load and store instructions, as shown in Figure 6.2. PEBS sampling is local to a core, which provides process association. From the samples,

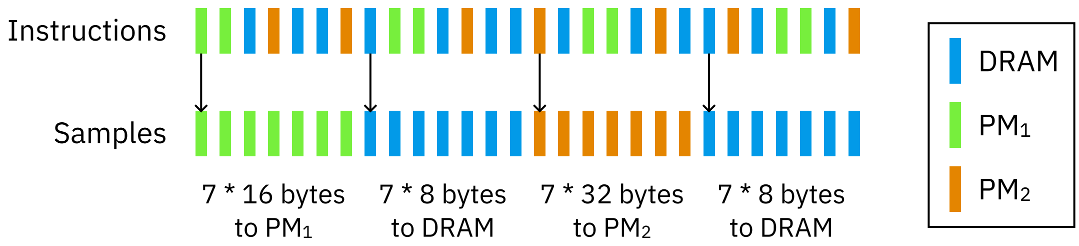


Figure 6.2: Overview of our sampling approach. The CPU periodically (here: every seventh instruction) writes a memory instruction sample. Based on the sampled information, we estimate the number of accessed bytes for the whole sampling period.

we obtain the accessed address for device association and the instruction type to estimate bandwidth.

In the following, we first describe PEBS, then how we use it to estimate bandwidth. Finally, we discuss limitations of this approach.

6.3.1 Processor Event Based Sampling (PEBS)

Processor Event Based Sampling (PEBS) is a feature of Intel processors for sampling the processor state at certain performance counter events [61, Vol. 3B §20.3.1.1.1]. When a PEBS-enabled performance counter overflows, the processor arms the PEBS hardware. At the next monitored event, the armed PEBS hardware writes a *PEBS record* with the current processor state to a memory buffer. A PEBS record contains all integer registers (including the instruction pointer) and optionally the virtual address of memory access instructions [61, Vol. 3B §20.3.8.1.1]. After writing a PEBS record, the processor resets the performance counter to a configurable value, allowing control over the sampling interval. When the memory buffer with PEBS records is full, the processor triggers an interrupt, which prevents missing samples due to overflows.

Only a small subset of performance counter events, called *precise events*, may be used for PEBS [61, Vol. 3B §20.3.8.1.2]. For our purpose of tracking PM accesses, two precise events are relevant:

`MEM_INST_RETIRED` Counts all instructions that access memory. Can be further filtered by loads or stores. We propose using this counter to sample stores (sub-event `ALL_STORES`).

`MEM_LOAD_RETIRED` Counts all instructions that load from memory. Can be further filtered by cache hits or misses for all layers. We propose using this counter to sample loads that miss the L3 cache (sub-event `L3_MISS`).

The processor also provides precise events for loads to local and remote PM. These events could provide higher accuracy if there is no need for accounting DRAM reads.

6.3.2 Bandwidth Estimation

To estimate a per-device bandwidth, we need to know which device was accessed and how much data was accessed. We assume that the PEBS samples are uniformly picked from the monitored instructions. Thus, by scaling measurements from the

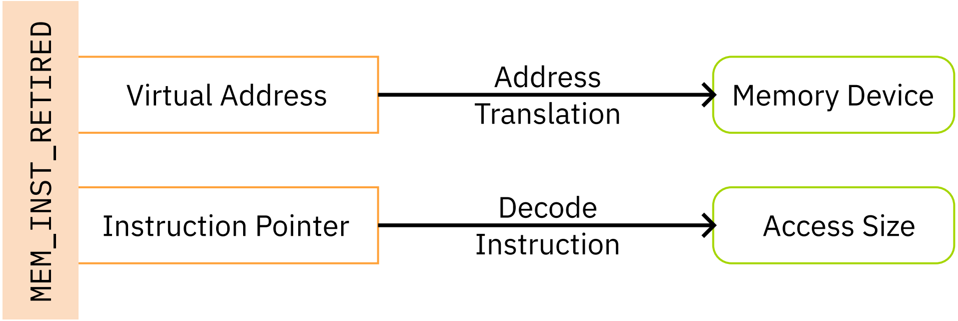


Figure 6.3: Overview of our handling of PEBS samples.

sampled instructions by the sampling interval, we obtain approximations of the real values over time.

As shown in Figure 6.3, we determine the memory device from the virtual address and the access size from the instruction pointer. With a regular address translation of the accessed address, we obtain a physical address. We can then look up the physical address in the physical address ranges of the PM devices.

For the access size, we need to read and decode the instruction at the recorded instruction pointer.[22] The size of the instruction's operands determines the access size.

From the access size S (bytes) and the sampling interval I (number of instructions) we calculate the number of accessed bytes $A = S \cdot I$ and the bandwidth $B = \frac{\Delta A}{\Delta t}$. For each process and memory device, we provide a counter of written bytes (ΣA) and let consumers of these counters calculate the bandwidth.

6.3.3 Limitations

While sampling memory accesses with PEBS provides a bandwidth estimation for each process and device, it comes with limitations regarding latency, overhead, and accuracy.

The estimation latency is controlled by the sampling interval. A smaller sampling interval reduces the estimation latency. However, since PEBS samples from all memory access instruction, we have to observe multiple samples before we can expect a useful estimate. For example, if 10% of an application's memory instructions access PM, only every 10th PEBS record indicates a PM access. When such an application starts running, it can therefore take up to 10 sampling intervals before we detect any PM accesses. Additionally, samples may be processed in batches, further increasing the latency.

Compared to regular performance counters, PEBS introduces additional overhead from recording PEBS samples and processing the sample buffer. Akiyama and Hirofuchi [23] estimate 200-300 ns of CPU overhead for each recorded PEBS sample. Additionally, they observe cache pollution from the samples, which can cause additional cache conflicts for cache-sensitive applications. Overhead from sample processing depends on the sampling interval (how many samples are produced) and

[22] A PEBS record contains two instruction pointers: of the instruction that triggered the event, and of the following instruction. We need the former instruction pointer here.

the PEBS buffer size (how often are samples processed). We evaluate the overhead from sample processing in our prototype below.

There are multiple sources of inaccuracy. For loads, the limitation regarding prefetching discussed in Section 6.2.1 applies.

We cannot detect stores that overwrite dirty cache lines. In such a case, the previous store to that cache line has not caused a memory access, resulting in an overestimated write bandwidth. When using PM for persistence, cache flushes are necessary for crash consistency. We therefore assume that stores never overwrite dirty cache lines and are always immediately flushed to memory.

The performance counter for store instructions also counts cache flush instructions such as `clwb`. This introduces an accounting error for workloads that include cache flush instructions, as there are now two instructions for one memory access. We only count proper store instructions and ignore cache line flushes. In Section 6.6.1, we evaluate the accounting error from cache line flush instructions. We leave heuristics for correcting this error as future work.

Finally, PEBS may show a bias in which instructions are sampled. Yi et al. describe the *shadow effect* [129]: After a performance counter overflow, there is a delay before the PEBS hardware is armed. Any events during this delay are hidden from sampling. If there are long-running instructions such as memory fences, the PEBS hardware is more likely armed during such longer instructions. It will therefore sample instructions immediately after long-running instructions with a higher likelyhood. Such a bias can cause estimation errors.

6.4 Implementation

We implement a prototype of PEBS-based write bandwidth estimation as a Linux kernel module. It sets up PEBS directly by writing to model-specific registers since Linux's perf_event API [18] is not designed for use from a kernel module. To allow controlling latency and sampling overhead, the PEBS buffer size and the sampling interval are configurable as module options.

The module processes PEBS samples either when the PEBS buffer is full (via the interrupt) or on task switch. It finds PM ranges based on the kernel's iomem ranges after a software page table walk. The prototype only distinguishes accesses to PM and DRAM, but could support fine-granular counters for separate PM address ranges. To determine the memory access size, it uses the Zydis disassembler library [30], which provides the size of decoded operands.

The module makes the counters available over multiple interfaces. The per-task counters are stored in `struct task_struct`. Additional files in procfs allow reading their values from userspace.

For low-latency monitoring and control, the module additionally provides a shared memory buffer with the counters of currently-running tasks. Userspace applications can memory-map this file and read the current counters for each CPU core.

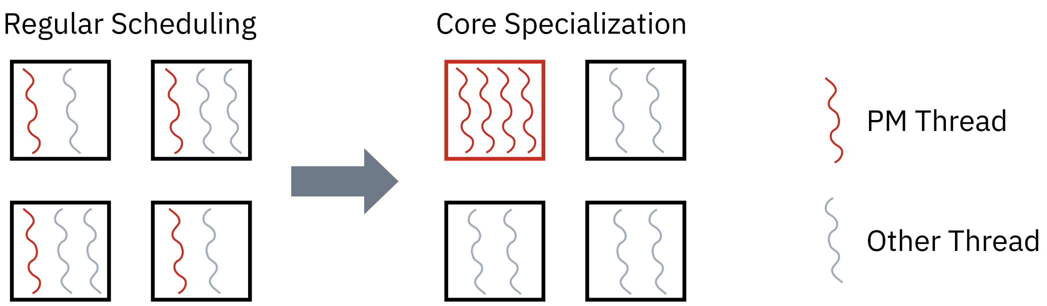


Figure 6.4: Core specialization can reduce PM overload by identifying threads accessing PM and pinning them to a subset of the available cores. By reducing stall cycles from the PM threads, this strategy makes more CPU time available to other non-PM threads.

6.5 Scheduling

PM bandwidth and parallelism management are highly dependent on the application. Since a scheduling mechanism can only control a process's CPU activity as a whole, any process throttling for PM accesses also throttles other work in that process unrelated to PM access. PM applications should therefore directly use the accounting information and apply targeted throttling of their PM accesses accordingly.

Nevertheless, we now propose a generic scheduling approach for mitigating slowdown and CPU stalls from parallel PM accesses in userspace applications. The idea is to employ *core specialization*, as illustrated in Figure 6.4. In a PM overload situation (left), threads accessing PM are scheduled on multiple cores at the same time. These threads saturate the available PM bandwidth, and their PM accesses stall the cores. They therefore also reduce the available CPU time for other processes that do not access PM.

Our approach introduces a monitor process that reads the accounting information for all currently running threads. If the threads saturate the available PM bandwidth, the monitor enables core specialization. It pins all threads that access PM to a subset of the cores. The result is shown on the right side of Figure 6.4: The PM threads share a single core (red) and therefore can no longer overload PM. Even though each PM thread has less CPU time available due to the restriction to the specialized cores, they now make more progress since the number of CPU stall cycles is reduced. All other threads (grey) are rebalanced to the remaining cores by the scheduler. These threads now have more CPU time available since they no longer need to share time with the stalling PM threads.

If the monitor detects that a pinned thread no longer accesses PM, it reverts the pinning. Once the number of active PM threads falls below a threshold, the monitor stops core specialization and allows PM threads to run on all cores again.

The primary challenge with this approach is choosing the number of specialized PM cores. If the number is too large, the PM threads are still able to overload PM with their parallel accesses. A number too low could take CPU time away from PM threads that do not exclusively access PM. The accounting data alone does not allow detection of whether a thread uses little PM bandwidth due to stalls or because it is performing other work. The approach therefore requires fine-tuning according to the expected workloads.

6.6 Evaluation

Our evaluation aims to answer the following questions:

- Can the accounting mechanism accurately detect the number of bytes written by an application?
- What is the trade-off between performance overhead for the application and the latency of the accounting?
- Can our scheduling approach based on core specialization prevent PM overload?

We choose prime numbers as PEBS sampling intervals to avoid repeated sampling of the same instruction in a loop smaller than the sampling interval. In the following, we compare three PEBS sampling intervals: 1999, 4999, and 9973.

6.6.1 Accuracy

We evaluate the accuracy of the accounting mechanism with a microbenchmark that writes a fixed amount of data to PM. We then compare the accounting result with the actual number of written bytes. The microbenchmark is implemented as a memset loop that uses one specific instruction to write sequentially to PM. We test non-temporal store instructions and regular (temporal) store instructions with multiple access sizes. For the regular stores, we also test flushing each written cache line (64 bytes) with a `clwb` instruction. We run the microbenchmark with multiple PEBS sampling intervals (see above) and parallel threads (1 to 8).

In the implementation of the microbenchmark, we ensure that the core memset loop contains only memory accesses writing to PM, as shown in Listing 6.1. The benchmark therefore verifies that our measurement method does not introduce any inherent inaccuracy.

Figure 6.5 shows the results. The accuracy for store instructions without cache flushes is very high, with a maximum error of approximately 0.5% for 8 byte temporal stores.

For the benchmarks that include cache line flush instructions, we see a larger error. Smaller access sizes of 8 and 16 bytes are overestimated by 11% and 24%,

```
00:  lea     (%rdi,%rcx,1),%rax
04:  movntdq %xmm0,(%rax)
08:  add     $0x10,%rcx
0c:  cmp     %rsi,%rcx
0f:  jb      00
```

(a) Non-temporal stores with access size 16 bytes

```
00:  lea    (%rdi,%rcx,1),%rax
04:  movdqu %xmm0,(%rax)
08:  movdqu %xmm0,0x10(%rax)
0d:  movdqu %xmm0,0x20(%rax)
12:  movdqu %xmm0,0x30(%rax)
17:  clwb   (%rax)
1b:  add    $0x40,%rcx
1f:  cmp    %rsi,%rcx
22:  jb     00
```

(b) Temporal stores with access size 16 bytes and `clwb`

Listing 6.1: Disassembly of the memset loop writing to PM. All store instructions write to PM.

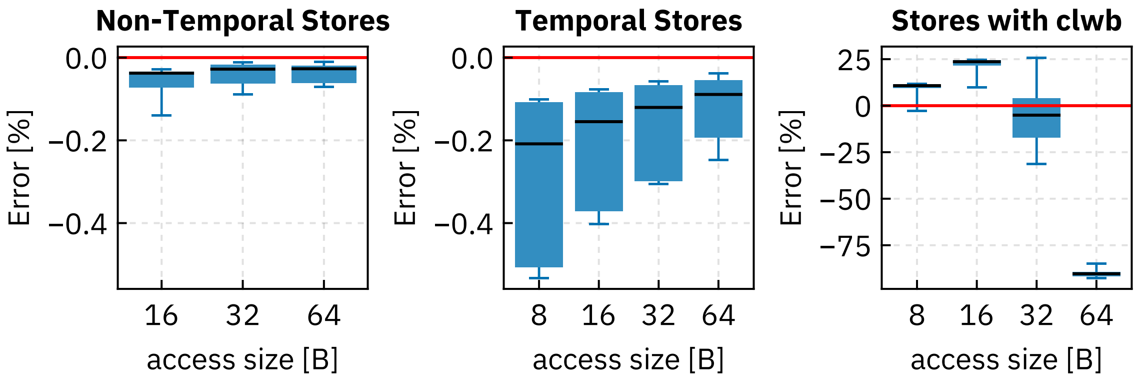


Figure 6.5: pc61 Boxplot of the accounting error for a userspace write benchmark with different instruction types and access sizes. The benchmark writes with one to eight threads, and the accounting uses a PEBS sampling interval of 1999, 4999, and 9973. The whiskers show the minimum and maximum values.

respectively (median). This overestimation is explained by the additional `clwb` instructions, which are less likely to be sampled due to the *shadow effect* (see Section 6.3.3). Since the `clwb` instruction takes longer than the small memory accesses, the store instruction after `clwb` is sampled more often. 11% and 20% of the instructions are `clwb` instructions in our benchmark with access sizes of 8 and 16 bytes, respectively, which matches the observed error.

With larger access sizes of 32 and 64 bytes, the runtime of the store instructions increases, and the shadow effect reverses. At 32 bytes, we observe roughly equal numbers of store instructions and `clwb` instructions, resulting in a medium underestimation of 5.1%. With 64 byte store instructions, there are equal numbers of stores and cache line flushes. However, the `clwb` instructions are now sampled more often than the store instructions, resulting in a large underestimation of 90%.

6.6.2 Overhead and Latency

To measure overhead and latency, we instrument the kernel module. We capture measurements of the runtime of PEBS event processing and timestamps when events are processed. As the accounting target, we run a fio *devdax* benchmark that writes to PM from userspace (see Section 4.2.3). With fio, we observe a mix of PM and DRAM events.

There are two parameters that control overhead and latency. The sampling interval controls how often events are sampled by the PEBS hardware. A higher sampling interval reduces the number of events and therefore decreases the overhead but increases the latency. The PEBS buffer size controls how many events are collected for batch processing. A larger buffer reduces the overhead by reducing the number of PEBS interrupts but increases the latency, as multiple events need to be collected.

Overhead

Figure 6.6 shows the processing time for individual events as well as totals for fio writing 1 GiB of data to PM. The sampling interval has little effect on the per-event processing time. However, processing events in batches improves the processing

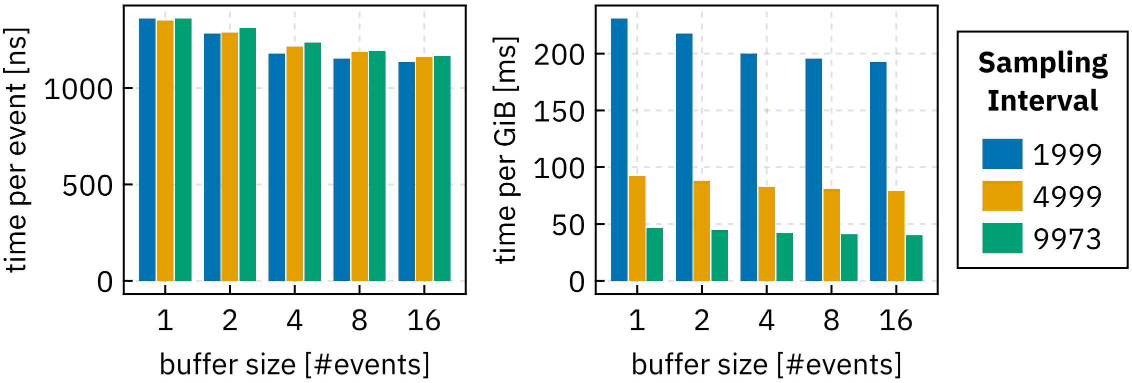


Figure 6.6: pc61 Processing time per event (left) and per GiB written by a fio benchmark (right).

time slightly, likely due to better cache utilization: From a buffer size of one event to two events, the processing time decreases by 4.7%, up to 15% at 16 events.

For the total processing time for writing 1 GiB, the sampling interval has a much larger effect than the buffer size. From sampling interval 1999 to 4999, the average processing time decreases by 59%, and another 49% from 4999 to 9973. This inverse proportionality arises since a larger sampling interval results in fewer events per GiB.

Latency

In Figure 6.7, we plot density distributions of the latency between processing two batches of PEBS events. We show latency for all PEBS events (i.e., PM and DRAM) in the left column, and latency after filtering for PM events in the right column.

We first discuss the results for a buffer size of one event (bottom row). The observed latencies for all PEBS events (left column) are approximately proportional to the sampling interval, with distributions centered around median latencies of 56 µs,

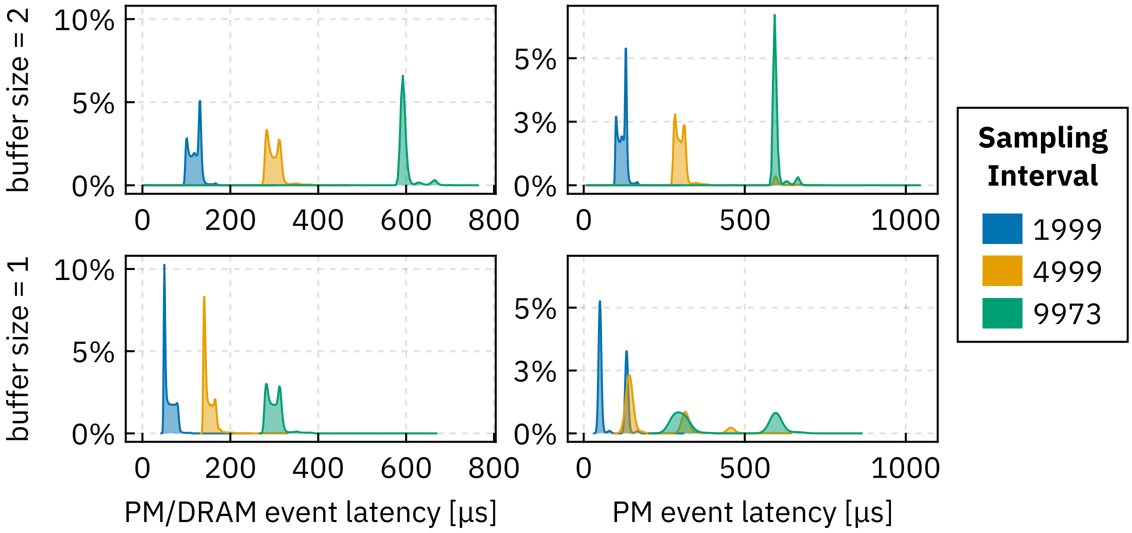


Figure 6.7: pc61 Density distribution of event latency for PEBS buffer sizes 1 and 2 (rows). The left column shows the distribution for any event and the right column shows only PM events.

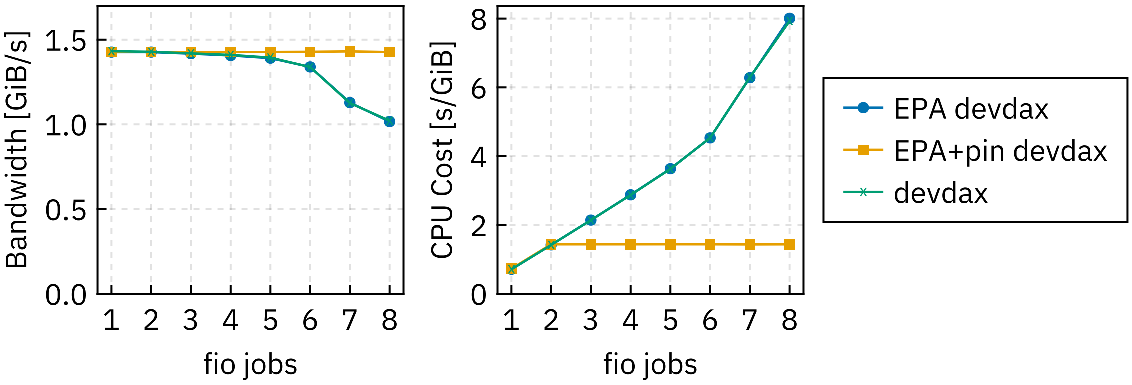


Figure 6.8: pc61 Bandwidth and CPU cost of a fio devdax benchmark. We compare *devdax* as the baseline with enabled accounting (*EPA devdax*) and with core specialization (*EPA+pin devdax*).

150 µs, and 300 µs for the sampling intervals 1999, 4999, and 9973. After filtering for PM events, a new pattern emerges. Secondary peaks are now visible at multiples of the latency at the primary peak, which follows from the filtering: If an event is a DRAM event, another PM event can appear earliest after one sampling period.

Batching events with a buffer size of two events (top row) doubles the observed latencies, with new median latencies of 120 µs, 300 µs, and 590 µs. However, since there are now two events in each batch, there is a higher likelyhood of observing at least one PM event in every batch. The secondary latency peaks for the PM events therefore disappear in the top right plot.

Given these results, we recommend a buffer size of one event and controlling the latency/overhead tradeoff only by choosing a sampling interval. The sampling interval is roughly proportional to both processing time per GiB of written data and latency. In contrast, a larger buffer size increases latency proportionally, but has only a minor effect on processing time.

6.6.3 Scheduling

We evaluate our scheduling based on core specialization with the fio *devdax* benchmark (see Section 4.2.3). In Figure 6.8, we compare three configurations: plain fio without accounting or scheduling (*devdax*), fio with accounting but no scheduling (*EPA devdax*), and fio with accounting and scheduling (*EPA+pin devdax*). For the accounting, we set a sampling interval of 9973 and a buffer size of one event. We configure our scheduling to pin PM-heavy processes to two cores and record fio bandwidth and CPU cost (see Chapter 4).

Comparing *devdax* with and without accounting, we observe that the accounting has no noticeable effect on the bandwidth or the CPU cost. This is consistent with our previous measurements, as the expected overhead is less than 50 ms per GiB (Figure 6.6).

The automatic pinning achieves the intended result. The total bandwidth remains stable after five jobs, whereas it drops without our scheduling. Since the pinned fio jobs run only on two CPUs, our approach also effectively limits the CPU cost.

6.7 Related Work

Multiple previous works have proposed using PEBS to monitor memory accesses.

Nonell et al. [97] show that monitoring load addresses with PEBS is feasible in an HPC environment. They evaluate very small sampling intervals (64 to 256) with larger PEBS buffers (32 KiB, fitting 170 samples), showing a maximum overhead of 10%. They propose using the PEBS data to detect access patterns and steer allocations in heterogeneous-memory HPC systems.

HeMem [102] uses PEBS to inform tiered memory management. HeMem samples memory load and store instructions to track per-page access counts and classify pages as hot or cold for tiering. In contrast to our approach, HeMem does not estimate bandwidth from the samples.

As an alternative to PM monitoring with PEBS, Dicio [100] uses performance counters for write pending queue delay to detect PM overload. In contrast to our approach, Dicio cannot attribute this overload to specific processes. Instead, it distinguishes between latency-critical and best-effort jobs. In an overload situation, all best-effort jobs are throttled to ensure sufficient bandwidth for the latency-critical jobs.

Finally, Gottschlag et al. [47] use core specialization to isolate AVX-512 workloads. When executing AVX-512 instructions, some CPUs reduce their frequency, which can slow down unrelated processes. The core specialization therefore isolates the frequency reduction to specific cores. This is in contrast to our approach, where core specialization reduces load at PM as a shared resource.

6.8 Discussion

Once the operating system hands out a DAX mapping to a userspace application, it cannot control access to this mapping anymore. This is problematic for PM mappings, since PM suffers from overload if too many threads access it in parallel. In this chapter, we propose an approach for monitoring access to DAX mappings by sampling memory access instructions. We show that our approach can mitigate parallel PM accesses with automatic core specialization.

Our approach fulfills the requirements set in Section 6.1. PEBS samples are collected for each CPU core, allowing association with individual processes and threads. By translating virtual addresses, the approach can associate memory accesses with an arbitrary number of memory devices based on their physical address ranges.

Our evaluation shows that the choice of sampling interval can effectively control the latency–overhead trade-off. At an event latency of under 500 µs, an overhead of less than 50 ms per GiB of data written to PM can be expected.

The primary challenge for the accuracy of our approach is the CPU caches. Since PEBS sampling occurs at the CPU core, there is no way to detect whether a store remains in the CPU caches or is immediately flushed to memory. Further, cache-line flush instructions are hard to account for, since they are not uniformly sampled by the PEBS hardware.

As future work, we expect that future (CXL) memory devices could support the operating system in providing more accurate accounting mechanisms with lower overhead. At the device level, there is no risk of inaccuracy due to CPU caches. However, the challenge of providing process association remains. We propose that future PM devices allow associating device memory ranges with address space identifiers and provide counters for read and write accesses for each address space. The operating system could then provide accounting information as in this chapter by arming and reading the on-device counters when mapping pages and scheduling processes.

Chapter 7

Crash Consistency Testing

Achieving crash consistency in persistent memory applications is difficult. As discussed in Section 2.4, the developer must carefully insert cache line flush and memory fence instructions. These instructions do not affect the correct operation of the application during normal runtime. Rather, they have a visible effect only in the event of a crash.

Crash consistency testing tools allow developers to verify the crash behavior of their applications. In this chapter, we review previous work on crash consistency testing. In particular, we describe the crash consistency pipeline that multiple previous works implement. **Suvi**, the crash consistency tester we introduce in Chapter 8, also builds on top of this pipeline. Finally, we describe crash consistency testing tools related to **Suvi**.

7.1 Failure Points and Crash Images

An application is crash-consistent if it never produces inconsistent PM contents in the event of a crash. To test applications for this property, we need to consider where in the program code the application could crash (*failure points*) and what the PM at these points might contain (*crash images*).

Since the time of a crash cannot be predicted or controlled, we need to consider every instruction of the program as a potential failure point for testing. However, we can reduce the number of failure points for testing by combining equivalent points. Instructions other than PM primitives or those writing to PM do not directly change PM contents. Testing failures at these instructions will always yield the same results as testing at adjacent PM primitives or PM stores. Note that we cannot generally detect such instructions statically, since the addresses of memory accesses or cache flushes are calculated at runtime.

Crash consistency testing approaches that consider reordered PM modifications (e.g., due to out-of-order execution or caches; see Section 2.4) can further reduce failure points to *ordering points*. These are instructions, such as memory fences, that act as a barrier for out-of-order execution.

At the failure points, a crash consistency tester verifies that no inconsistent PM states are possible based on PM modifications that are not completely persisted (i.e., flushed from caches and ordered with a fence). To confirm crash consistency bugs, it can generate crash images as witnesses. A crash image represents possible PM contents at a failure point. It is constructed by combining the fully-persisted PM contents with in-flight modifications at the failure point. We discuss crash image generation in **Suvi** in Section 8.2.

We distinguish two types of crash images. A *complete crash image* is a crash image that includes all in-flight modifications at the failure point. There is always exactly one complete crash image at every failure point. A *partial crash image* is a crash image that contains only a subset of in-flight modifications. If there is more than one PM modification at a failure point, multiple partial crash images may exist. Note that neither complete nor partial crash images are unique and may appear at multiple failure points.

To check a crash image for consistency, its contents need to be recovered by the PM application.

7.2 Types of Crash Consistency Bugs

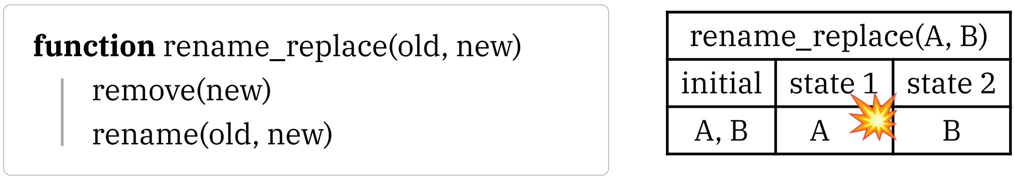
We distinguish three types of crash consistency bugs: logic bugs, missing flushes, and ordering bugs. Discussing these types separately is useful because they require different strategies for detection.

7.2.1 Logic Bugs

A PM application has a logic bug if a broken PM state appears when modifications are applied in strict program order. Such a bug appears regardless of the use of PM primitives. They are not caused by missing cache flushes or memory fence instructions [81].

Figure 7.1 shows an example of a logic bug in a file system.[23] The rename file system operation should replace the destination file atomically if it exists [14]. A broken implementation might remove the destination file first and then perform the rename operation. If protected by a lock, such an implementation would behave correctly at runtime but would produce an invalid crash state in which the destination file is missing.

LeBlanc et al. identify logic bugs as the most common type of bug in PM file systems [81]. To detect logic bugs, it is necessary to examine the semantic state stored on PM rather than only analyze the use of PM primitives.



```
function rename_replace(old, new)
    remove(new)
    rename(old, new)
```

rename_replace(A, B)		
initial	state 1	state 2
A, B	A 💥	B

Figure 7.1: Pseudocode example of a logic bug in a PM file system. The function should atomically replace file *new* with *old*, but an intermediate state exists where *new* does not exist.

[23] We found a similar logic bug in NOVA's rename function with Vinter [68].

7.2.2 Missing Flush

PM applications need to use cache flushes or non-temporal store instructions to ensure that their modifications reach PM. If a modification is missing a flush, it may remain in the volatile CPU caches and will be lost in the event of a crash.

An unflushed modification is not necessarily a correctness bug, since PM applications can use PM to store transient data. However, given the lower performance of PM compared to DRAM, using PM for transient data could still be considered a performance bug.

Missing flushes can be detected with approaches such as symbolic execution [96] or trace analysis [46, 86]. These approaches cannot determine whether the missing flush constitutes a correctness bug. In Vinter and **Suvi**, we detect correctness bugs due to missing flushes by checking whether multiple semantic states are possible after an operation.

7.2.3 Ordering Bug

Modifications may not reach PM in the order of the program's store instructions. As described in Section 2.4.1, there are multiple sources of reordering. The cache can write back cache lines in an unpredictable order due to conflicts. Store instructions (e.g., non-temporal stores on x86) may be weakly ordered and thus subject to reordering by the CPU. PM programs need to use memory fences to enforce a specific ordering.

An ordering bug arises when memory fences are missing in a way that allows broken states on PM. Figure 7.2 shows an example where fences are critical to avoid an ordering bug. Since the valid flag protects accesses to the data, it must not be set to 1 while the data is not completely written.

There are multiple approaches for finding ordering bugs in PM applications. An obvious, but often impractical idea is to generate crash images with all possible orderings of in-flight modifications. This approach suffers from combinatorial explosion: For N independent writes, there are 2^N possible crash images. Crash consistency testing tools that generate crash images to detect ordering bugs therefore need a strategy for reducing the search space.

Alternative approaches that do not rely on testing crash images include manual annotation of persistence checks (PMTest [86]) or detecting dependencies between modifications and the recovery code (XFDetector [85]).

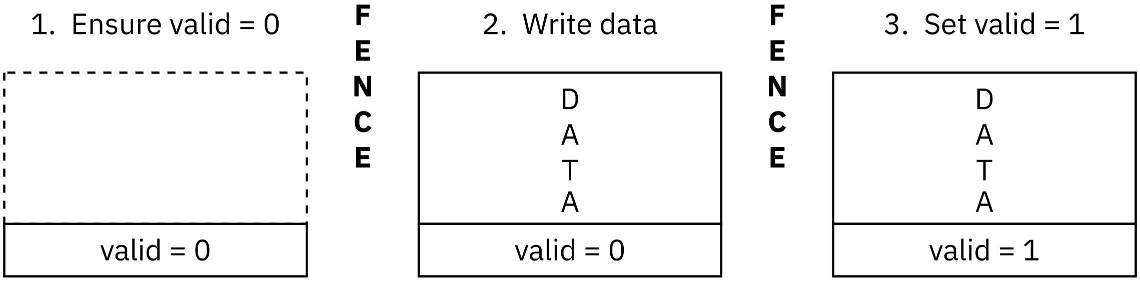


Figure 7.2: Steps when writing a journal entry (compare Section 2.4). Without separating the modifications to the data and the valid flag with fences, we would encounter an ordering bug.

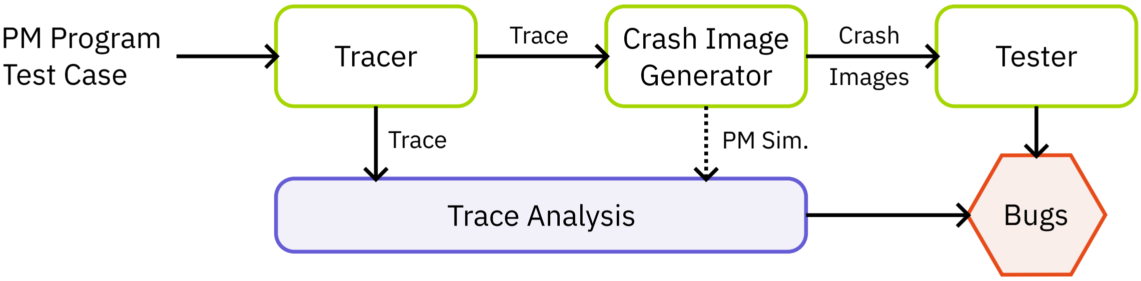


Figure 7.3: Components of a generic crash consistency testing pipeline. Variants of such a pipeline are found in most crash consistency testing tools.

7.2.4 Performance Bugs

Use of PM primitives can have an effect on performance. For example, memory fences act as a barrier to out-of-order execution, stall the CPU pipeline and reduce utilization of the core's execution units. Application developers therefore generally want to avoid unnecessary PM primitives. Since extra cache flushes or memory fences do not affect the crash consistency of PM applications, we categorize these cases as performance bugs.

Detection of performance bugs in previous work is generally restricted to simple cases [45, 46, 85, 96]. These include cache flushes for cache lines without modifications and memory fences without any pending stores to PM. However, the testing tools cannot confirm whether these extra instructions actually harm performance.

7.3 Crash Consistency Testing Pipeline

A crash consistency testing pipeline has three primary stages, as shown in Figure 7.3. The tracer performs a dynamic analysis of the tested application and creates a trace of PM accesses and crash consistency primitives. The crash image generator replays the trace, keeping track of PM contents and in-flight modifications. At failure points in the trace, it generates crash images. The tester checks the crash images for inconsistencies, usually by running a recovery program.

Besides the analysis based on crash images, crash consistency testing tools can also analyze the trace directly for potential bugs. Trace analysis directly yields bug reports from patterns in the trace or based on the PM and cache simulation during replay. Since it does not use crash images as witnesses of a broken PM state, trace analysis can generally only hint at bugs, but not confirm them.

We describe the crash consistency testing tools in the following section in terms of the crash consistency pipeline in Figure 7.3. The authors of the tools might use other terms (e.g., replayer instead of crash image generator) or combine some stages (e.g., crash image generator and tester).

7.4 Tracing Approaches

There are multiple approaches for tracing memory accesses and PM primitives. They differ in how much overhead they cause and how much manual work is required from the developer to set up tracing. Table 7.1 shows an overview of the approaches.

Approach	black-box	overhead	risk of user error
Binary Translation	✓	high	none
Compiler Instrumentation	✓	medium	none
Function Tracing	✗	low	medium
Manual Annotation	✗	low	high

Table 7.1: Comparison of tracing approaches.

7.4.1 Binary Translation

A virtual machine (VM) with binary translation works by translating each instruction from the source ISA to the host ISA. Virtualization with binary translation is possible at the level of processes or for a full system. The primary purpose of such VMs is to run programs compiled for a different ISA than the host ISA (e.g., x86 programs on an ARM CPU). By inserting hooks into specific instructions during translation, binary translation allows inspecting and tracing the execution of the VM.

Binary translation allows for black-box testing. It works with unmodified binaries. As all instructions are translated, a tracer based on binary translation is guaranteed to detect and trace all PM accesses and crash consistency primitives. However, modifications may be necessary if the program uses instructions that are not supported by the emulator. For example, QEMU [28] does not support AVX-512 instructions, so these instructions need to be replaced for testing.[24]

The main drawback of binary translation is its high overhead.[25] Every instruction needs to be translated, not just traced instructions. To detect PM accesses, every memory access instruction needs instrumentation.

Examples of emulators with full system virtualization usable for tracing are QEMU [28] and PANDA [38]. Intel Pin [88] implements userspace binary translation.

7.4.2 Compiler Instrumentation

The compiler can insert tracing hooks while compiling a program. This approach is similar to binary translation in that tracing is applied automatically at the instruction level. It therefore enables black-box testing with no risk of user error.

Compiler instrumentation introduces less overhead than binary translation since the binary does not run in a virtualized environment, avoiding the need to rewrite all instructions. Additionally, all instrumentation occurs statically at compile time. However, the compiler in general cannot detect whether a memory access targets PM or DRAM. It therefore must insert hooks for all memory accesses.

The main drawback of compiler instrumentation is that it is less generic than binary translation. The target program must be available as source code in a programming language supported by the compiler.

Witcher [45] is an example of a PM crash consistency testing approach that uses an LLVM compiler pass to instrument programs for tracing.

[24] PM software often assumes that AVX-512 is available since it is supported by all CPUs that work with Optane PM (see Section 2.1).

[25] Previous works have estimated an overhead between 5x and 20x [104].

7.4.3 Manual Annotation

Manual annotation allows the most flexibility and lowest overhead for tracing. The developer manually inserts calls in the PM application to trace PM events. Consequently, there is no unnecessary instrumentation. Support for both kernel and userspace software is possible. Tracing high-level abstractions is possible, for example PMDK transactions [19].

However, manual annotations carry a high risk of user error. There is no guarantee that the annotations match the actual application behavior. Additionally, fully annotating a PM application requires a large amount of effort from the developer.

PMTest [86] is a crash consistency testing tool that relies on manual annotation. We describe it in Section 7.5.2.

7.4.4 Function Tracing

Function tracing instruments a program at the level of functions. Such instrumentation is possible with low overhead using mechanisms such as Linux Kprobes [70] and Uprobes [40]. This approach is especially useful if the PM program uses an abstraction layer such as PMDK [19] for accessing PM.

The primary challenge with this approach is translating high-level traced functions to low-level PM primitives. Depending on the functions, this translation can require considerable manual effort and carries a high risk of errors.

Chipmunk [81] is a crash consistency testing tool based on function tracing. It is designed for testing PM file systems, where no common abstraction layer for PM access exists. We describe its tracing approach in more detail in Section 7.5.6.

7.5 Crash Consistency Testing Tools

We now examine previous approaches to crash consistency testing of PM file systems. Figure 7.4 shows a timeline of PM file systems and the crash consistency testing tools that were evaluated on these file systems. Section 2.3 provides an overview of the PM file systems. We describe previous approaches to crash consistency testing that support file systems or that introduced ideas adopted in **Suvi**. Table 7.2 compares the pipeline stages of these crash consistency testing tools. We describe each tool in more detail in the following sections.

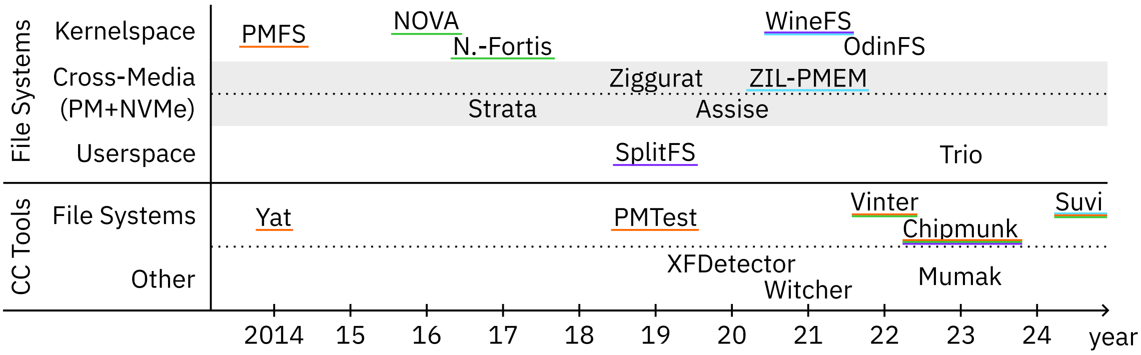


Figure 7.4: Timeline of PM file systems and crash consistency (CC) testing tools. Underlines indicate the file systems each tool was evaluated on.

Tool	**Tracer**		**Crash Images**	**Tester**	**Trace Analysis**
	Approach	K			
Yat [80]	hardware VM	✓	random subsets	run recovery, fsck	-
PMTest [86]	manual annotation	✓	-	-	manual checking rules
XFDetector [85]	binary translation	✗	no subsets	run recovery	post-failure reads
Witcher [45]	compiler pass	✗	likely-correctness	output equivalence	performance bugs
Vinter [68]	binary translation	✓	reads heuristic	check unique states	-
Chipmunk [81]	function tracing	✓	function coalescing	oracle state comparison	-
Mumak [46]	binary translation	✗	deduplication by stack trace	check recovery	detect patterns

Table 7.2: Comparison of crash consistency testing tools. Column "K" indicates kernel support; these tools were used to test PM file systems.

7.5.1 Yat

Yat [80] was the first crash consistency testing tool for PM file systems. It was specifically developed for testing PMFS [41]. Yat introduces the basic crash consistency testing pipeline, consisting of a record phase (called tracer in this thesis) and a replay phase combining crash image generation and testing of images.

Yat was introduced long before real systems with PM were on the horizon. Its PM model therefore included a *pm_wbarrier* instruction (later called `pcommit`, see Section 2.4.1) for flushing data between the memory controller and the PM modules. Compared to **Suvi**, it implements a simpler crash consistency model that does not take global store order into account.

Yat's record phase runs the test case in Intel's internal hypervisor. Like **Suvi**, it therefore has full support for kernelspace file systems. Unlike **Suvi**, Yat's tracing is not based on binary translation. The hypervisor is based on hardware virtualization and traces memory accesses with page table permissions and exceptions. For tracing cache flush and pm_wbarrier instructions, it requires recompilation of the traced software to insert illegal instructions.

Yat's replay phase generates crash images at every pm_wbarrier instruction by selecting subsets of active writes. To combat combinatorial explosion, it implements a simple heuristic that coalesces adjacent writes to the same cache line into a single, atomic write. This heuristic is aimed at memcpy loops but may cause false negatives. If the number of subsets is still too large, Yat selects a limited number of random subsets.

Yat tests each crash image by loading it in the VM, running the recovery, and verifying consistency with an fsck application. In contrast to **Suvi** and other later crash consistency testers, it does not compare the output state across different crash images.

The authors used Yat to test PMFS during its development. They do not describe the test cases they used for the record phase.

7.5.2 PMTest

PMTest [86] is a crash consistency testing tool based on manual annotation. At each PM access, cache flush, and memory fence, the developer needs to insert calls into PMTest's library. In addition, PMTest needs *checking rules* in the application under test. These rules assert a particular state about PM objects, such as persistence or ordering with other objects. The PMTest library serializes each PM access, cache flush, memory fence, and checking rule as a trace entry.

When running an annotated application, the PMTest library passes trace entries to the PMTest checking engine running in the same process via shared memory. PMTest also supports kernelspace PM applications. In this case, the kernel library passes trace entries to a userspace PMTest checking engine via a FIFO.

The checking engine processes the trace entries with multiple worker threads. It replays the PM accesses to *shadow memory* while keeping track of the persistence status of all modified addresses. When encountering a checking rule, the checking engine verifies the assertion. If any assertion fails, PMTest yields a bug.

We categorize PMTest as a form of trace analysis. Even though it replays the trace while keeping track of PM contents, it never yields crash images.

PMTest is relevant to this thesis since the authors demonstrate its use on PMFS [41]. However, we argue that PMTest's required manual annotation by the file system developer is error-prone and may lead to missed bugs. Later approaches to crash consistency checking, including **Suvi**, feature black-box testing and automatic bug detection.

7.5.3 XFDetector

XFDetector [85] is a crash consistency testing tool that introduces the concept of *post-failure tracing*. It detects bug by matching read accesses of the post-failure recovery procedure with pre-failure writes, cache flushes, and fences.

XFDetector's tracer is based on userspace binary translation with Intel Pin [88]. It therefore does not support testing kernelspace PM file systems. In addition to low-level tracing of PM accesses, cache flushes, and fences, XFDetector can also trace PMDK [19] library calls. The tracer is used both for pre-failure tracing of the test case, and for post-failure tracing of a recovery procedure.

XFDetector generates one crash image containing all modifications at each *failure point*. The developer needs to manually annotate a region of interest, in which XFDetector automatically inserts failure points before every *ordering point* (fences or PMDK writeback functions). In addition, the developer can annotate additional failure points.

For each crash image, XFDetector creates a post-failure trace of the application's recovery program. It then performs trace analysis with both the pre-failure and the post-failure trace. For every read access in the post-failure trace, it checks for *cross-failure races* with writes in the pre-failure trace. XFDetector replays both traces while tracking the persistence state of all modified memory locations. If a value read in the post-failure trace was not properly persisted in the pre-failure trace, then the application has a cross-failure race. However, not all cross-failure races are bugs. XFDetector requires manual annotation of *commit variables*, for which cross-failure races are allowed since they protect other data.[26]

While XFDetector cannot test PM file systems, its idea of post-failure tracing was influential for Vinter's reads heuristic (described below), which **Suvi** inherits.

7.5.4 Witcher

Witcher [45] is a crash consistency testing tool that introduced *likely-correctness conditions* for crash image generation and *output equivalence checking* for testing crash images.

Witcher's tracing is implemented as an LLVM compiler pass that creates an instrumented binary. Besides memory accesses, cache flushes, and fences, Witcher also traces control flow instructions.

To generate crash images, Witcher first analyzes data and control dependencies with static analysis of the program binary (during instrumentation) and with dynamic trace analysis. Based on these dependencies, Witcher infers likely persistence orderings between memory locations. Witcher's authors call the set of inferred persistence orderings for the whole trace *likely-correctness conditions*. In contrast to the other approaches we describe here, Witcher requires test cases that not only write to PM, but also read the data back, so that the trace contains appropriate data and control dependencies.

Witcher then replays the trace while keeping track of PM contents. If a PM state at a fence could violate a likely-correctness condition, Witcher yields a crash image that includes the violating modifications. However, such a violation is not always a bug.

To confirm a bug, Witcher performs *output equivalence checking*. The authors implement this check specifically for testing *durable linearizability* in key-value stores. It requires two oracles. The first oracle records the output of the test case (i.e., a query of the key-value store) without a crash, and the second oracle records the output with the crashing operation rolled back. Witcher reports a bug if the output from the recovered crash image does not match the output of one of the oracles.

Witcher also implements trace analysis for performance bugs. It detects unnecessary flushes and fences.

Although Witcher's output equivalence checking is limited to key-value stores, it has inspired similar mechanisms for confirming crash consistency bugs in later approaches for testing file systems. Vinter and **Suvi** confirm atomicity bugs by identifying unique states for each operation. Chipmunk implements an oracle for file system operations.

[26] See our example in Section 2.4, where the "valid" flag is a commit variable.

7.5.5 Vinter

Vinter [68] is our original approach to black-box crash consistency testing of PM file systems. It originated from the master's thesis of Samuel Kalbfleisch [69]. We extend Vinter to **Suvi** in this thesis.

Vinter's tracer is based on full system emulation with binary translation. It uses PANDA [38], which offers hooks for memory accesses and arbitrary instructions. Compared to Yat [80], Vinter does not require manual annotation of cache flushes and memory fences and can trace unmodified file systems. Vinter's tracer can optionally capture stack traces for each trace entry, which help with debugging.

Similar to previous works, Vinter generates crash images by replaying the trace and generating subsets at each fence. To avoid combinatorial explosion, Vinter introduces a heuristic based on recovery reads. Like XFDetector [85], it performs post-failure tracing with a fully-persisted crash image (i.e., including all current modifications). Vinter then only considers modified cache lines that were read post-failure for generating crash images with subsets. We improve this heuristic for **Suvi** and describe it in more detail in Section 8.4.1.

Vinter's tester loads crash images, runs recovery, and extracts a *semantic state* from the file system. It then automatically detects two crash consistency properties, single final state and atomicity, based on the number of unique semantic states. Unlike Witcher's output equivalence checking [45], Vinter does not require oracles to discover states for comparison. Its crash image generation in combination with a separate state extraction program guarantees that the initial and final states are among the set of states for an operation. In Section 8.6, we describe this approach in more detail and extend it with additional automatic reporting.

Vinter tests file systems with a set of 16 manually-written test cases, each covering a basic file system operation. These test cases include traced hypercalls to delimit the tested operation from setup code.

7.5.6 Chipmunk

Chipmunk [81] is a crash consistency testing tool specifically for file systems.

Chipmunk uses function-level instrumentation to trace PM accesses. It hooks into the file system's helper functions for writing data to PM using Kprobes [70] and Uprobes [40]. For each of these functions, the developer needs to write handlers that create appropriate trace entries.

Although this approach to tracing does not require any modifications to the tested file systems, it requires in-depth knowledge of the tested file system and considerable manual effort from the developer. As an example, the tracing code for NOVA has around 1400 lines of code [82, `chipmunk/loggers/logger-nova.c`]. It traces four functions (non-temporal `memcpy` and `memset`, helper functions for cache flushes and memory fences). The tracing function for `memcpy` encodes that the underlying Linux implementation does not properly flush some unaligned accesses, which is a detail that the NOVA developers originally missed [68, §5.3.1]. In contrast, tracing with binary translation as in Vinter and **Suvi** does not require such knowledge.

Chipmunk's crash image generation is similar to previous works. It replays the trace, reconstructs PM contents, and creates images with a subset of in-flight modifications. Chipmunk reduces the search space by coalescing multiple writes originating from the same traced function (e.g., a call to `memcpy`). Otherwise, it caps the number of writes considered for subsets.

To test crash images, Chipmunk mounts the file system and compares its contents to an oracle. The oracle runs the original test case and records the file system contents at each system call. If the file system contents of the crash image do not match the contents given by the oracle, Chipmunk reports a bug.

Chipmunk features automatic workload generation using ACE [93] and Syzkaller [7]. With ACE, the authors generated all workloads consisting of up to two file system calls, and a subset of workloads with three calls. Syzkaller automatically generates random system call sequences guided by code coverage information. For Chipmunk, the authors restricted Syzkaller to generate sequences of file system operations.

Compared to Vinter and **Suvi**, Chipmunk achieves higher coverage of test cases with its automatic workload generation. Vinter and **Suvi** do not require a separate oracle for comparing file system states. Instead, they rely on hypercalls in the manually-written test case that mark the start and end of the tested operations.

7.5.7 Mumak

Mumak [46] is a crash consistency testing tool focused on fast black-box analysis.

Mumak's tracing uses binary translation of userspace software with Intel Pin [88]. It traces PM accesses, flushes, and fences. For each failure point (flushes and fences), Mumak also records a stack trace and inserts it into a *failure point tree*. The failure point tree represents all code paths that lead to failure points (the leaves of the tree).

When replaying the trace, Mumak only generates a crash image if the corresponding leaf in the failure point tree has not been visited yet. It therefore deduplicates failure points by stack trace. The crash images always contain all modifications. Mumak does not generate images with subsets. However, it inserts failure points at cache flushes in addition to fences.

Compared to most previous approaches to crash image generation, Mumak therefore significantly reduces the required effort: It generates crash images at fewer failure points due to deduplication, and it generates exactly one crash image per failure point.

Mumak tests crash images by running a recovery program, reporting a bug if the recovery fails. It does not consider the image's semantic state.

Since Mumak generates and tests fewer images with its testing pipeline, there is a high chance of missing bugs. For this reason, Mumak's authors propose trace analysis to detect patterns of PM misuse. However, in contrast to the testing pipeline, trace analysis cannot confirm that a detected pattern actually is a bug in the tested application.

In Section 8.4.3, we adopt Mumak's approach for **Suvi**, allowing fast crash consistency testing of PM file systems.

Chapter 8

Suvi: Crash Consistency Testing for PM File Systems

In this chapter, we introduce **Suvi**, our approach to crash consistency testing for PM file systems. As discussed in Chapter 7, there are different types of PM software, which may exhibit different types of crash consistency bugs. **Suvi** features a testing pipeline with replaceable yet interoperable components. Table 8.1 shows an overview of these components. We examine each of these components in the following sections.

An important goal of **Suvi** is the ability to test unmodified software. Any modification for testing, such as inserting trace points or compiling a special configuration of kernel components for userspace testing, might change the analysis (see Section 7.4). **Suvi**'s tracer enables black-box testing by observing the execution of a file system in a virtual machine. The file system's underlying storage medium dictates the choice of tracer. For PM tracing, **Suvi** needs to insert hooks into all instructions that access PM or affect instruction ordering. The PM tracer is thus based on binary translation. In contrast, the hypervisor provides NVMe as an emulated device. The NVMe hooks are therefore placed in the hypervisor and work with hardware-accelerated VMs as well.

The crash image generator combines stores to PM to form crash images. Crash images are possible contents of the PM after a crash, according to the platform's crash consistency semantics. The primary challenge of the crash image generator is *combinatorial explosion*. With a large number of pending store operations, it is not feasible to generate all possible crash images. **Suvi**'s two crash image generator

Tracer	Crash Image Generator	Tester
• PM only (binary translation) • NVMe only (hardware-accelerated) • NVMe and PM (binary translation)	**Suvi-Fast** Fast crash image generation (finds logic bugs) **Suvi-Reads** Heuristic-based crash image generation (finds ordering bugs)	• Semantic state extraction and analysis • Trace analysis

Table 8.1: **Suvi**'s pipeline options

algorithms aim to reduce the search space without missing images that exhibit bugs. The first algorithm, **Suvi-Fast**, does not combine subsets of pending stores. Instead, it generates images with stores strictly in program order, which is sufficient for finding logic bugs. The second algorithm, **Suvi-Reads**, aims to find ordering bugs and thus must consider subsets of pending stores. It reduces the search space with a heuristic tailored to common PM access patterns, such as journaling.

Finally, the tester examines crash images to detect crash consistency violations. It loads each crash image in a virtual machine, then runs recovery code and a state-extraction program that prints a serialized representation of the state on PM. By examining unique states, the tester can automatically detect intermediate states that must not appear for atomic operations.

Suvi also implements a simple trace analysis. It can detect patterns in the trace that indicate certain bugs, including performance bugs and missing flushes.

8.1 Tracer

The first stage of **Suvi**'s crash consistency analysis is to observe a workload's interaction with persistent storage. The resulting artifact is a trace file that later stages use for their analysis.

We identify a number of requirements for the tracer:

Black-box testing. We want to test unmodified applications. In particular, we do not want to require a manual definition of tracing points from the application developer. See Section 7.4 for a discussion of tracing approaches.

Kernel-mode software. **Suvi** is an approach for testing file systems, which are often implemented as a kernel component. The tracer, therefore, needs to support tracing code running in kernel mode.

Performance. The tracer should not unnecessarily slow down the workload, as a faster tracer allows testing more test cases.

Debugging metadata. Besides the storage-access trace, we want to capture additional information that helps later analysis stages or the developer in understanding found bugs.

We also identify non-goals:

Multicore testing. **Suvi** is not a tool for detecting race conditions and other multicore correctness issues. We assume that the tested software is already free from race conditions as a prerequisite for crash consistency.

Figure 8.1 shows a high-level overview of the tracer design. The tracer plugin, running in the hypervisor, receives PM and NVMe events and serializes them into a trace file. Collection of PM and NVMe events is independent of each other and can be enabled separately. In the following sections, we have a detailed look at these tracing approaches.

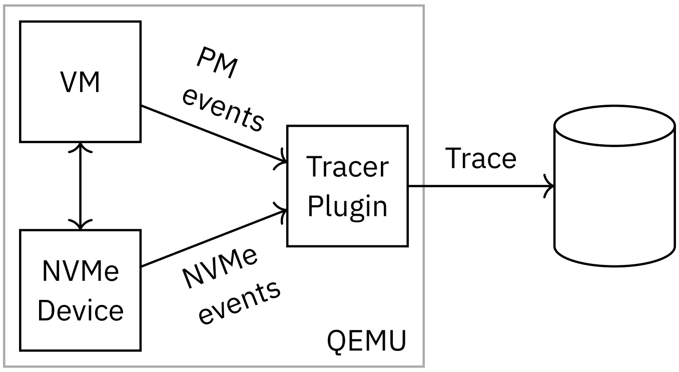


Figure 8.1: Overview of the tracer. Hooks in the VM's translated code capture PM events, and hooks in the virtual NVMe device capture NVMe events. The tracer plugin receives these events and serializes them into a trace file.

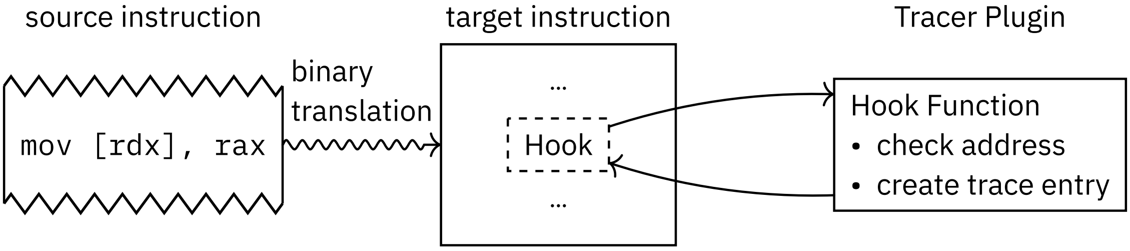


Figure 8.2: The PM tracer works by inserting hooks into the translated code. In this example, the emulator translates a `mov` instruction with a hook that calls into the tracer plugin.

8.1.1 PM Tracing

The CPU accesses PM directly with load and store instructions. Additional ordering instructions, such as cache line flushes and memory fences, ensure that stores reach PM in a particular order. Correct ordering is critical for achieving crash consistency.

Suvi's PM tracer uses virtual machines with dynamic binary translation to trace these instructions. With binary translation, **Suvi** can perform black-box analysis of unmodified applications.

Figure 8.2 shows an overview of the tracer. With dynamic binary translation, an emulator translates instructions from a source ISA to a target ISA. This translation provides the possibility to insert additional code and to hook relevant instructions.

The hooked instructions fall into three categories, as shown in Table 8.2: write, cache flush, and memory fence. The hook functions collect parameters from the hooked instruction and create a trace entry if the instruction is relevant. Write and cache flush events are relevant when the memory mapped at their associated address is PM. We do not trace fences if there are no other traced events since the last fence.

For **write** events, the trace contains the physical address of the store instruction, the number of written bytes, the written bytes, and whether the instruction was non-temporal. On x86, the written data has a size of 1 to 64 bytes, depending on the instruction and the register it references. The exact instruction does not matter for the analysis. We only need to identify non-temporal instructions that do not write to the CPU caches. The tracer also optionally supports **read** events with similar properties. One of **Suvi**'s crash image generation algorithms, **Suvi-Reads**, uses read events for its heuristic (see Section 8.4.1). Otherwise, the tracer does not record reads.

The **flush** events are more complex since the underlying x86 instructions differ slightly in their semantics.[27] For this reason, we store the assembler mnemonic in addition to the physical address of the flush. The crash image generator can then use this information to simulate the behavior of the respective instruction.

Finally, **fence** events do not carry any additional information for the crash image generator. The tracer still records the mnemonic to aid users in understanding the trace.

Metadata

For all event types, the tracer records metadata. Metadata is not required for detection of crash consistency bugs but is useful for users of **Suvi** debugging crash consistency violations. **Suvi** records the following metadata for PM traces:

Program counter. The program counter is stored in the x86 register `rip`. Since reading a VM register is cheap, the tracer includes the program counter for every PM trace entry.

Stacktraces. For all event types, the tracer can optionally record a stacktrace by following frame pointers. The frame pointers form a chain on the stack with the initial pointer stored in the `rbp` register on x86 [87]. The tracer can therefore follow the frame pointers without any knowledge about the running application, in contrast to more advanced stack unwinding techniques such as DWARF [42]. Compilers often omit frame pointers to improve performance [48]. However, they are always enabled in the Linux kernel, making them available for our file system analysis. Since unwinding the stack with frame pointers requires multiple VM memory accesses, recording stacktraces slows down the tracer. Users can therefore enable stacktraces if needed.

Besides their use for debugging, stacktraces are required for **Suvi-Fast**, one of **Suvi**'s crash image generation algorithms (see Section 8.4.3).

8.1.2 NVMe Tracing

NVMe is an asynchronous, command-based protocol. The operating system writes commands into a ring buffer and then waits until the NVMe device writes a corresponding completion into another ring buffer. An NVMe tracer thus needs to capture these commands and completions. We describe the relevant parts of NVMe in Section 2.4.2.

Since **Suvi** runs its test cases in a virtual machine, the operating system interacts with a virtual NVMe device that is part of the hypervisor. As discussed in Section 2.4.2, we configure a "worst-case" NVMe device with a volatile write cache, a block size of 512 bytes, and a maximum power-fail atomicity of one block. **Suvi**'s NVMe tracer uses hooks in this virtual NVMe device to assemble trace entries (Figure 8.3).

[27] `cflushopt` and `clwb` are weakly ordered, whereas `clflush` is strongly ordered. See Section 2.4.1.

Type	Parameters and Filters	Example x86 Instructions
write **(read)**	• Physical address: u64 *filter*: within PM area • Size: 2^0 to 2^6 bytes • Data: `[u8]` • Non-Temporal: `bool`	`mov` (temporal) `movnt` (non-temporal)

cache flush	• Physical address: u64
	filter: within PM area
	• Mnemonic: (`clflush`/`clflushopt`/`clwb`)
		`clflush`
		`clflushopt`
		`clwb`

memory fence	• *filter*: have writes since last fence	`sfence` `mfence`

Table 8.2: Overview of PM trace entry types.

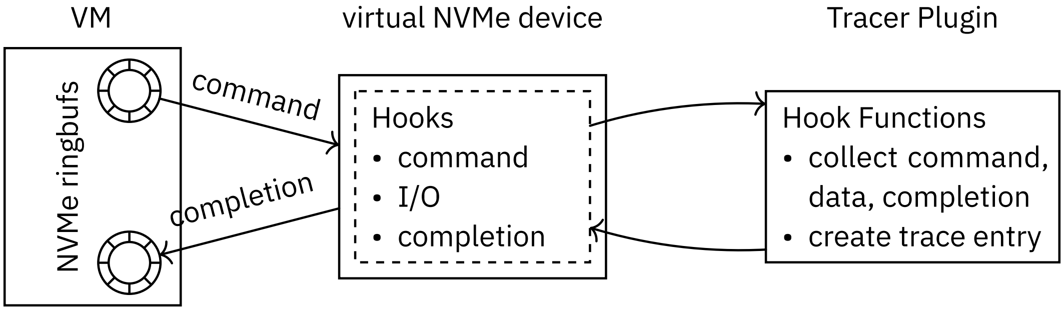


Figure 8.3: The NVMe tracer uses hooks in the request handling of a virtual NVMe device. These hooks call into the tracer plugin, which assembles a trace entry for each request.

The NVMe trace has three events named read, write, and flush, listed in Table 8.3. They correspond to NVMe commands with the same names. As with the PM trace, only write and flush commands are relevant for crash consistency testing.

A **write** event includes the offset of the write on the SSD, the size of the write, and the written data. The offset is traced as a byte offset and is always a multiple of the SSD block size.

Similar to the PM tracer, the NVMe tracer can optionally trace **read** events with the same properties.

The **flush** events do not require any additional data, since NVMe flush commands apply to the whole SSD and are not scoped to specific addresses.

Type	Parameters and Filters
write **read**	• Offset in bytes: u64 • Size in bytes: u64 • Data: `[u8]`
flush	*no parameters*

Table 8.3: Overview of NVMe trace entry types.

The NVMe specification includes more commands that modify data, for example *Write Zeroes*. As discussed in Section 2.4.2, Linux file systems do not use these commands. **Suvi** thus does not trace them. However, our approach permits tracing additional NVMe commands if necessary.

8.1.3 Hypercalls

Suvi treats the test VM as a black box and does not know which programs or operations are running in the VM. However, tests need to communicate some information to indicate the current state of the test (i.e., whether it is running or an error has occurred) and to distinguish different test phases. The tracer provides a hypercall mechanism for this purpose.

Suvi supports four kinds of hypercalls:

Start. Indicates the start of the test case. To reduce tracing overhead during VM startup, the tracer starts tracing events only after this hypercall.

Success or Fail. Indicates that the test case finished successfully or encountered an error. The tracer expects one of these hypercalls as a signal to stop the VM.

Checkpoint <ID>. A test case can use checkpoints to separate different operations within the test. The checkpoints are numbered. The checkpoint hypercall carries the checkpoint identifier as an additional argument.

Listing 8.1 shows an example of how these hypercalls are used within a test case.

8.1.4 Discussion

Suvi's tracer enables black-box testing of unmodified PM file systems, including userspace and kernelspace software. The primary challenge is performance, since binary translation introduces a large overhead compared to native execution. In Section 9.1, we discuss how we ensure that the tracing routines add as little overhead as possible on top of the binary translation.

The NVMe tracer does not require binary translation, since its hooks are located in the hypervisor's virtual NVMe device. However, this property introduces a different limitation. NVMe is an asynchronous protocol that allows implementations freedom

```
hypercall start
try:
  # Set up test environment by initializing and mounting the file system.
  mkfs /dev/pmem0
  mount /dev/pmem0 /mnt
  hypercall checkpoint 1
  test_operation_1 /mnt # First operation between checkpoints 1 and 2
  hypercall checkpoint 2
  test_operation_2 /mnt # Second operation between checkpoints 2 and 3
  hypercall checkpoint 3
  # End of test: indicate success.
  hypercall success
catch:
  # If any previous operation failed, communicate test failure.
  hypercall fail
```

Listing 8.1: Pseudo-code example of a test case using hypercalls to communicate its state with **Suvi**.

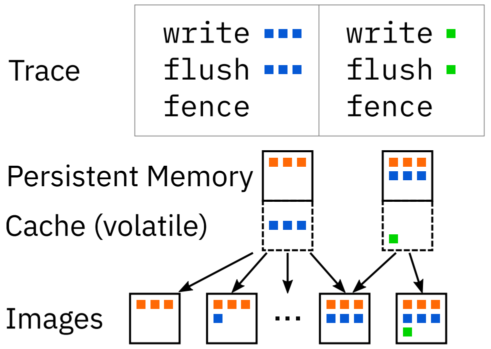


Figure 8.4: The crash image generator replays the trace and keeps track of volatile data in caches and persisted data in PM. At certain points, it generates crash images by combining PM contents with volatile data. A colored square represents a unit of data that can be written atomically.

in how they process commands. In particular, an SSD may delay and reorder commands. Since our approach hooks a virtual NVMe device, it does not explore this freedom. QEMU's NVMe device, as used by **Suvi**'s implementation, processes commands synchronously and in order. Therefore, **Suvi** cannot detect bugs that stem from improper sequencing of NVMe commands and completions.

8.2 Crash Image Generator

Suvi's crash image generator takes a trace file and replays its entries while keeping track of the contents of non-volatile memory (both PM and NVMe) as well as caches. Its output is *crash images*, which are potential contents of the non-volatile memory in the event of a crash at a certain point in the trace.

Figure 8.4 shows an example. The trace starts with a write of three data units (blue squares, second row), followed by flush and fence operations. At this point, **Suvi** treats these writes as volatile. Due to out-of-order execution and spurious cache flushes, a subset of the writes according to the system's crash consistency semantics may reach PM in the event of a crash. The resulting crash images are thus a combination of PM contents (orange squares, first row) and a subset of the new writes. After the fence, all flushed writes move to PM and are safe in the event of a crash.

In the following sections, we first describe **Suvi**'s underlying model for generating PM crash images. This model describes how to replay the trace and defines the search space for crash images. We then describe **Suvi**'s algorithms for efficient crash image generation. **Suvi-Reads** efficiently explores fine-granular reorderings of writes and allows detection of misuse of PM primitives. It works by analyzing read accesses of a recovery procedure. **Suvi-Fast** allows a fast analysis for logic bugs by generating fewer crash images. For cross-media file systems, we then describe how to generate NVMe crash images and how to combine these with PM crash images.

8.2.1 Model Goals

We set the following goals for the PM and NVMe simulation:

Accuracy. The model should be as accurate as possible with respect to the specified hardware behavior. In particular, it should not generate crash images that are impossible according to the specification.

Fast trace processing. Individual trace entries should be efficient to process. For example, the initialization of a NOVA file system of size 5 MiB results in a trace containing around 300 000 entries.

Fast crash image generation. The internal state of the crash image generator should allow fast combination of pending writes into crash images.

8.2.2 Crash Image Metadata

The tester, **Suvi**'s final testing pipeline stage, must be able to associate crash images with a specific failure point. This is necessary, for instance, to analyze crash states for each logical operation in the test (indicated by checkpoints in the trace). **Suvi** therefore stores crash metadata with every crash image.

The crash metadata contains the following information:

- A cryptographic hash of the crash image data, uniquely identifying the image.
- Results from **Suvi-Reads** (Section 8.4.1): Which cache lines in the image were modified and read by the recovery?
- A list of failure points at which this crash image was generated.

A failure point is identified with the following information:

- The location of the failure point in the trace.
- The current checkpoint identifier (Section 8.1.3).
- Information on how the crash image generator created the image, which is one of the following:
  - **Nothing.** No pending writes were included.
  - **Everything.** All pending writes were included.
  - **Subset.** A strict subset of the pending writes was included. This case includes all information on how the subset was chosen. For PM images, this includes the subset of cache lines, the write limit for ordered stores, and whether NT stores were applied.

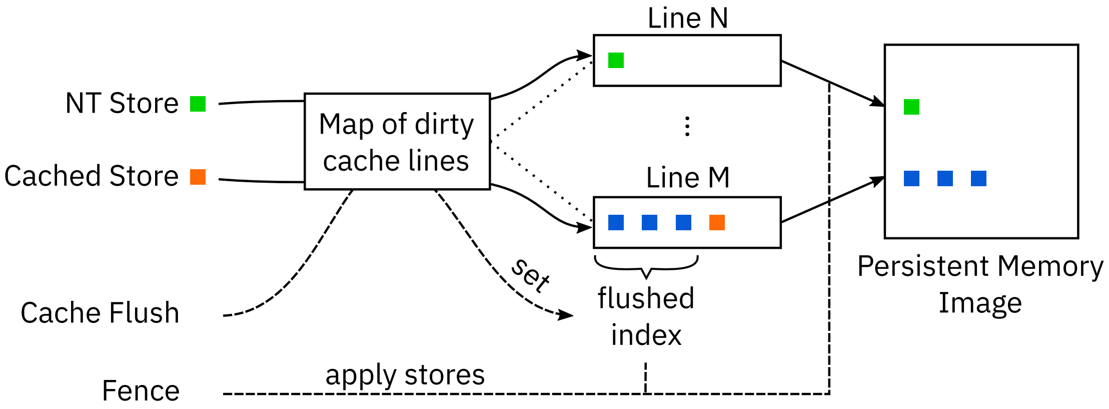


Figure 8.5: **Suvi**'s simulation of the PM write path. Stores are collected per cache line. A flush sets an index in that cache line to mark all previous stores as flushed. A fence applies all flushed or non-temporal stores to the PM image.

8.3 PM Crash Image Model

Suvi needs to keep track of all stores that are not yet fully persisted and that may be lost in the event of a crash. The instruction set architecture provides certain guarantees about the ordering and behavior of such stores. We introduced these semantics in Section 2.4.1. We now translate these semantics into a model that satisfies the goals given above. We first introduce our model for systems with volatile caches, then extend it to support systems with persistent caches (eADR).

8.3.1 Trace Replay

Figure 8.5 shows the main components and their interaction with trace entries. Persistent memory is simulated with an in-memory image (i.e., a byte array). All data in this image is fully persisted and survives crashes.

Store commands in the trace are not directly applied to the PM image, since they might be lost after a crash. **Suvi** places them into a map of dirty cache lines. Each dirty cache line maintains a list of stores and a *flushed index* into this list to track cache flushes.

A cache flush instruction (`clwb` or `clflushopt`) marks all preceding stores to a cache line as flushed by setting the flushed index to the length of the list of stores. Note that **Suvi**'s model does not support the older `clflush` instruction, which has additional ordering properties compared to `clflushopt` (see Section 2.4.1). These ordering properties make `clflush` execute slower than `clflushopt`, which is why PM software generally does not use this instruction.

Finally, a memory fence (`sfence`) clears all flushed and non-temporal stores from the dirty cache lines and applies them to the PM image. If the line contained only NT stores or if all cached stores were flushed, it is then removed from the map of dirty cache lines. Otherwise, cached stores without a flush remain, and the flushed index for this line is reset to zero.

The model as described so far can accurately replay the trace while keeping track of dirty cache lines and PM contents. It is sufficient for basic crash image generation and for detecting some types of bugs. For example, the PM image is a valid crash image after processing each fence. Any dirty cache lines remaining after replay indicate that the application is missing cache flush instructions. We now augment additional information to this model to enable generation of valid crash images at any position in the trace.

8.3.2 Failure Points

A failure point is a position in the trace for which **Suvi** generates crash images. These crash images simulate a failure occurring at that position in the trace. Since crashes can occur at any moment, we want to achieve a full coverage of failure points. However, generating crash images after every trace entry would produce many redundant images. Due to instruction reordering and CPU caches, the amount of volatile data increases with every store command in the trace. Therefore, the set of possible crash images also grows with every store command.

Memory fences are ordering points where the CPU halts execution until previous memory accesses have completed. The amount of volatile data can thus shrink at memory fences. To avoid missing crash images, it is necessary to generate images before every memory fence.

With these observations in mind, we design **Suvi** to generate crash images only at memory fences. By keeping track of dependencies between store entries, we ensure that the crash images generated at the fence include all crash images that would be generated for failure points since the previous fence. In the following, we describe these dependencies as well as our data model for working with them.

For the detection of the single final state property (see Section 8.6.2), the tester needs to determine the application state at the beginning and end of each operation in the test. **Suvi**'s heuristics can decide to skip crash image generation at regular failure points (see Section 8.4). We therefore introduce additional failure points at each checkpoint that are never skipped.

8.3.3 Global Store Ordering

Regular stores on x86 are strongly ordered, even if they target different cache lines. Since **Suvi**'s model collects stores per cache line, it needs to retain information about global store order separately. **Suvi** therefore keeps a global counter of all regular stores and includes its current value for each store. For crash image generation, **Suvi** can collect stores from the dirty cache lines and restore the global ordering with the store counters.

In addition to counter values in each store, **Suvi** also keeps track of the store counter value at the last memory fence instruction. The fence counter ensures that **Suvi** always includes unflushed but fenced stores in later crash images.

As an alternative design, we might consider keeping a global list of stores instead of lists per cache line. Such a global list would preserve store ordering without additional effort. However, this design would make cache-line-scoped operations, such as cache flushes, more expensive to process, since they would need to scan the global list to find stores to the same cache line. We therefore decided to collect stores per cache line.

8.3.4 Mixed Non-Temporal and Cached Stores

In contrast to regular stores to caches, non-temporal stores are weakly ordered and thus do not follow a global order. However, as discussed in Section 2.4.1, non-temporal stores are ordered with other stores going to the same cache line. A cache line receiving both types of stores thus restricts the global order of non-temporal stores to that line.

Suvi models a sequence of stores with three kinds of elements.

Cached Store. A single regular store to the cache. Carries a write counter value for global ordering.

NT Stores. One or more non-temporal stores, coalesced into a single entry. These stores were not preceded by cached stores since the last memory fence. Their earliest appearance relative to stores to other cache lines is thus not restricted.

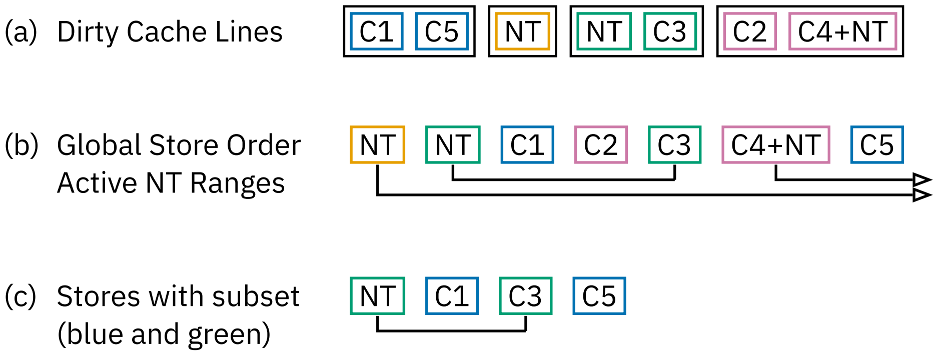


Figure 8.6: Example of how cached and non-temporal stores are handled when generating crash images. The cached stores with write counters (C1-C5) and non-temporal stores (NT) from four dirty cache lines (a) are collected into a sorted list (b). While walking the list, the crash image generator keeps track of active NT stores to each cache line. This process is repeated for subsets of the dirty cache lines, as in (c).

However, their latest appearance may be restricted by following cached stores to the same cache line.

Cached then NT. The final entry type is a combination of the other two: a single cached store followed by one or more non-temporal stores. This entry type restricts the earliest appearance of the non-temporal stores. They may appear only after the preceding cached store.

Although a sequence of cached and non-temporal stores to a single cache line could be tracked as separate items, **Suvi** requires the combined item to establish a global ordering for these stores. We describe this process first for volatile caches and then extend it for persistent caches.

8.3.5 Crash Images with Volatile Caches

The goal of crash image generation is to generate images that include a valid subset of stores to dirty cache lines. The algorithm therefore needs to take all ordering constraints into account.

We illustrate the process with an example shown in Figure 8.6. In the top row (a), it shows four dirty cache lines that have received five strongly ordered cached stores (C1 to C5) and three weakly ordered NT stores.

There are two ways to choose a subset of stores. First, by selecting a subset of dirty cache lines. This method simulates cache eviction. Only lines evicted from caches (e.g., due to a conflict) end up on PM. Similarly, a line receiving non-temporal stores might be lost in the write-combining buffer. For example, in Figure 8.6, we might decide to include only stores from the blue and green cache lines (C1, C5, green NT, C3).

Second, we can form a valid store subset by choosing a prefix of stores according to their global order, as tracked by the store counter values. It is important to take the global order into account instead of deciding on prefixes per cache line. For example, in Figure 8.6, if we decide to include store C5 from the first cache line, we must also include stores C2 to C4 and the green NT store from the other cache lines.

Suvi's crash image generation combines these two ways of forming subsets. We describe the selection of a store prefix here, and discuss forming subsets of dirty cache lines in Section 8.4.1. As an optimization, **Suvi** only performs the store prefix selection once, but simultaneously applies stores to multiple PM images according to the selected cache line subsets. We walk through an example with cache line subsets below.

Suvi collects all current stores in a list and sorts the list by the stores' store counters. Weakly ordered non-temporal stores do not carry a write counter and sort before all regular stores, since they could appear before them. Combined "cached then NT" stores are sorted according to the cached store's write counter. Row (b) of Figure 8.6 shows the resulting sorted list for all cache lines, and row (c) for the blue and green cache lines.

Suvi then walks through this list. If it encounters a non-temporal store (either standalone or combined), it stores it in a map (cache line number → NT store). If it encounters a regular store, it checks the map for a previous NT store to the same line. If there is one, **Suvi** removes that NT store from the map and applies it to the PM images according to the cache line subsets. Finally, **Suvi** always applies the cached store immediately to the PM images, since its global order is fixed.

With this strategy, the map of NT stores keeps track of the non-temporal stores that are active (i.e., could be part of a crash image). The lines and arrows in Figure 8.6 indicate the ranges in which the non-temporal stores are part of the map.

After each regular store that has not been fenced yet (i.e., its store counter value is higher than that of the previous fence), **Suvi** can emit crash images. The first crash image is the current state of the PM image. Second, **Suvi** applies all active NT stores from the map to a clone of the PM image.[28] This forms the second crash image. If there are only non-temporal stores without regular stores, **Suvi** generates crash images after collecting all of them.

Going back to the earlier example, Table 8.4 shows how the list of stores is processed. Consider the row with store C3. When looking up its cache line in the map of NT stores, **Suvi** finds the green NT store. **Suvi** applies that store (and C3) to the crash image and removes it from the map. The stores C1, C2, green NT, and C3 are then applied to the PM image and are thus part of the first crash image. The map of non-temporal stores contains only the yellow NT store. The second crash image therefore also includes that store.

The need for combined "cached then NT" items follows from this algorithm. These items must be sorted correctly with the preceding regular store. A separate NT item with a store counter would not work, as crash images generated at the previous store already need to include the NT store.

Note that Table 8.4 does not show the selection of cache line subsets, which occurs independently of determining the store prefixes. Given a set of stores (one row in the table), **Suvi** yields multiple crash images in which only the stores to a subset of the cache lines are included.

[28] We discuss partial application of NT stores below.

#	Stores	NT Map	Store Prefixes	
1	NT	NT	-	
2	NT	NT NT	-	NT NT
3	C1	NT NT	C1	... NT NT
4	C2	NT NT	C1 C2	... NT NT
5	C3	NT	C1 C2 NT C3	... NT
6	C4+NT	NT +NT	C1 C2 NT C3 C4+	... NT +NT
7	C5	NT +NT	C1 C2 NT C3 C4+ C5	... NT +NT

Table 8.4: Visualization of the crash image generation algorithm for the stores from Figure 8.6 (b). The algorithm processes the stores sequentially (top to bottom) and generates store prefixes with NT stores (right) and without NT stores (left).

#	Stores	Subset without NT	Subset with NT
1	NT	-	-
2	NT		NT
3	C1	C1	C1 NT
4	C2		
5	C3	C1 NT C3	-
6	C4+NT		
7	C5	C1 NT C3 C5	

Table 8.5: Visualization of crash images generated for a subset with the blue and green cache lines. At every step in Table 8.4, blue and green stores are applied to two subset PM images with and without trailing NT stores.

In Table 8.5, we show an example for one subset consisting of the blue and green cache lines. For every store prefix in Table 8.4, the stores to the blue cache line (C1 and C5) and the green cache line (NT and C3) are applied to PM images and yielded as crash images. This process occurs simultaneously for multiple cache line subsets.

8.3.6 Crash Images with Persistent Caches (eADR)

We now extend the algorithm above for crash images on systems with persistent caches. Recall that for volatile caches, **Suvi** chooses both a subset of dirty cache lines and a prefix of stores according to their global order. With persistent caches, all dirty cache lines are always present in all crash images. Consequently, **Suvi** does not choose subsets for applying regular stores and works with all dirty cache lines. However, the CPU collects non-temporal stores in its write-combining buffers. Even with eADR, these buffers are not persistent (see Section 2.4.1). Choosing subsets is thus still necessary for non-temporal stores.

Action	Volatile Caches	Persistent Caches
Subsets for regular stores	Yes, based on lines from heuristic	No, use stores from all dirty lines with regular stores
Emit crash images	At regular store, if it was not fenced before	At regular store, if its line was selected by heuristic
Include NT stores	All collected NT stores	For each subset of dirty lines with NT stores

Table 8.6: Differences in the crash image generation algorithm for volatile and persistent caches.

On the other hand, subsetting by selecting a prefix of ordered stores is still valid with persistent caches. This is therefore the primary way of generating crash images for systems with persistent caches.

Table 8.6 shows a summary of adjustments for persistent caches. The algorithm does not use the subsets indicated by the heuristic and collects stores from all dirty cache lines that have at least one regular store. It then processes the sorted list of stores identically, applying cached stores and tracking active non-temporal stores in a map. When encountering a regular store, **Suvi** emits a crash image only if the store was to a line indicated by the heuristic (see Section 8.4.1). **Suvi** thus avoids creating crash images that are unlikely to be interesting.

Suvi then generates subsets of all lines with active non-temporal stores. For each subset of lines, it applies the non-temporal stores to those lines on a clone of the PM image at that point and emits the resulting image.

Stores	NT Map	Crash Images with NT Subsets
NT	NT	-
NT	NT NT	NT NT NT NT
C1	NT NT	C1 C1 NT C1 NT C1 NT NT
⋮		

Table 8.7: Excerpt of crash image generation with persistent caches for the stores from Figure 8.6 (b). Cache line subsets are generated only for non-temporal stores.

	Crash Images with Partial NT Stores					
First Cache Line	NT1	NT12	NT123	NT1	NT12	NT123
Second Cache Line	NT1	NT1	NT1	NT12	NT12	NT12

Table 8.8: Crash images with partial application of NT stores for two cache lines. The first cache line contains three NT stores, and the second contains two NT stores. The NT stores are ordered per cache line but may be interleaved between cache lines.

8.3.7 Partial Application of Non-Temporal Stores

The algorithm described so far only implements partial application of temporal stores. Since temporal stores have a global order across all cache lines, generating crash images with a subset of these stores is feasible.

In contrast, there is no such ordering between non-temporal stores to different cache lines. Consequently, arbitrary interleavings of prefixes of the stores going to individual cache lines are possible, as illustrated in Table 8.8. This makes crash image generation with partial non-temporal stores more expensive than with only partial temporal stores.

Suvi avoids generating an excessive number of crash images by always including all pending non-temporal stores for a given subset of cache lines. After analysis has finished, a heuristic can detect failure points where pending non-temporal stores had a direct effect on the discovered states. **Suvi** can then repeat the crash image generation and explore crash images with a subset of non-temporal stores at these failure points. We describe this heuristic in Section 8.4.2.

8.4 PM Crash Image Heuristics

In the previous section, we described how to replay the trace and how to generate valid crash images from the replay state. The remaining challenge is that generating all possible crash images is only feasible if the number of dirty cache lines is small. For N dirty cache lines, there are 2^N cache line subsets. We therefore need a way to reduce the set of dirty cache lines before passing it to the crash image generator. **Suvi** implements three heuristics (**Suvi-Reads**, **Suvi-NT**, and **Suvi-Fast**) for this purpose, which are described in the following sections.

Suvi-Reads allows efficient exploration of the possible crash states by filtering the set of dirty cache lines based on the behavior of the program's recovery procedure. **Suvi-Reads** therefore can discover misuse of crash consistency primitives, such as missing fence instructions.

Suvi-NT addresses the problem that unconstrained non-temporal stores pose to the crash image generation algorithm. By detecting failure points where non-temporal stores have an immediate effect on the resulting semantic state, **Suvi-NT** allows exploration of more crash images that have a high likelihood of exposing bugs.

Finally, **Suvi-Fast** focuses on rapid detection of logic bugs. Logic bugs are defects that appear even if all stores are applied according to program order. There is therefore no need to generate crash images based on subsets of cache lines. **Suvi-Fast** therefore accelerates crash image generation and enables the analysis of more complex test cases.

8.4.1 Suvi-Reads: Efficient Exploration of Crash States

The goal of **Suvi-Reads** is to reduce the set of dirty cache lines passed to the crash image generation algorithm. It achieves this by leveraging program behavior commonly found in PM applications.

The CPU architecture only guarantees crash atomicity of writes up to 8 bytes in size (see Section 2.4.1). Applications need to use patterns such as journaling to persist

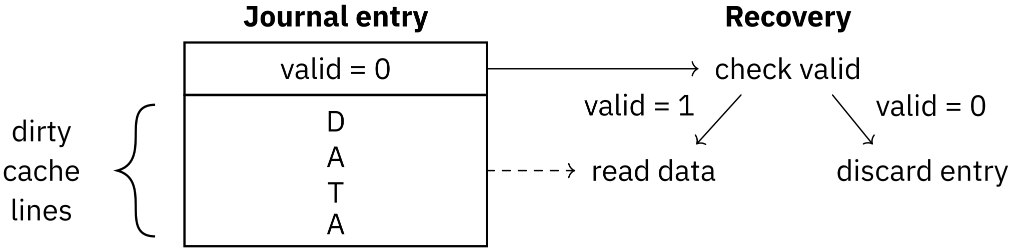


Figure 8.7: Situation when generating crash images after a journal entry's data field has been written. A recovery procedure will read the journal entry's data only if it is marked as valid. Consequently, subsets of dirty cache lines in the data area will never result in new crash states.

larger amounts of data atomically. Figure 8.7 shows a situation in the middle of a journaling operation. The journal entry's data field has been written, but the entry has not yet been marked valid. At that point, **Suvi** would observe a large number of dirty cache lines, depending on the size of the journal entry. Generating all possible subsets of these cache lines is likely infeasible.

After a crash and restart of an application employing journaling, a recovery procedure inspects each journal entry. Invalid entries are discarded, since their data field might be incomplete. Only if a journal entry is marked valid is its data field read and applied to the application state.

Therefore, there is no point in generating cache line subsets of a journal entry's data field. Recovery always discards the entry (as long as the valid field is set separately) and never reads the data. **Suvi-Reads** thus needs to detect this recovery behavior to avoid generating crash images with invalid journaling data.

Vinter-Heuristic: Observing Recovery Behavior

Suvi extends the heuristic introduced for Vinter [68] (called Vinter-Heuristic in the following). We first describe Vinter-Heuristic in its original form, then extend it to **Suvi-Reads** by addressing a limitation that could lead to false negatives.

Vinter-Heuristic is based on the idea of *post-failure tracing*, as originally proposed for XFDetector [85]. In contrast to XFDetector, Vinter-Heuristic uses the tracing results only as input to its crash image generation and does not directly derive bugs from them. We describe XFDetector in Section 7.5.3.

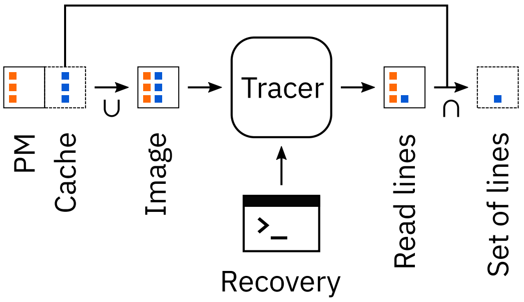


Figure 8.8: Vinter-Heuristic loads a fully persisted image (PM ∪ dirty cache lines) into the tracer and executes a recovery procedure. Its output is the intersection of the dirty cache lines and the lines read by the recovery procedure.

Figure 8.8 shows an overview of the process. Vinter-Heuristic creates a fully persisted image by applying all dirty cache lines to a clone of the current PM state. It then loads the resulting image into the tracer. In contrast to regular pre-failure tracing, the tracer now traces read rather than write accesses. The tracer executes a recovery procedure on the image. The recovery procedure needs to load the image and perform crash recovery if necessary. Finally, it should access all reachable state contained in the image by running the program used for state extraction (see Section 8.6).

After tracing finishes, the access trace contains every load instruction to the PM image. Assuming that the state extraction program accesses the complete state contained in the PM image, we now know exactly which parts of the image are relevant for crash images. We could omit all parts of the image not referenced by the trace without affecting the extracted state. Note that **Suvi** still generates only crash images that could occur according to its PM model. It does not remove any data that cannot be lost in a real crash.

As discussed above, the goal of Vinter-Heuristic is to reduce the set of dirty cache lines to make subset generation feasible. Vinter-Heuristic thus calculates the set of cache lines that the recovery accessed in the trace. It then removes any lines that were not dirty. The result is therefore all dirty cache lines that were read by the recovery procedure.

Suvi then uses this set for crash image generation (see Section 8.2). With volatile caches, only subsets from these lines will be generated. Consequently, dirty cache lines that were not accessed by the recovery will not appear in the generated crash images.

With persistent caches, **Suvi** will similarly use the heuristic result for non-temporal stores. In contrast, all cached stores need to be included in the crash images. **Suvi** still uses the lines from the heuristic to decide whether to generate a crash image at a certain store. If a store accesses a cache line that is not read by the recovery, **Suvi** does not create a crash image for that store.

Limitations of Vinter-Heuristic

Vinter-Heuristic makes certain assumptions about the behavior of the PM application. If the application violates these assumptions, applying the heuristic may result in false negatives, since crash images exhibiting a bug might not be generated. The key property is that the post-failure tracing uses a fully persisted image as its basis. If the set of lines read by the recovery is smaller with the fully persisted image than with intermediate images, the heuristic might miss important lines.

Consider the journaling example from Figure 8.7. So far, we have considered only the process of writing new journal entries. With new entries, setting the valid bit extends the set of lines read by the recovery. If the data field is modified at the same time (i.e., without a flush and fence), appropriate crash images will be produced, including partially written data.

In contrast, removing a journal entry by setting its valid bit to 0 shrinks the set of lines read by recovery. If the application modifies the data field at the same time, for example, to remove sensitive data from the journal, there will be no crash images containing this data. Figure 8.9 shows this situation.

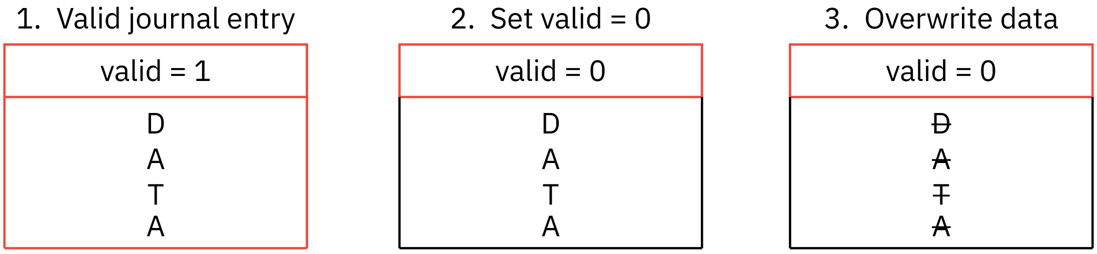


Figure 8.9: Steps performed when invalidating a journal entry. The fields read by the recovery process are marked in red. A missing fence between steps 2 and 3 could result in partially overwritten, yet still valid, crash images. However, Vinter-Heuristic will not include modifications to the data field in crash images and thus might miss bugs.

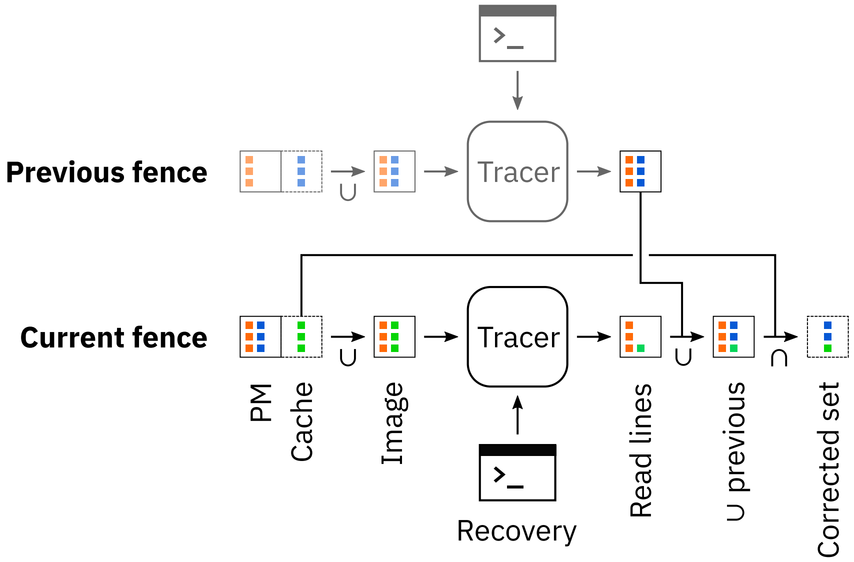


Figure 8.10: **Suvi-Reads** extends Vinter-Heuristic by detecting when the set of read lines shrinks. It keeps the previous heuristic tracing result and restores any lines that disappeared.

Suvi-Reads: Mitigating False Negatives

We now describe **Suvi-Reads**, our extension of Vinter-Heuristic that mitigates the limitation described above. The main insight is that situations as in Figure 8.9 can be detected by comparing the results from previous invocations of the heuristic.

Figure 8.10 shows an example of this process. At the current fence, the blue cache lines are overwritten. Some of these lines no longer appear in the recovery trace. **Suvi-Reads** corrects this by taking the previous recovery trace and adding its lines to the set of lines yielded from the current trace.

In the example, **Suvi-Reads** adds two additional cache lines (blue) to the set of read lines at the current fence. Since these cache lines are dirty, they end up in the corrected set and are used for crash image generation. Remember that the heuristic operates on sets of cache line numbers, regardless of the contents of these cache lines. The colors in the figures serve only to illustrate where the modifications occur.

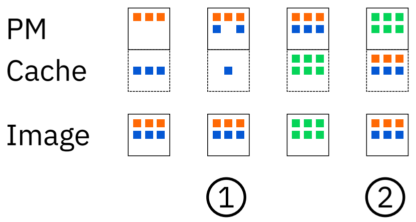


Figure 8.11: Sample replay (from left to right) that discovers identical fully persisted images in two different situations. Each column shows the state of PM and caches at a fence, as well as the resulting fully persisted image.

Reusing Heuristic Results

It is possible to encounter identical fully-persisted crash images at different fences. The fully-persisted crash image is the input for the the recovery procedure. We assume a deterministic recovery procedure that always yields the same output for a given input. **Suvi** can therefore reuse the previous result without rerunning the recovery.

However, it might still be necessary to evaluate subsets, since the crash images also depend on the previous PM contents. Figure 8.11 shows an example.

The first duplicate image at (1) is a typical situation for an application that is missing a cache flush. The contents written to one cache line stay in the cache. There were no other writes, so the fully persisted image remains the same. Subset generation could be skipped without missing any crash images.

This is not the case at the second duplicate image. At fence (2), the PM contents are different and would appear in crash images that include a subset of the cache contents. Skipping subset generation at this fence would therefore lead to missed crash images.

With these two cases in mind, we choose to have **Suvi** always generate subsets when encountering a duplicate fully persisted image. This strategy avoids missing crash images in situations such as at fence (2). It is possible to detect situations such as at fence (1), where the full replay state is a subset of the state at a previous fence. However, we find that since missing cache flush instructions are usually unintended, the amount of volatile data in the caches is small, making subset generation inexpensive.

This choice differs from Vinter. Vinter skips subset generation entirely when encountering a duplicate fully persisted image.

8.4.2 Suvi-NT: Detecting NT-Dependent Semantic States

Generating crash images with partial prefixes of non-temporal stores is expensive, as described in Section 8.3. Arbitrary interleavings of the individual stores to each cache line are possible (see Table 8.8), creating a large search space. **Suvi** therefore by default only generates crash images with either none or all NT stores to a cache line. This strategy keeps the search space manageable but may lead to missed crash states.

Suvi-NT is a heuristic that detects failure points where partial application of non-temporal stores may yield additional crash states. In contrast to **Suvi-Reads**, **Suvi-NT**

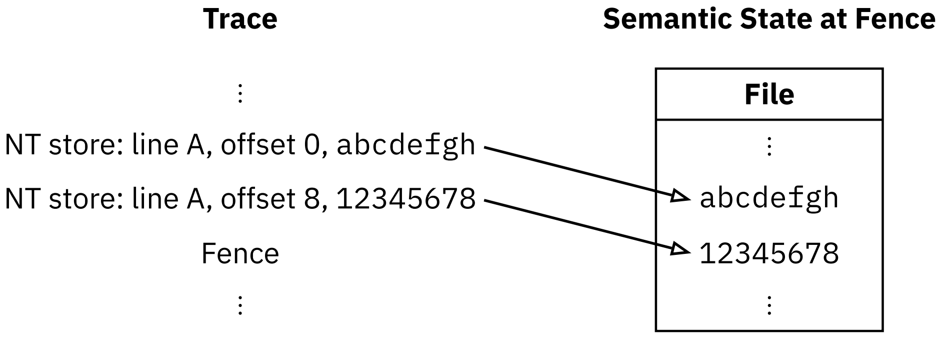

Trace
Semantic State at Fence
⋮
NT store: line A, offset 0, abcdefgh
NT store: line A, offset 8, 12345678
Fence
⋮
File
⋮
abcdefgh
12345678
⋮

Figure 8.12: Example of a state that has a direct dependency on NT stores. Because stores to the file data appear immediately at the subsequent fence, omitting the second store would yield a different semantic state.

cannot run during crash image generation, since it requires information from semantic state extraction. For this reason, **Suvi-NT** has separate analysis and crash image generation phases. The analysis phase runs as part of the tester. If it detects any problematic failure points, the crash image generation can be repeated with this information.

The key idea of **Suvi-NT**'s analysis is to identify semantic states that have a direct dependency on non-temporal stores. If the inclusion of a non-temporal store results in a change in the semantic state, it is likely that applying a prefix of the non-temporal stores will result in a new semantic state.

Figure 8.12 shows an example where this is the case. An application has issued a file write of size 16 bytes with the data `abcdefgh12345678`. The file system uses two non-temporal store instructions to store the file data to PM. The semantic states extracted from the crash images at the following fence immediately include this file data. Since both non-temporal stores write to the same cache line, there are no cache line subsets that would select only one of these stores. **Suvi** would therefore not generate a crash state with only the first store and would consider the operation atomic.

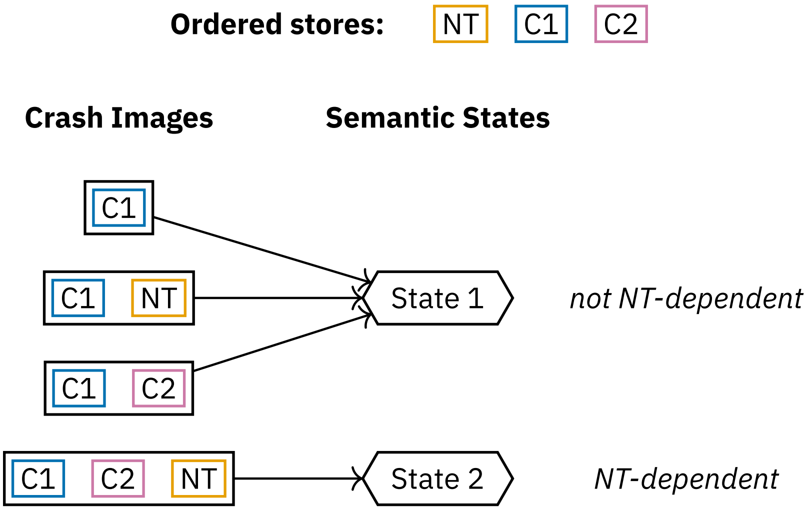


Figure 8.13: Example of detecting NT-dependent semantic states. State 2 and its originating crash are NT-dependent because all originating crash images include NT stores. Refer to Table 8.4 for an explanation of this visualization of the crash image generation algorithm.

Suvi-NT's analysis phase operates as follows. For each semantic state, **Suvi-NT** collects all originating crash images and their failure points. From the crash metadata (see Section 8.6.1), it determines whether all crash images at a particular failure point include NT stores. If that is the case, **Suvi-NT** outputs the originating crash at this failure point as NT-dependent. Figure 8.13 shows an example of this process.

The resulting list of NT-dependent crashes serves as the input for **Suvi-NT**'s crash image generation phase. When the crash image generator reaches a crash point in the list, it generates additional crash images with prefixes of the non-temporal stores to each cache line, as shown in Table 8.8. Since **Suvi** generates additional images only according to the heuristic, spending additional time exploring these combinations is acceptable.

8.4.3 Suvi-Fast: Fast Crash Image Generation for Logic Bugs

We have now seen how **Suvi-Reads** can make crash image generation feasible when there is a large number of dirty cache lines. However, even with the reduced set of lines from **Suvi-Reads**, a large number of images may be generated.

Such an effort is not always necessary to detect bugs. Exploring store reorderings is important to detect misuse of PM primitives, such as a missing memory fence. Logic bugs, in contrast, do not require this effort. These bugs occur when the program temporarily reaches an invalid state during runtime. For logic bugs, generating only a fully persisted image is therefore sufficient.

Mumak [46] is a crash consistency testing tool for userspace software. It uses a testing pipeline similar to **Suvi**'s but traces userspace applications with dynamic binary instrumentation rather than virtual machines. Mumak's differentiating feature is its deduplication of crash points based on stack traces, allowing fast crash image generation. **Suvi-Fast** adopts this strategy for **Suvi**, allowing a fast analysis of kernel file systems. We describe Mumak in more detail in Section 7.5.7.

Table 8.9 summarizes the differences in crash image generation between **Suvi-Reads** and **Suvi-Fast**. As discussed above, **Suvi-Fast** does not explore subsets of stores. At each crash point, it generates only a crash image that includes all pending stores. To ensure more fine-grained steps between images, **Suvi-Fast** and Mumak

	Suvi-Reads	Suvi-Fast
failure points	• memory fences *(with full coverage of failure points since previous fence)*	• cache flushes • memory fences
failure point filter	none	one crash image per unique stacktrace
crash image contents	• no in-flight stores • all stores • subset of stores with **Suvi-Reads** heuristic	• all stores

Table 8.9: Comparison of crash image generation with **Suvi-Reads** and **Suvi-Fast**.

also generate crash images at each cache-flush instruction.[29]

In addition to generating fewer crash images per failure point, **Suvi-Fast** also generates crash images at fewer failure points. It identifies unique failure points by deduplicating them based on their stack trace. As described in Section 8.1.1, **Suvi**'s tracer can optionally record stack traces for each trace entry. For **Suvi-Fast**, these stack traces are mandatory. As proposed by Mumak's authors, **Suvi-Fast** records all failure points in a *failure point tree*. The program counter at the failure point forms the leaf in the tree, and each entry of the stack trace forms the inner nodes. Failure points are inserted into the tree by walking from the root and creating nodes if necessary. If no new nodes are inserted, the failure point has already been visited, and **Suvi-Fast** will not generate another crash image.

Discussion

Crash image generation with **Suvi-Fast** is superficially similar to the crash image generation algorithm for persistent caches described in Section 8.3. In both cases, cached stores are always applied to the crash images in program order. The difference lies in the handling of non-temporal stores. With **Suvi-Fast**, the non-temporal stores are also applied strictly in program order. Otherwise, **Suvi** generates subsets for these stores.

Although **Suvi-Fast** reduces the amount of work performed by the crash image generator, its requirement for stack traces increases the amount of work in the tracer. The total time advantage compared to **Suvi-Reads** is therefore not obvious. If a particular test case already generates few crash images with **Suvi-Reads**, the cost of recording stack traces may outweigh the faster crash image generation with **Suvi-Fast**. We analyze the performance of **Suvi-Fast** in Section 10.3.1.

8.5 Cross-Media Crash Images

So far, we have described our approach for generating PM crash images. We now extend this approach to support cross-media file systems that store data on both PM and NVMe. We first describe how **Suvi** generates NVMe crash images, then how it combines the PM and NVMe images to cross-media crash images.

8.5.1 NVMe Crash Images

Suvi's strategy for generating NVMe crash images largely follows the approach for PM crash images. Since the crash consistency model for NVMe is simpler than for PM, there are some simplifications. As we discuss in Section 2.4.2, the two primary commands relevant for crash consistency are *Write* and *Flush*. A *Write* updates one or more data block, and a *Flush* ensures that all previously completed *Write* commands are persisted on the SSD.

[29] These additional explicit failure points are not useful for **Suvi-Reads**, since the crash images its subset algorithm discovers include the crash images generated there. As discussed in Section 8.3, **Suvi**'s subset generation covers all possible failure points since the previous memory fence.

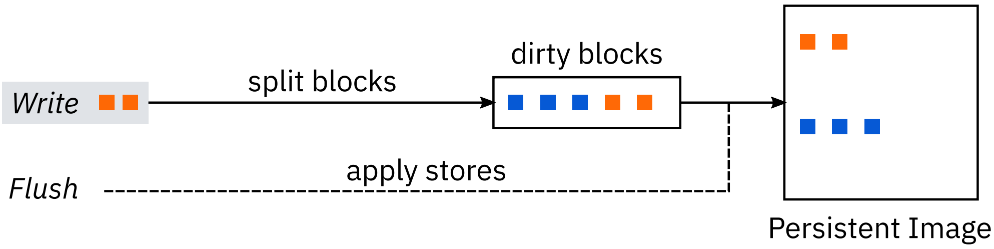


Figure 8.14: **Suvi**'s simulation of the NVMe write path. Written blocks are collected individually into a list. A *Flush* command clears the list and applies it to the persistent image.

Trace Replay

Individual NVMe *Write* commands do not have any ordering constraints. For NVMe replay, it is therefore sufficient to keep track of the fully persisted memory image plus a list of current *Write* commands, as illustrated in Figure 8.14.

A single NVMe *Write* command can contain multiple blocks. Since atomicity is only guaranteed at the granularity of blocks, **Suvi** splits each *Write* command in the trace into its blocks and appends these to the list of dirty blocks. When encountering a *Flush* command, **Suvi** drains the list and applies each *Write* to the memory image.

Failure Points and Crash Images

Due to a lack of other ordering constraints, **Suvi** treats every *Flush* command as a failure point to generate crash images. At these failure points, any combination of dirty blocks with the base persistent image is a valid crash image. **Suvi** always generates two basic crash images: one that includes all dirty blocks, and another with no dirty blocks. Finally, **Suvi** generates images with random subsets of the dirty blocks. Since it is infeasible to generate all possible subsets, **Suvi** only generates images with random subsets up to a fixed maximum number.

Heuristics

Suvi does not implement any heuristics for generating NVMe crash images. We found that **Suvi**'s heuristics for PM crash images are not a great fit for generating NVMe crash images.

Suvi-Reads traces read accesses from a recovery procedure and generates crash images from subsets of these accesses. Although it is possible to adapt this approach for NVMe crash image generation, we found that it does not sufficiently reduce the search space in practice. The reason for this difference is in the usual access strategy. A PM file system usually organizes its data structures on PM for direct access. Therefore, it recovers and reads data with targeted accesses of individual metadata fields.

In contrast, a classic file system works with an asynchronous block interface. For performance and simpler access, it is beneficial to load metadata in batches (e.g., the full superblock structure or the entire journal). Applying **Suvi-Reads** to NVMe, we therefore observe file systems always reading most of the dirty blocks.

Suvi-Fast is the second heuristic that reduces the number of generated crash images by deduplication based on stack traces. Our approach to NVMe tracing works by hooking the virtual NVMe device in the hypervisor. At this location, we cannot

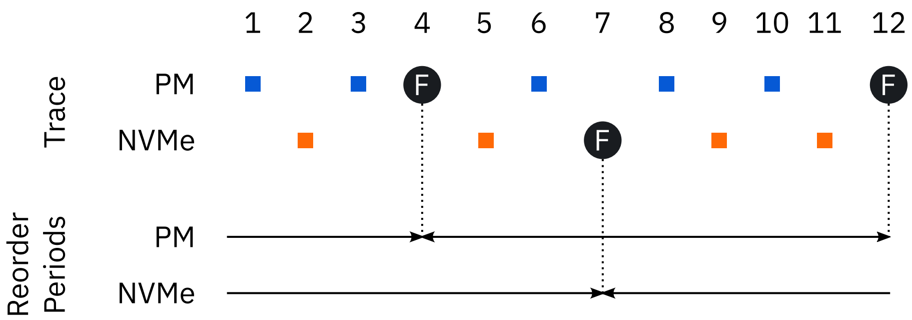


Figure 8.15: **Suvi**'s tracer records a combined PM and NVMe trace, but the PM and NVMe failure points (F) do not necessarily coincide. This results in overlapping reordering periods for the PM and NVMe writes (squares).

capture stack traces for **Suvi-Fast**. Additionally, we do not expect such stack traces to be as useful for NVMe as for PM due to the abstraction from the block layer.

8.5.2 Combined Crash Images

So far, we have described how **Suvi** generates PM-only and NVMe-only crash images. For testing cross-media file system, we need combined crash images with both PM and NVMe data. Combining the crash images is challenging for two reasons.

First, NVMe and PM failure points (PM fences and NVMe *Flush* commands) may be at different positions in the trace. In such cases, is not clear whether they can be reordered. Figure 8.15 shows an example. The NVMe reordering period for the NVMe *Flush* at position 7 extends to before the PM fence at position 4. However, we cannot know whether the file system enforces that the *Flush* (7) happens after the fence (4).

Second, combining an arbitrary number of PM crash images P with an arbitrary number of NVMe crash images N results in $P \cdot N$ combined crash images. Analysis of cross-media file systems might therefore require smaller cutoffs for both PM and NVMe crash images to avoid excessive analysis time.

We solve these challenges in **Suvi** with insight from the NVMe driver implementation in Linux. We find that all interaction with the NVMe submission and completion queues involves instructions that act as fences for PM access. Therefore, a situation as shown in Figure 8.15, where an NVMe *Flush* is not accompanied by a PM fence, is not possible in practice.

With this observation in mind, we design **Suvi** to combine PM and NVMe crash images as follows. At each PM and NVMe failure point, **Suvi** generates the two basic crash images (all current writes, no current writes, which might be identical if there are no writes) for both PM and NVMe. It then generates additional images with subsets for either PM or NVMe according to the current failure point. Finally, **Suvi** combines all PM and NVMe images, resulting in P or $2P$ combined images at a PM failure point, and N or $2N$ combined images a NVMe failure point.

8.5.3 Discussion

Although **Suvi** could be used to analyze NVMe-only file systems, this is not a goal of our design. As PM is directly accessed with load and store instructions, black box testing with virtual machines is beneficial as it avoids errors (see Section 7.4).

In contrast, classic file systems generally work with abstract block devices. For example, the block layer in the Linux kernel translates generic block I/O requests into device commands such as NVMe or SCSI [33]. This makes the block layer the ideal level of abstraction for analyzing classic file systems. CrashMonkey [93] is an approach for crash consistency testing that hooks into the Linux block layer.

We believe that our strategy for combining PM and NVMe crash images strikes a good balance between crash state exploration and efficiency. Since every NVMe command is always accompanied by PM fences, generating and combining subset images for both PM and NVMe is unlikely to discover new crash states.

However, since our approach is based on recording and replaying a single execution of the test case, it cannot detect freedom in the asynchronous NVMe command processing. Going back to the example in Figure 8.15, it is impossible to detect whether the file system waits for the *Flush* command (7) to complete before issuing PM writes 8 and 10. Such a detection would require recording the test case multiple times with variations in timing, which is out of scope for **Suvi**.

8.6 Tester

The tester is the final stage of **Suvi**'s pipeline. Its goal is to check whether any generated crash images exhibit crash consistency bugs.

Suvi's pipeline can automatically detect the following types of bugs:

Recovery crashes The application might encounter an error while recovering a crash image. We consider it a bug if the application crashes or otherwise aborts the recovery.

Multiple final states At the end of every PM operation, all important data should be fully persisted on PM. If **Suvi** detects multiple application states at the end of an operation, we consider it a bug. By inspecting the application state, **Suvi** does not flag cases in which applications store temporary data in PM.

Atomicity violations Finally, **Suvi** can detect operations that exhibit intermediate user-visible states.

Suvi can detect these bugs in a black box fashion, without understanding the structure or contents of the application's crash images or the state contained in these images. **Suvi** relies on a test-specific *state extraction* program that reads the user-visible state from the crash images. The second step, *state analysis*, then detects the bugs above from the extracted states. We describe these steps in detail in the following sections.

8.6.1 State Extraction

The crash image generator yields a set of crash images. As **Suvi** treats the tested application as a black box, there is no way to detect bugs directly from the crash images. With state extraction, **Suvi** allows the tested application to perform recovery

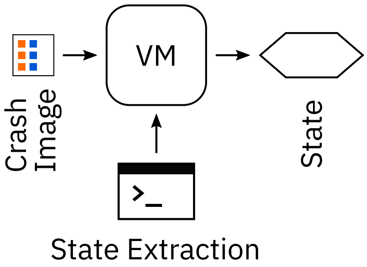


Figure 8.16: A state extraction program specific to the tested application runs in the virtual machine. It performs crash recovery on the input image and serializes the resulting application state.

on each crash image and prints a normalized representation of the user-visible state contained in the image.

The underlying process is very simple, as shown in Figure 8.16. **Suvi** loads each crash image into a virtual machine running a state extraction program. It then reads the output and stores it as the state of the image.

However, there is complexity in the following questions, which are addressed in the following sections:

- How are the resulting states associated with crash images and failure points?
- Which properties are necessary for the representation of the extracted state?
- How is the state of a file system extracted?

To detect errors during crash image recovery, the state extraction runs in a virtual machine provided by the tracer. Memory access tracing is disabled. The tester expects the state extraction program to report success or failure with a hypercall (Section 8.1.3) and records the result.

State Metadata

How are the resulting states associated with crash images and failure points?

For the state analysis, the origin of a crash state is critical. **Suvi** must be able to associate a particular state with the failure points that lead to that state. For example, in test cases with multiple file system operations, we want **Suvi** to analyze the crash consistency properties of each operation.

With every crash image, the crash image generator stores metadata that associates the image with a particular failure point (see Section 8.2.2). The tester extends this metadata during state extraction by adding an association from the extracted state to the originating crash image.

The complete metadata forms a tree for each unique state, as shown in Figure 8.17.

Requirements for State Representation

Which properties are necessary for the representation of the extracted state?

The tester detects crash consistency bugs by comparing the extracted states. As **Suvi** treats the tested application as a black box, it cannot have any knowledge about the structure or content of the extracted states. The tester therefore relies on a simple byte-wise comparison of the extracted states to detect equivalent states. We derive the following requirements from this approach:

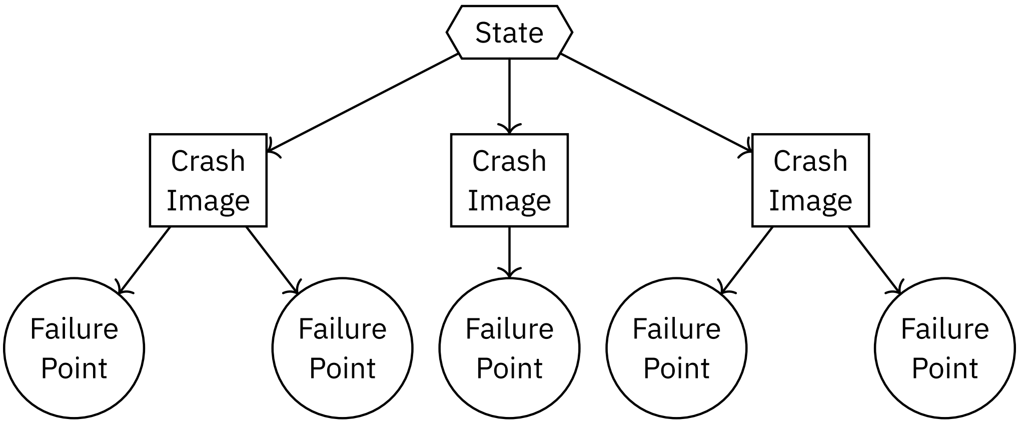


Figure 8.17: Structure of the metadata tree originating from a single semantic state. The state originates from one or more crash images, which in turn originate from one or more failure points.

First, the extracted state must contain only user-visible state. Using the identity function as the state extraction program (i.e., passing through the crash image without changes) would clearly provide a complete representation of the state contained in a crash image. However, such a representation is not useful for further analysis. To implement crash-atomic operations on PM, applications must first write new data without modifying the application's visible state. Only after everything has been written to PM can the application make the new data visible. Consequently, states where the new data is partially written but not yet visible are equivalent to the user, but the tester would not detect these images as equivalent since their crash images differ.

Second, the extracted state must be serialized to a normalized, deterministic format. For example, the order of entries in a hash map is often non-deterministic or even actively randomized [20, 26]. Two hash maps with identical contents might therefore serialize to different representations if a consistent ordering is not enforced.

Finally, although not required for **Suvi**'s analysis, we found that a human-readable format is beneficial for debugging. If **Suvi** detects an atomicity violation, the first debugging step is always to develop an understanding of the states that **Suvi** discovered. A human-readable format makes such an analysis easy and avoids the need to load each state (or crash image) in another interactive VM. We describe the debugging process in more detail in the following section.

State Extraction for File Systems

How is the state of a file system extracted?

Since **Suvi**'s primary testing targets are file systems, we now describe our approach to state extraction for a file system. Most operating systems provide a standardized interface for accessing different file system implementations. On Linux and other Unix-like operating systems, this interface conforms to the POSIX standard [14]. We therefore define the user-visible file system state as all state that can be discovered through the POSIX file system API. The relevant POSIX functions for reading the file system are:

- `opendir()` and `readdir()` for traversing directories

```
{
  "/mnt": {
    "typeflag": "D",
    "st_ino": 1,
    "st_mode": 16877,
    "st_nlink": 2,
    "st_uid": 0,
    "st_gid": 0,
    "st_size": 4096,
    "st_blocks": 0,
    "st_atim_sec": 1738581554,
    "st_atim_nsec": 0,
    "st_mtim_sec": 1738581555,
    "st_mtim_nsec": 0,
    "st_ctim_sec": 1738581555,
    "st_ctim_nsec": 0
  },
  "/mnt/myfile": {
    "typeflag": "F",
    "content": "test",
    "st_ino": 33,
    "st_mode": 33188,
    "st_nlink": 1,
    "st_uid": 0,
    "st_gid": 0,
    "st_size": 4,
    "st_blocks": 8,
    "st_atim_sec": 1738581555,
    "st_atim_nsec": 0,
    "st_mtim_sec": 1738581555,
    "st_mtim_nsec": 0,
    "st_ctim_sec": 1738581555,
    "st_ctim_nsec": 0
  }
}
```

Listing 8.2: Example `fs-dump` output from a folder with a single file.

- `lstat()` for reading metadata of discovered files (including directories and symlinks)
- `open()` and `read()` for reading contents of regular files
- `readlink()` for reading symbolic link targets

We design a tool, `fs-dump`, that uses these functions to traverse a directory tree. For each discovered file, directory, or symbolic link, it records the metadata returned by `lstat()`. For files and symlinks, `fs-dump` also records the file contents and link targets. `fs-dump` collects this data in a map indexed by the full path of each file. Finally, it serializes the data to JSON and prints the result. Listing 8.2 shows an example of the resulting output.

`fs-dump` satisfies the properties given in the previous section:

- Since the POSIX API is used for reading the file system state, all state discovered by `fs-dump` is user-visible.[30]
- Deterministic output is ensured by sorting the keys in the JSON serialization.
- The JSON is formatted with appropriate whitespace to make it human-readable.

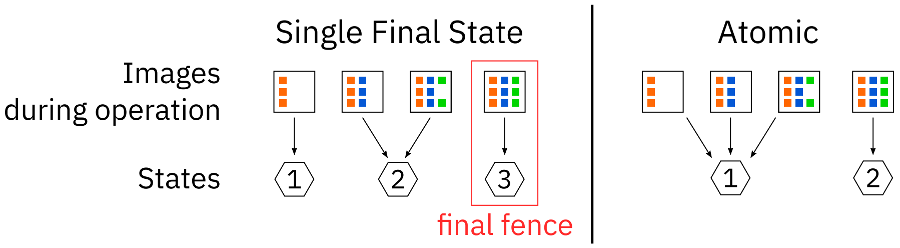


Figure 8.18: The tester detects single final state and atomicity of an operation by counting unique states at the final fence or during the entire operation.

[30] A file system might, however, offer access to additional state through non-standard APIs. A file-system-specific state extraction tool would be required to test such APIs.

8.6.2 State Analysis

We now describe how the tester can detect single final state and atomicity violations given a list of extracted states along with their metadata. Figure 8.18 shows a high-level overview of the approach. Using the crash metadata associated with the crash images, the tester discovers states that were generated during the test operation. By counting the number of unique states at the final fence or during the entire operation, the tester detects crash consistency as follows:

Single Final State An operation has a single final state if exactly one unique state has been extracted from images generated at the last fence of the operation.

Atomic An operation is atomic if it has a single final state and at most two unique states in total.

Suvi follows the definitions of these two properties as introduced for Vinter [68].

Analysis Algorithm

We now describe the algorithm that checks the two crash consistency properties.

Since tests delimit individual (file system) operations with checkpoints, the tester starts by collecting states for each checkpoint into the following data structures:

```
type CheckpointID = i64
struct CheckpointFP {
    trace_id: u64
    states: Set<State>
}
let checkpoint_states: Map<CheckpointID → Set<State>> ←
    collect states discovered from checkpoint N to N+1
let checkpoint_fp: Map<CheckpointID → CheckpointFP> ←
    find earliest failure point of checkpoint N
```

The crash metadata forms trees with the states at the root (see Figure 8.17). The current checkpoint ID is part of the failure-point metadata (see Section 8.2.2). For each state, the tester therefore traverses all failure points and inserts the state into the appropriate sets in `checkpoint_states`. Additionally, it finds the earliest failure point for each checkpoint by comparing the `trace_id` and collects the states at that failure point.

```
for each checkpoint_id within analysis range {
    let is_single_final_state: bool ←
        checkpoint_fp[checkpoint_id + 1].states.len() = 1
    let is_atomic: bool ←
        is_single_final_state and checkpoint_states[checkpoint_id].len() ≤ 2
    print is_single_final_state, is_atomic
}
```

```
Test: test_hello-world

Command: hypercall checkpoint 0 && sync &&
         hypercall checkpoint 1 && touch /mnt/myfile && sync &&
         hypercall checkpoint 2 && echo HelloWorld > /mnt/myfile &&
         hypercall checkpoint 3 && sync &&
         hypercall checkpoint 4

Checkpoint 1 -> 2:
Trace entries 295565 -> 295691
2 states: c1s0, c1s1
Single final state: c1s1
atomic

Checkpoint 2 -> 3:
Trace entries 295691 -> 300021
3 states: c1s1, c2s0, c2s1
2 final states: c2s1, c2s0
Dirty cache lines at checkpoint: 46848
not atomic

Checkpoint 3 -> 4:
Trace entries 300021 -> 300022
2 states: c2s0, c2s1
2 final states: c2s1, c2s0
Dirty cache lines at checkpoint: 46848
(atomic)

0 NT-dependent states
```

Listing 8.3: Example output from the tester analyzing the “hello-world” test result for NOVA.

Once the maps are populated, the tester can determine the crash consistency properties for each checkpoint.

- The operation between checkpoints N and N+1 has a single final state if exactly one state is discovered at the first failure point of checkpoint N+1.
- The operation is atomic if it has a single final state and at most two states are discovered between checkpoints N and N+1.

Output

Listing 8.3 shows an example of the tester’s output. After a header showing the test name and the test command, the tester prints the following information for each checkpoint within the analysis range defined for the test:

- A list of discovered states.
- A list of final states.
- Whether the operation is atomic.

The name of each state corresponds to a file created by the state extraction step, allowing manual inspection of these states. **Suvi** assigns these names using the checkpoint ID and a counter based on when the state was first discovered. A state name is printed in green if it originated from at least one fully persisted crash image, and otherwise in purple. The presence of purple states hints at a misuse of PM primitives in the tested application, since such states would never appear during the regular runtime of the application.

To aid with debugging detected crash consistency bugs, the tester can optionally print the following information:

- Line-by-line diffs of the discovered states (Listing 8.4 (a)).
- The locations of the checkpoints in the trace, allowing a trace analysis for this checkpoint (Section 8.7.1).

```
2 final states:  c2s1, c2s0
--- c2s0.txt
+++ c2s1.txt
@@ -21,3 +21,3 @@
     "typeflag": "F",
-    "content": "HelloWorld\n",
+    "content": "HelloWor\u0000\u0000\u0000",
     "st_ino": 33,
```

(a) Line-by-line diff between the two final states.

```
c2s0 at fences:
  300017
  300021
  300022
c2s1 at fences:
  300017 (dirty lines: 46848)
  300021
  300022
```

(b) Failure points where the two final states were discovered.

Listing 8.4: Additional optional output from the tester analyzing the "hello-world" test result for NOVA.

- In the case of a single final state violation, the dirty cache lines at the checkpoint.
- The failure points from which each state originated (Listing 8.4 (b)).

The final line shows the results from **Suvi-NT**, which we describe in Section 8.4.2.

8.7 Trace Analysis

Suvi implements two types of trace analysis. The first type, which we call *trace debugging*, provides tools that give **Suvi**'s users a better understanding of the results of the crash consistency testing pipeline.

The second type, which we call *trace heuristics*, detects patterns in the trace that suggest a crash consistency bug.

8.7.1 Trace Debugging

We introduce **Suvi**'s trace debugging by continuing the analysis of the "hello-world" test on the NOVA file system. As shown in Listing 8.3, **Suvi**'s testing pipeline detected multiple final states. This problem occurs when modified data remains in the caches at the end of an operation. A user who is debugging this problem therefore wants to understand where this data originates and where a cache flush must be introduced.

Filtering Trace Entries by Cache Line

Suvi provides the set of dirty cache lines for a single final state violation. We introduce a command to print all trace entries that access these cache lines, as well as relevant memory fences and hypercalls.

Listing 8.5 shows the output for the dirty cache line of the NOVA "hello-world" test case. We can see two phases. First, NOVA writes zero to each byte of the cache line followed by a cache line flush. Then, it writes "HelloWorld⏎"[31] to the cache line with an eight byte non-temporal store and three regular single byte stores. Since these regular stores are not followed by a cache line flush, they remain in the caches and cause the single final state violation. A user of **Suvi** can now examine the stack traces associated with these *Write* entries in the trace to determine where a cache line flush needs to be introduced.

Understanding Suvi's PM Simulation

From the trace output alone, it is not always obvious how **Suvi** discovered a particular crash image. For instance, in our NOVA test case, there are three store instructions, so

[31] The ⏎ symbol indicates a newline character.

```
     0 Hypercall start 0
295564 Hypercall checkpoint 0
295565 Hypercall checkpoint 1
295691 Hypercall checkpoint 2
295784 Write     0x2dc00b size 1
       00    .
295785 Write     0x2dc00c size 1
       00    .

[... 50 Write entries skipped ...]

295836 Write     0x2dc03f size 1
       00    .
299869 Flush     0x2dc00b clwb
299933 NT-Write 0x2dc000 size 8
       48 65 6c 6c 6f 57 6f 72   HelloWor
299934 Write     0x2dc008 size 1
       6c    l
299935 Write     0x2dc009 size 1
       64    d
299936 Write     0x2dc00a size 1
       0a    .
299937 Fence sfence
300021 Hypercall checkpoint 3
300022 Hypercall checkpoint 4
300023 Hypercall success 0
```

Listing 8.5: Shortened output from **Suvi**'s read-trace command analyzing accesses to the unflushed cache line of the "hello-world" test on NOVA. For each trace entry, **Suvi** prints the entry's index, its type, and any additional information. In particular, for *Write* entries, the destination address, the size, and the written data (as hexadecimal bytes and ASCII) are printed.

we might expect three unique states with partial file data. However, **Suvi** discovered only two final states.[32] We therefore introduce a command that annotates the trace with the state of **Suvi**'s PM simulation.

Listing 8.6 shows the output for the first fence where the broken state `c2s1` was discovered, as indicated by the tester (see Listing 8.4 (b)). We can see that the three store instructions writing the "ld⏎" of "HelloWorld⏎" are still part of the volatile state

```
fence id 300017
  line 14019
    offset  8 counter 4332 data ________________40c02d0000000000________________
________________________________________________________________________________
    Flush

  line 46848
    offset  8 counter 4260 data ________________6c______________________________
________________________________________________________________________________
    offset  9 counter 4261 data ________________6c64____________________________
________________________________________________________________________________
    offset 10 counter 4262 data ________________6c640a__________________________
________________________________________________________________________________
```

Listing 8.6: Shortened output from **Suvi**'s process-trace command. It shows the state of **Suvi**'s PM simulation at a particular failure point (see Section 8.3). For each store to a cache line, the output shows the offset within the cache line, the internal store counter for tracking the global store order, and the volatile contents of the cache line after the store. The current store is marked in red. The blue background indicates that this part of the cache line was read by the recovery for **Suvi-Reads** (see Section 8.4.1).

[32] Vinter's PM simulation was less accurate than **Suvi**'s and generated invalid crash images with a subset of these store instructions, resulting in four final states. [68, §5.3.1]

(compare Listing 8.5). The eight-byte write updates a pointer[33], making all changes to the file visible atomically. Since both the pointer update and the previous stores to the file contents use regular (temporal) store instructions, the instruction set architecture guarantees a global ordering between these stores (see Section 2.4.1). **Suvi** tracks this ordering with the counter printed in Listing 8.6. As the pointer update is ordered after the stores to the file contents, states with only a prefix of these stores are not possible. Either both cache lines are included in the image with all writes (resulting in state `c2s0`), or at least one of the cache lines is missing (resulting in truncated file contents or the original file).

8.7.2 Trace Heuristics

Suvi adapts its trace heuristics from previous work [46]. It detects the following patterns in the trace:

Redundant flush or fence. A flush or fence instruction that does not have any preceding stores to PM. This is a performance bug and does not indicate a crash consistency issue.

Missing flush or fence. Reported when there are stores that have not received a flush and fence at the end of the trace.

Overwrite without flush or fence. Reported when a store instruction overwrites a preceding store that did not receive a flush and fence.

Unordered flushes. Reported when there are multiple flush instructions between two fences.

These heuristics can only hint at potential bugs arising from misuse of PM primitives. In particular, unordered flushes frequently arise when a PM application needs to persist data larger than one cache line. The trace heuristics cannot detect whether such a persist operation is, for example, protected by a journal. However, the trace heuristics are useful in conjunction with **Suvi**'s crash image generator heuristics: After a first analysis with **Suvi-Fast**, the trace heuristics can identify tests that warrant further analysis with **Suvi-Reads**.

[33] The stack trace associated with that store points to the function `nova_update_tail`, which updates the tail pointer of an inode log. In NOVA, each inode has its own journal [125].

Chapter 9

Suvi: Implementation

Suvi is implemented in approximately 10 000 lines of Rust code. Its source code is available at https://github.com/lluchs/suvi

In this chapter, we describe details from **Suvi**'s implementation. We first describe **Suvi**'s two implementations of the tracer component. Then we take a look at two implementation details that are critical for **Suvi**'s performance: memory images and parallelization.

9.1 Tracer

Suvi implements two tracers with different features, one based on PANDA and one on plain QEMU. We describe the differences between these implementations below. In Section 10.3.2, we compare the performance of the tracer implementations.

Both implementations use a similar architecture. Our tracing code is compiled as a shared library, which is linked into the emulator. The emulator provides hooks for memory accesses and other instructions, which call into our library during guest code execution. While the guest is paused, our library can then inspect guest CPU registers, translate memory addresses, and access guest memory to assemble a trace entry (see Section 8.1).

We ensure that the guest execution is blocked as little as possible by passing the trace entries to a separate thread for output. This thread then serializes the trace entry and writes it to a compressed output file. The guest execution continues immediately without waiting for expensive I/O system calls.

9.1.1 PANDA

PANDA [38] is a fork of QEMU with additional features focused on reverse engineering. It runs virtual machines with binary translation and allows interaction with the guest code through two APIs, plugin and libpanda.

First, plugins built with the plugin API are linked into PANDA and allow direct access to the guest state, including instruction hooks. **Suvi**'s PANDA tracer is built on top of panda-rs, a Rust adapter for PANDA's plugin API [92].

Second, libpanda provides an API for configuring and running a PANDA instance and then interacting with it. We use its Python API to launch PANDA, to load and store PM snapshots, and to interact with the guest over a serial device. Although libpanda can also install hooks, we opt for a Rust plugin for memory and instruction tracing due to its lower overhead.

The primary limitation of PANDA is the age of the underlying codebase. PANDA was forked from QEMU 2.9.1, released in 2017 [107]. Since this version of QEMU does not include support for NVMe devices, we needed a different approach for tracing cross-media file systems.

9.1.2 QEMU

Current versions of QEMU include support for TCG plugins [29], which allow hooking into QEMU's emulation backend called *Tiny Code Generator* (TCG). These plugins are linked into QEMU and allow interaction similar to PANDA plugins, although with limited functionality.

We build **Suvi**'s QEMU tracer based on QEMU version 8.0. QEMU required patches to add missing functions to the TCG plugin API. In particular, we needed to introduce functions for reading and writing guest memory. Additionally, QEMU's code generation translates cache line flush instructions to NOPs, which discards the address parameter. To trace these instructions with the address, we modified their code generation to trigger a memory access.

For NVMe tracing, we introduced hooks into QEMU's virtual NVMe device. These hooks call into our tracer plugin. Every NVMe trace entry requires information from multiple hooks, since an NVMe command is processed in multiple steps. A hook in the NVMe request handler provides the NVMe request with its arguments, a hook in the DMA handler provides the data for write commands, and a hook in the completion handler detects command completion. Each hook passes a message to the output thread, which assembles a trace entry.

9.2 Memory Images

The primary challenge for performance in the crash image generator is handling memory images. Depending on the file system under test, **Suvi** deals with memory images that have a size of up to hundreds of MiB.

Suvi's memory images need to support the following operations:

- **Clone** the image for independent mutation. Since the crash image generator applies different subsets of writes to a base image, it is not possible to mutate a single image sequentially.
- **Persist** the image to a file. **Suvi** decouples crash image generation and analysis, which allows parallel analysis.
- Calculate a **hash** of the image contents. **Suvi** uses these hashes to determine whether it has generated an image before.

We implement efficient support for these operations with two key ideas: copy-on-write files and partial hash pre-calculation.

9.2.1 File System Copy-on-Write

Some modern Linux file systems, including btrfs and XFS, support sharing data blocks between multiple files. Unlike hard links, modifications to such files remain private through a copy-on-write mechanism. File copies with shared data blocks are created with the `FICLONE` ioctl [21]. They are commonly referred to as *reflink* copies.

Our strategy in **Suvi** is to use memory-mapped temporary files as memory images. For reflink copies to work, these temporary files must be located on the same file system as the output directory. To clone an image, **Suvi** creates and maps a new temporary file that shares the data blocks from the original image. To persist an image, **Suvi** performs a reflink copy of the image's data blocks into the destination file.

Compared to a regular memory buffer, this strategy has two advantages. First, it avoids copying data as much as possible. Both the clone and the persist operations only need to copy file metadata. Only modifications to the images require a copy-on-write operation for that part of the file.

Second, the strategy saves space by sharing file blocks in the output. Since crash images often differ by only a few bytes, most file data within the crash image files is shared.

9.2.2 Hash Memoization

Suvi needs to calculate a hash of each memory image before persisting it to detect whether it has already discovered an identical crash image. This calculation is expensive for large images. Additionally, with the copy-on-write strategy above, reading the entire freshly mapped image causes expensive page faults.

We reduce this cost by using a hash function based on Merkle trees, which allows reuse of previously calculated intermediate values. We choose BLAKE3 [98] as the hash function. As shown in Figure 9.1, BLAKE3 splits its input into chunks of size 1024 bytes, then recursively hashes pairs of chunks until it reaches the root of the tree.

For each memory image, **Suvi** memoizes a configurable level (counted from the leaves) of this tree. When modifying the memory image, **Suvi** invalidates the corresponding intermediate hash value via a bitmap. A later hash calculation must re-calculate the intermediate values and then finalize the hash.

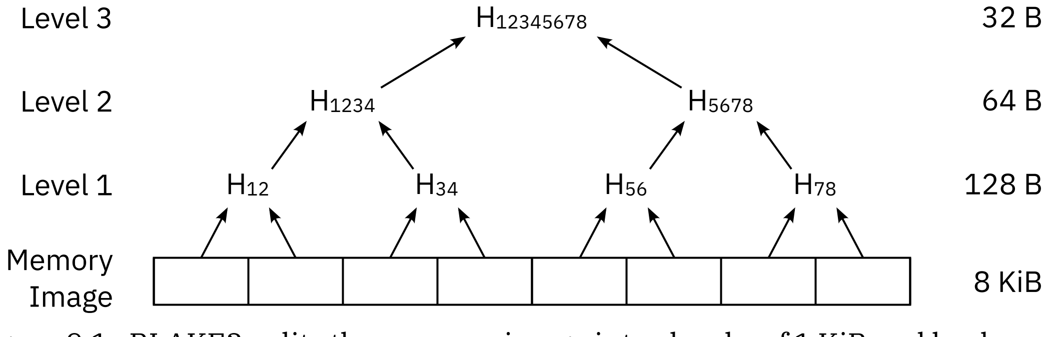


Figure 9.1: BLAKE3 splits the memory image into chunks of 1 KiB and hashes pairs of chunks recursively. The root of the tree is the final hash value. The right side shows the total size of each level. Every intermediate value has a size of 32 bytes.

The choice of level is a trade-off between how much data of the memory image must be read and how large the memory overhead from the cached hash values is. For example, at level 5, each intermediate hash value corresponds to $2^5 = 32$ chunks or 32 KiB of data. Re-calculating one intermediate value therefore requires accessing eight 4 KiB pages. For a 100 MiB memory image, the intermediate hash values need 100 KiB, and the bitmap needs 400 bytes of memory. We evaluate this trade-off below.

9.2.3 Evaluation

We evaluate the performance of the memory images with microbenchmarks that exercise the individual operations. We compare our implementation based on memory-mapped files (mmap) with a simple implementation based on Rust vectors (Vec). Rust vectors are dynamically allocated managed arrays. The clone operation allocates a new vector and copies the data, and the persist operation writes the data to the destination file.

We run the benchmarks on pc62, which we describe in Section 3.2. We set up an XFS file system on the SSD to hold the memory-mapped files. Note that **Suvi** does not use Optane PM for tracing or the memory images.

We evaluate memory images of size 5 MiB and 100 MiB, which is a typical range for the file system images we test in **Suvi**. For example, we test PMFS and NOVA with 5 MiB images.

Memory-mapped images

Figure 9.2 compares the clone and persist operations of the two implementations. Both operations are significantly faster with the mmap implementation. Cloning a 5 MiB mmap image requires 56 µs, which is 88% faster than the Vec implementation. On 100 MiB images, mmap clone is 90% faster than Vec clone. The persist operation is 96% faster on 5 MiB images and 97% faster on 100 MiB images.

With the reads heuristic, we typically observe 26% more calls to clone than to persist, since each unique image is only persisted once. If **Suvi** generates 100 crash images with size 100 MiB (i.e., 100 calls to persist and 126 calls to clone), the mmap images save 10 s of time in the crash image generator.

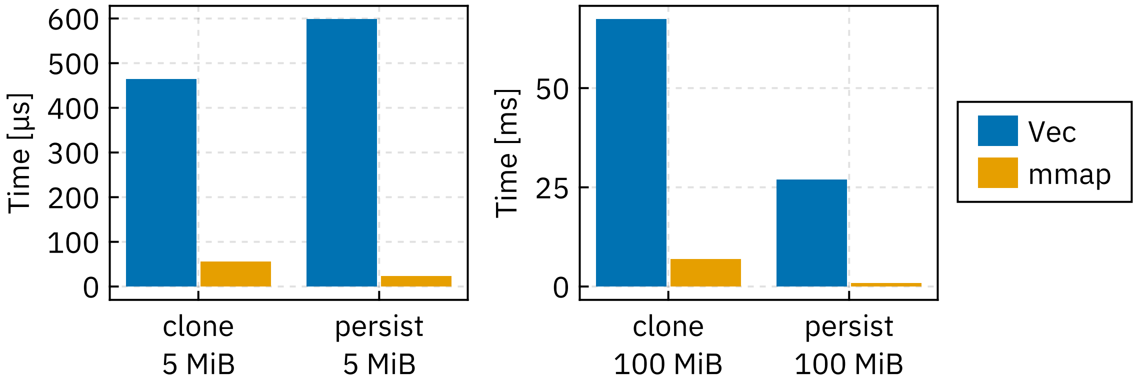


Figure 9.2: pc62 Comparison of clone and persist operations on memory images with size 5 MiB and 100 MiB.

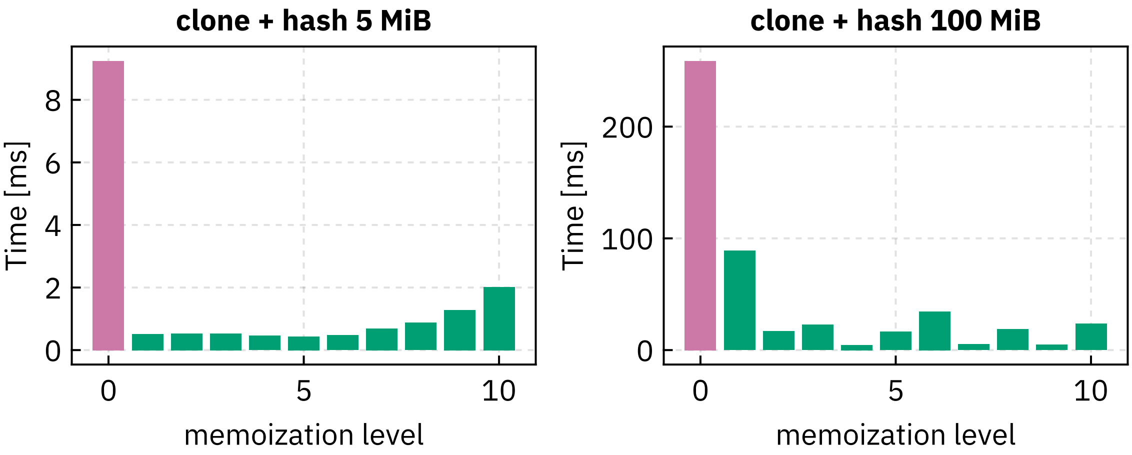


Figure 9.3: pc62 Evaluation of memoization levels for a benchmark that clones a memory image, modifies one byte, then calculates the hash. The bar at 0 shows runtime without any memoization.

Hash Memoization

We evaluate memoized hashing with a benchmark that clones a memory image, overwrites and invalidates one byte, and then calculates the hash. The benchmark includes a clone operation because the reduction of page faults for the mmap images is a major advantage of memoization and because this combination matches usage in **Suvi**.

In Figure 9.3, we compare different memoization levels, including no memoization (shown as level 0). Any memoization level yields a large speedup compared with a regular hash calculation that always reads the entire buffer. Lower memoization levels reduce the amount of data that must be rehashed but increase the size of the memoized hash values that must be copied.

For the 100 MiB image size, there is a large variance in runtime between the different memoization levels. These results are stable for multiple iterations within one benchmark run (i.e., cloning and hashing one origin image multiple times) but differ between benchmark runs. We attribute this to how the XFS file system allocates the blocks of the origin image file, which affects the performance of the clone operation and the resulting page faults.

With 5 MiB images, there is less variance. The runtime for levels one to six is consistently low, with a small decrease between four and six. At levels higher than six, the runtime rises because the cost of recalculation outweighs the savings from copying less memoized data.

Based on these results, memoization levels between four and six provide good performance, especially for small 5 MiB images. We chose level five for **Suvi**.

9.3 Parallelization

There are two opportunities for parallelization in **Suvi**, as shown in Figure 9.4: the state extraction within a test and the entire pipeline across multiple tests.

Within the testing pipeline, the tracer and the crash image generator must run strictly sequentially. The state extraction, in contrast, is trivially parallelizable. Each

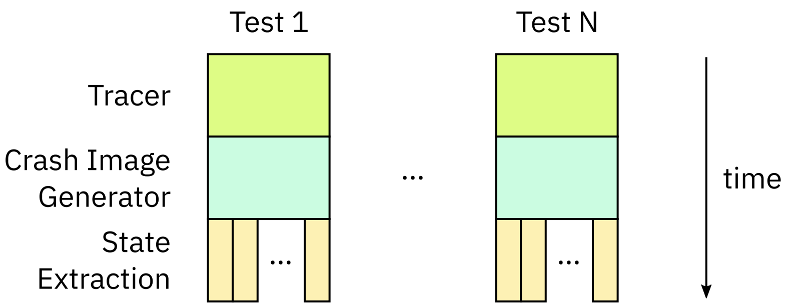


Figure 9.4: Parallel execution in **Suvi** during state extraction and across multiple tests.

instance extracts the state of one crash image, which is independent of all other crash images.

Similarly, multiple invocations of **Suvi**'s pipeline on different tests are independent and can be executed in parallel. However, combining these parallelization opportunities can easily result in resource exhaustion if multiple **Suvi** pipelines run parallel state extraction at the same time.

We solve this problem by running the pipeline stages as separate jobs in a shared thread pool. The tracer and crash image generator make up one job per test. For the state extraction, each instance extracting one crash image runs in a separate job. This allows parallel execution of all pipeline stages without exceeding resource limits.

9.3.1 Discussion

Suvi's strategy for parallelization achieves good resource utilization when enough tests (≥ number of CPU cores) are analyzed in parallel. However, **Suvi**'s runtime for analyzing a single test is dominated by the sequential tracer and crash image generator stages. There are two opportunities to improve parallel resource optimization within the **Suvi** pipeline, as shown in Figure 9.5.

First, rather than running the pipeline stages strictly in sequence, **Suvi** could trigger state extraction immediately as it discovers new crash images (Figure 9.5 (a)). We do not implement this approach for **Suvi**, since it introduces additional complexity while often gaining little time. Assuming that state extraction takes similar time for all crash images, the extraction of the final generated image determines when the pipeline finishes. However, if there are fewer crash images than CPU cores, the

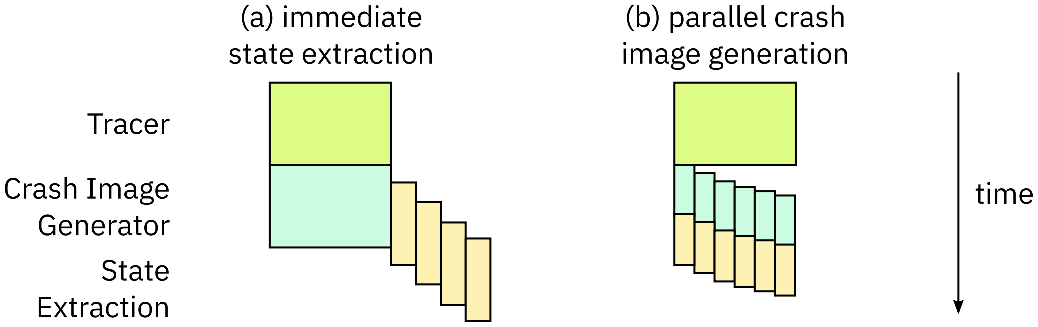

(a) immediate state extraction
(b) parallel crash image generation
Tracer
Crash Image Generator
State Extraction
time

Figure 9.5: Opportunities for improving parallel resource utilization for a single test.

separate state extraction phase takes just as long, since all states are then extracted in parallel. For example, when analyzing the NOVA file system, all except two of **Suvi**'s regular test cases have fewer than 36 crash images and can be processed in parallel on our test machine with 36 CPU cores.

Second, the crash image generator itself could be parallelized (Figure 9.5 (b)). By duplicating the state of the PM simulation at each fence, **Suvi** could continue trace replay in one thread while generating crash images in another. The primary time gain would stem from **Suvi-Reads** (see Section 8.4.1), since multiple recovery traces could run in parallel. We do not implement this approach either due to the high additional complexity. Duplicating the state of the crash image generator is additional work that would slow down the common case of parallel test execution. Additionally, **Suvi** includes an alternative heuristic, **Suvi-Fast**, that minimizes time spent generating crash images.

Chapter 10

File System Testing with Suvi

In this section, we discuss using **Suvi** to analyze crash consistency of PM file systems. We first describe the virtual machines with the test environment, then the file system test cases. We evaluate **Suvi**'s performance and discuss the analysis results.

10.1 Virtual Machine Setup

Suvi analyzes file systems running in a virtual machine. Rather than using a virtual machine image of a regular Linux distribution, we create minimal environments for our file system tests. This approach has two advantages. First, a minimal environment reduces the resources necessary for running the virtual machine. Most drivers and services that are included in generic images are not needed for the file system tests, but increase boot times and memory utilization. Second, since we precisely control each component that is part of the virtual machine, we improve reproducibility of the test results.

With direct Linux boot, QEMU supports starting a Linux kernel without the usual firmware startup. We provide QEMU with a Linux kernel image and an *initramfs* image. QEMU loads both files into the guest memory and then runs the kernel entrypoint. As part of its boot procedure, the kernel extracts the initramfs image into an in-memory root file system. Finally, Linux runs the init program contained in the image. The initramfs image is immutable. Any changes to the root file system in the guest are lost once the virtual machine terminates.

We prepare an initramfs based on BusyBox [22], which is a collection of common Unix command-line tools in a single executable. Besides BusyBox, the initramfs contains file system administration tools such as `mkfs` and our `fs-dump` tool for state extraction (see Section 8.6.1). The init process starts a POSIX shell that waits for input from the serial device. Once the shell is ready, **Suvi** sends commands over the serial device that contain the test case, run file system recovery, or the state extraction program.

We ensure reproducible test environments by building the Linux kernels and initramfs images with Nix [39]. This is particularly important for older research file systems such as PMFS which do not build with modern toolchains.

10.2 Test Cases

Suvi includes two types of test cases. First, manually-written test cases originally written for Vinter, which we extend. Second, automatically-generated sequences of file system operations.

10.2.1 Vinter Test Cases

Suvi primarily uses the 16 hand-written test cases inherited from Vinter [68]. These test cases are implemendet as short shell scripts and cover most POSIX file system operations. For **Suvi**, we modify some of the test cases so that every checkpoint contains at most one file system operation. These modifications enable fully automatic testing, including reporting results. Further, we introduce two additional test cases that cover the `truncate` and `fallocate` operations.

There are the following test cases:

Test Name	**Tested Operations**
Hello World	create file, write to file
append	append data to file
atime	read file, updating its access time
ctime/mtime	delete file from directory, which updates the directory's modification timestamp
chmod	change file access mode
chown	change file owner
link	create and remove a hard link
symlink	create a symlink
mkdir/rmdir	create and remove a directory
rename: overwrite	rename a file, atomically overwrite target
rename: directory	rename a directory with contents
rename: long name	rename a file to a long name
touch	create a file, then update its accessed and modification timestamps
long name	create a file with a long name, then write to it
unlink	remove a file
update	write to the middle of a file
fallocate	increase file size with `fallocate`, then write to the allocated space
truncate	reduce file size with `truncate`

10.2.2 Automatic Test Case Generation

Besides manually-written test cases, **Suvi** also supports automatically generated test cases with ACE. Originally introduced with CrashMonkey [93] and extended for Chipmunk [81], ACE is an approach for automatically generating short sequences of file system operations.

ACE chooses a fixed-length sequence of file system operations from a pre-defined set. For each operation, it then randomly selects appropriate parameters. Most file system operations have dependencies. For example, a file must exist before it can be deleted. As final step, ACE resolves these dependencies by inserting additional operations.

ACE outputs tests in a high-level language called J-lang. For CrashMonkey and Chipmunk, the authors convert the J-lang tests to C++ source code, which is then compiled and executed.

We adapt ACE for **Suvi** as follows. We modify the test case generation to insert checkpoints around the last operation of the generated sequence. **Suvi** will therefore test atomicity only of the last operation in the sequence. This avoids duplicate work, since earlier operations are already covered by shorter test sequences. Generating and compiling C++ code from the J-lang files is not a good fit for **Suvi**, since it runs tests in a minimal virtual machine without a complete userspace. Instead, we build a J-lang interpreter that reads and directly executes file system commands.

We exhaustively generate and analyze tests with a sequence length of one (seq1) and two (seq2).

10.3 Performance

The performance of a crash consistency testing system is important for two reasons. First, faster analysis allows processing more test cases over time and therefore achieving higher test coverage. Second, once a bug is identified, the testing tool should be fast enough for interactive use on a small test set for debugging.

In this section, we examine **Suvi**'s performance when analyzing file systems. We focus on the following questions:

- How does the analysis performance of **Suvi**'s heuristics, **Suvi-Reads** and **Suvi-Fast**, compare?
- How do the two tracers based on PANDA and QEMU compare?
- How large is **Suvi**'s speedup from parallelization?

We base the following analysis on performance data from analyzing the NOVA file system with 1515 seq2 tests generated with ACE (see Section 10.2.2).

10.3.1 Heuristics

In Figure 10.1, we compare the median sequential runtime of **Suvi** with **Suvi-Reads** and **Suvi-Fast**. Both heuristics aim to avoid combinatorial explosion in the crash image generator. **Suvi-Reads** traces read accesses during file system recovery, and **Suvi-Fast** deduplicates failure points by stack trace. See Section 8.4 for a detailed description of these heuristics.

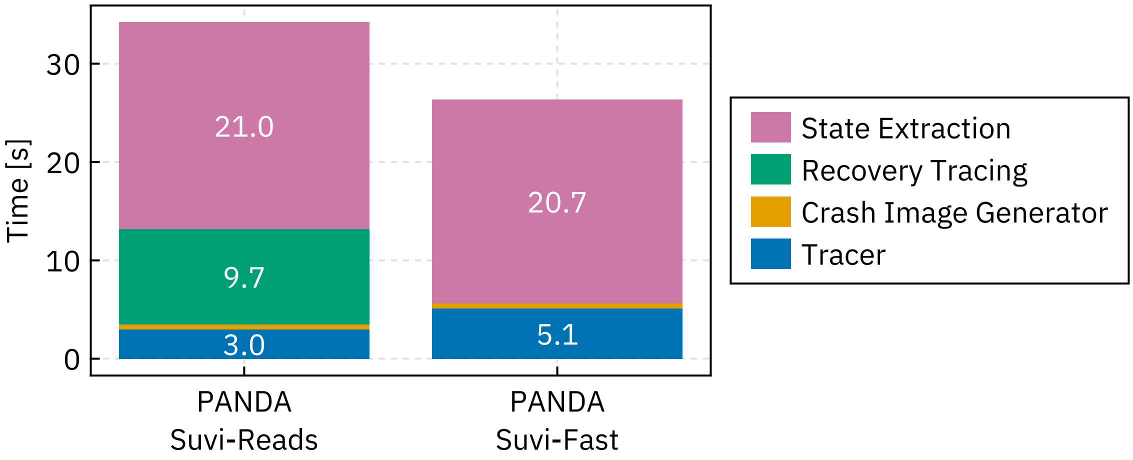


Figure 10.1: pc62 Comparison of analysis times for **Suvi**'s heuristics (**Suvi-Reads** and **Suvi-Fast**) with the PANDA tracer. Median sequential execution times for the ACE seq2 tests on NOVA.

The runtime differs significantly in the tracer and the crash image generator. **Suvi-Fast** requires collecting stack traces during tracing, which introduces a tracing overhead of 71%. While the core of the crash image generator (i.e., PM simulation and writing crash images) takes less than one second for both **Suvi-Reads** and **Suvi-Fast**, the recovery tracing for **Suvi-Reads** takes a significant amount of time. In total, tracing and crash image generation finish 57% faster with **Suvi-Fast** than with **Suvi-Reads**.

For both heuristics, the state extraction requires the largest amount of time, which scales linearly with the number of crash images. As **Suvi-Fast** generates fewer crash images than **Suvi-Reads**, we observe a slightly smaller state extraction time.

The time shown for state extraction in Figure 10.1 is for sequential execution. The tests generate 10 crash images on average. Assuming that there are enough CPU cores to extract all states in parallel, the state extraction finishes in 2.1 seconds.

10.3.2 Tracer Implementations

In Figure 10.2, we compare the QEMU and PANDA tracers. We can see that the QEMU tracer is significantly slower than PANDA. There is a slowdown of 2.4× for tracing the test case, of 5.8× for recovery tracing, and of 2.9× for state extraction.

QEMU and PANDA use different approaches for hooking instructions and accessing guest state. While PANDA has hooks within the core emulation logic and allows direct access to the internal guest CPU state, QEMU's TCG plugins insert additional instructions into the TCG stream and provide a limited plugin API for guest state access [43]. We assume that these differences, combined with the relative novelty of TCG plugins, lead to higher overhead for tracing with TCG plugins than with PANDA.

For this reason, we recommend using the QEMU tracer only for analyzing cross-media file systems, since the PANDA tracer does not support NVMe devices.

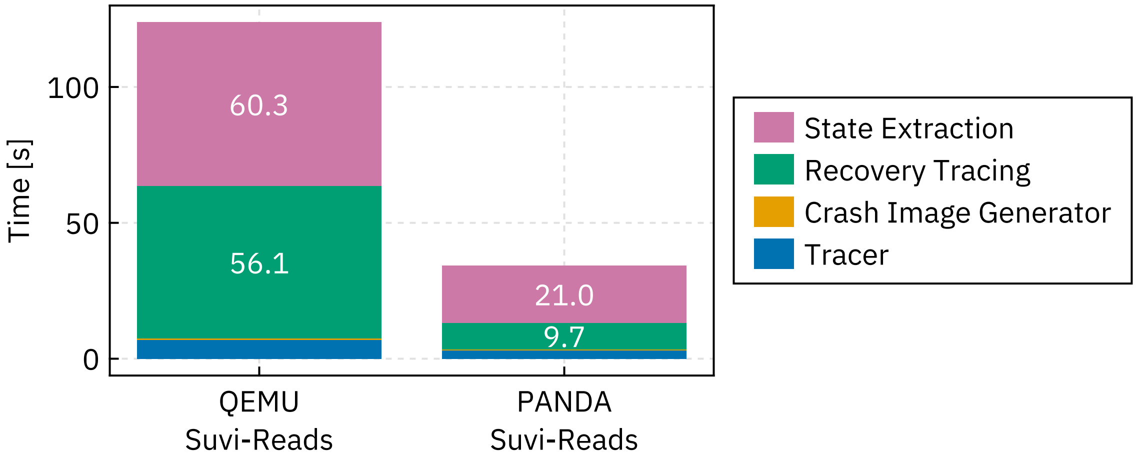


Figure 10.2: pc62 Comparison of analysis times for **Suvi**'s tracer implementations (QEMU and PANDA) with **Suvi-Reads**. Median sequential execution times for the ACE seq2 tests on NOVA.

10.3.3 Parallelization

We evaluate our strategy for parallel test execution in **Suvi** (see Section 9.3) by comparing the real runtime with the sum of the runtime of all pipeline stages. The parallel analysis with PANDA and **Suvi-Reads** finished in 55 minutes. Compared to a sequential runtime of 1968 minutes, this is a 35.9× speedup. The other configurations (QEMU, **Suvi-Fast**) show similar speedup. This is close to the maximum possible parallel speedup on our test system with 36 cores, indicating that our approach to parallelization is successful.

We evaluate speedup for running a single test case by running the "Hello World" test with PANDA and **Suvi-Reads**. **Suvi** generates 28 crash images for this test and finishes analysis in 25 seconds. Compared to a sequential runtime of 107 seconds, this is a 4.3× speedup. As we discuss in Section 9.3, the lower speedup for a single test is expected since only state extraction is run in parallel.

10.4 Results

We analyze the file systems NOVA, NOVA-Fortis, PMFS, WineFS, and ZIL-PMEM with **Suvi**. This thesis presents the first analysis of ZIL-PMEM, while the other file systems have previously been analyzed with Vinter [68] and Chipmunk [81].

Table 10.1 shows an overview of the analysis results. We discuss these results in more detail in the following sections.

Since all tests update timestamps, we only note an atomicity violation due to inatomic timestamp updates for the ctime/mtime test.

10.4.1 NOVA and NOVA-Fortis

Suvi identifies problems in multiple tests, including multiple final states and atomicity violations, and state extraction errors. We identify three root causes.

First, NOVA uses a memcpy function provided by the Linux kernel that uses non-temporal store instructions. Since non-temporal store instructions require an alignment of at least eight bytes, this memcpy function writes any remaining

Test	NOVA	NOVA-Fortis	PMFS	WineFS	ZIL-PMEM
Hello World	S	W	✓	✓	✓
append	✓	✓	✓	✓	✓
atime	✓	✓	✓	✓	✓
ctime/mtime	✓	E	A	A	A
chmod	✓	✓	✓	✓	✓
chown	✓	✓	✓	✓	✓
link	A	E	✓	✓	✓
symlink	S	S	✓	✓	✓
mkdir/rmdir	✓	E	C	✓	✓
rename: overwrite	A	E	C	✓	✓
rename: directory	A	E	✓	✓	✓
rename: long name	E	E	✓	✓	✓
touch	✓	W	✓	✓	✓
long name	E	W	✓	✓	✓
unlink	✓	✓	C	✓	✓
update	A	E	✓	A	✓
fallocate	✓	✓	✓	✓	✓
truncate	✓	E	C	✓	✓

Legend	
✓	Atomic
A	Atomicity violation
S	Multiple final states
E	State extraction error
W	Write error after recovery
C	Recovery crash

Table 10.1: Analysis results of testing PM file systems with the Vinter test cases (see Section 10.2.1).

unaligned bytes with regular store instructions, but does not flush these bytes from the caches.

These unflushed bytes result in multiple final states in **Suvi**'s analysis. For the "Hello World" test, we observe one extra state with truncated file contents. We show **Suvi**'s analysis output for this test case in more detail in Section 8.6.2. Similarly, **Suvi** detects truncated symlink targets and file names, which causes state extraction errors.

This bug is one where **Suvi**'s improved PM simulation results in different results. When analyzing this bug with Vinter, we discussed additional states where some, but not all of the unaligned bytes are present [68, §5.3.1]. **Suvi** correctly does not generate these states. As the file only appears after a later metadata update, x86 global store order enforces that all previous regular stores must have finished.

Second, NOVA does not update the hard link counter atomically as new hard links are created.

Third, rename operations have a logic bug that makes them inatomic. NOVA first removes the old source and target directory entries before creating a new entry for the rename target. **Suvi** therefore discovers states during rename operations where the file is missing completely.

NOVA-Fortis extends NOVA with data protection mechanisms, including checksums and error correcting codes. These features allow NOVA-Fortis to recover some missing unaligned bytes, like in the “Hello World” test. However, symlinks are not protected and show the same multiple final states as with NOVA.

With NOVA-Fortis, several tests report errors during state extraction or failing write operations after recovery. The root cause for these issues is that NOVA-Fortis does not always update checksums atomically. Files with a partially-written checksum report errors during read and write operations after recovery.

10.4.2 PMFS and WineFS

As we describe in Section 2.3.1, PMFS uses `cmpxchg16b` for atomic 16-byte updates, which actual PM hardware does not support. In our analysis with **Suvi**, this shows in inatomic timestamp updates and data loss in truncate operations. These problems are shared with WineFS, which builds on PMFS.

PMFS has another problem where certain updates to the filesystem super block trigger an assertion during recovery. Such a file system is then broken and cannot be mounted anymore. This bug was fixed in WineFS.

As detected by the “update” test, WineFS introduced an atomicity bug with write operations. PMFS performs a write that overwrites existing file data byte-by-byte rather than replacing the data atomically.

10.4.3 ZIL-PMEM

ZIL-PMEM is the only cross-media file system in our test set, which we test with the QEMU tracer (see Section 9.1). We do not find any crash consistency bugs in ZIL-PMEM.

The only problems that **Suvi** reports are by design of ZFS’s ZIL recovery. When the recovery modifies a file with logged updates, it also updates the modification timestamp of that file to the time of recovery. We do not consider this a crash consistency bug, since the file system is not wrong in noting that a modification was performed at recovery time.

10.5 Persistent Caches

The file systems analyzed in the previous section were designed for PM systems with volatile caches. With eADR, some PM systems have persistent caches, which eliminates the need for cache flushes (see Section 2.4.1). However, even with persistent caches, crash consistency remains a challenge. Non-temporal stores remain weakly ordered and require memory fences. Finally, intermittent invalid states can occur during regular execution as a result of logic bugs.

We assess the effect of persistent caches on file system crash consistency by repeating the file system tests from the previous section with **Suvi**’s eADR mode (see Section 8.3.6). In Table 10.2, we highlight tests where the results improve with persistent caches.

Test	NOVA	NOVA-Fortis	PMFS	WineFS	ZIL-PMEM
write	✓	✓	✓	✓	✓
ctime/mtime	✓	E	A	A	A
link	A	E	✓	✓	✓
symlink	✓	✓	✓	✓	✓
mkdir/rmdir	✓	E	C	✓	✓
rename: overwrite	A	E	C	✓	✓
rename: directory	A	E	✓	✓	✓
rename: long name	A	E	✓	✓	✓
touch	✓	✓	✓	✓	✓
long name	✓	✓	✓	✓	✓
unlink	✓	✓	C	✓	✓
update	✓	✓	✓	A	✓
truncate	✓	E	C	✓	✓

Legend	
✓	Atomic
A	Atomicity violation
E	State extraction error
C	Recovery crash

Table 10.2: Analysis results of testing PM file systems with persistent caches (eADR). We only include tests that had failures with volatile caches and show identical results in a lighter color.

Only NOVA and NOVA-Fortis show different results. With persistent caches, the missing cache flush in the memcpy function for unaligned data is no longer necessary. Test failures that originate from that bug therefore disappear.

All other bugs remain with persistent caches, including atomicity bugs in NOVA and WineFS, state extraction errors in NOVA-Fortis, and crashes in PMFS.

10.6 Discussion

Short and simple tests are often sufficient for finding crash consistency bugs in PM file systems. Since these file systems promise crash-atomic file system operations at low latency, their designs often include complex protocols for updating metadata. If there are bugs in these update protocols, **Suvi**'s record-and-replay approach can usually discover them from a single execution.

Persistent caches with eADR solve few real-world crash consistency bugs. Correct use of memory fences remains necessary for weakly-ordered NT stores, which are often preferable for performance (see Section 3.3). Additionally, logic bugs constitute the majority of the crash consistency bugs that we observe.

The testing effort with **Suvi** is low. Running **Suvi** on the manually written Vinter test cases takes only a few minutes. With the aid of **Suvi**'s tools for state analysis and trace debugging, an initial analysis of a file system takes less than one hour.

Still, we observe that crash consistency is not an evaluated goal of recent research file systems. As we describe in Section 2.3, the implementations of recently published file systems have major issues that are apparent without using a crash consistency testing tool. We hope that our research motivates future file system researchers and developers to properly implement and evaluate the crash consistency of their file systems.

Chapter 11

Conclusion

Persistent memory (PM) offers new opportunities for file systems by enabling direct access to storage with low latency, but it also introduces challenges for file system design and implementation. In particular, synchronous and fine-grained access to PM differs significantly from traditional asynchronous block interfaces to storage. In this thesis, we addressed two such challenges, with consequences for efficiency and correctness.

As a first challenge, we identified efficiency problems caused by synchronous access to PM from the CPU. If PM accesses are delayed due to overload, the CPU stalls, wasting CPU time and energy. With Intel Optane PM, such overload arises quickly under parallel load. PM software must therefore limit parallelism when accessing PM. However, CPU and energy efficiency are rarely explicit goals for PM file systems.

In Chapter 4, we introduced efficiency metrics that capture the CPU and energy cost of accessing data from a file system. We evaluated these metrics on multiple PM file systems, and found that most PM file systems do not limit parallel access to PM. These file systems therefore waste CPU time and energy under parallel load.

Beyond quantifying problematic behavior, our work also enables efficiency comparisons between different file systems. For file systems that include measures to limit parallel PM access, we found that higher throughput does not always imply better efficiency. We expect that our work will enable future file systems to include CPU and energy efficiency among their goals.

We then proposed measures to mitigate PM overload from parallel accesses. In Chapter 5, we described three approaches for controlling parallelism within PM file systems. The key idea of these approaches is to eliminate expensive on-CPU waiting by blocking processes during overload and allowing other processes to run.

Userspace processes can also access PM directly by requesting a memory mapping to PM from the file system. After providing such a mapping, the operating system has no further insight into or control over direct PM accesses. In Chapter 6, we proposed an approach for accounting direct PM accesses with association to individual processes. We showed that our appproach provides accurate throughput estimates at low latency. With the accounting information, the operating system can detect and mitigate PM overload from userspace accesses. We introduced a scheduling approach that performs automatic core specialization for PM processes and showed that this approach can prevent PM overload.

The second challenge for PM file systems is correctness in the presence of crashes. Volatile data in the CPU's write path is lost in the event of a crash, which can lead to data corruption. PM applications must therefore carefully manage their modifications through the use of PM primitives such as cache flushes and memory fences.

In Chapter 8 of this thesis, we introduced **Suvi**, an approach to crash consistency testing for PM file systems. **Suvi** is a black-box testing approach that traces the PM interaction of a file system in a virtual machine. **Suvi** then replays the trace, simulates the PM write path, and generates crash images, which represent possible PM contents in the event of a crash. From the crash images, **Suvi** can automatically detect crash consistency bugs such as atomicity violations.

Suvi improves upon previous approaches to crash consistency testing in multiple ways. It supports crash consistency testing for cross-media file systems by tracing and simulating NVMe storage devices. It uses an improved simulation of x86 crash consistency semantics that models x86 global store order more accurately than previous works. **Suvi** includes three heuristics for fast and targeted crash image generation. Our strategy for managing memory images allows **Suvi** to handle large crash images efficiently. Finally, **Suvi**'s analysis tools provide automatic detection of atomicity bugs and assist in determining their root cause.

In Chapter 10, we analyzed multiple PM file systems with **Suvi** using both manually-written and automatically-generated test cases. Finally, we discussed several bugs that **Suvi** detected in these file systems.

In combination, the contributions of this thesis provide a foundation for more correct and more efficient future PM file systems.

11.1 Outlook

This thesis was largely motivated by, and evaluated on, Intel Optane PM. Even though Intel canceled its Optane PM product line, we expect that the contributions of this thesis will remain relevant for future PM technologies. In particular, the CXL interface, supported by modern Intel and AMD server systems, allows independent development of new PM technologies.

Analysis of CXL devices has shown that performance degradation due to overload remains a concern [114]. Our metrics for file system efficiency can guide the design and implementation of future CXL-based PM file systems.

In Chapter 5, we explored DMA offloading as a mitigation for parallel PM accesses. Upcoming CXL-based hybrid SSDs can natively provide an asynchronous DMA interface as an alternative to synchronous direct access [113, 130]. However, managing these two access modes from the operating system remains an active area of research [50, 77].

In the area of PM access accounting, CXL enables new opportunities by allowing innovation at the device level. With support from the CXL device, more accurate accounting at lower overhead becomes possible. We outline such an approach in Section 6.8.

Finally, crash consistency testing continues to be important for CXL-based PM applications and file systems. The crash consistency semantics of x86 systems do not change with CXL. Therefore, **Suvi** can be used to analyze future PM file systems for CXL systems.

References

[1] 2014. peichen-cs/aerie. Retrieved June 18, 2025 from https://github.com/peichen-cs/aerie

[2] 2021. IPMCTL User Guide. Retrieved August 4, 2025 from https://docs.pmem.io/ipmctl-user-guide

[3] 2022. NDCTL User Guide. Retrieved August 4, 2025 from https://docs.pmem.io/ndctl-user-guide

[4] 2023. NVM Express NVM Command Set Specification 1.0d. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-NVM-Command-Set-Specification-1.0d-2023.12.28-Ratified.pdf

[5] 2024. NVM Express Base Specification 2.0d. Retrieved from https://nvmexpress.org/wp-content/uploads/NVM-Express-Base-Specification-2.0d-2024.01.11-Ratified.pdf

[6] 2025. NVSL/linux-nova. Retrieved June 12, 2025 from https://github.com/NVSL/linux-nova

[7] 2025. google/syzkaller. Retrieved August 12, 2025 from https://github.com/google/syzkaller

[8] Intel® Optane™ Persistent Memory 100 Series Product Specifications. Retrieved August 4, 2025 from https://www.intel.com/content/www/us/en/products/sku/series/190349/intel-optane-persistent-memory-100-series.html

[9] Intel® Optane™ Persistent Memory 200 Series Product Specifications. Retrieved August 4, 2025 from https://www.intel.com/content/www/us/en/products/sku/series/203877/intel-optane-persistent-memory-200-series.html

[10] Intel® Optane™ Persistent Memory 300 Series Product Specifications. Retrieved August 4, 2025 from https://www.intel.com/content/www/us/en/products/sku/series/213689/intel-optane-persistent-memory-300-series.html

[11] Linux NVDIMM documentation. Retrieved August 5, 2025 from https://nvdimm.docs.kernel.org/

[12] The Linux Kernel Archives. Retrieved May 20, 2025 from https://kernel.org/

[13] mmap(2) - Linux manual page. Retrieved June 3, 2025 from https://man7.org/linux/man-pages/man2/mmap.2.html

[14] POSIX.1-2024. Retrieved from https://pubs.opengroup.org/onlinepubs/9799919799/

[15] nvme-cli: NVMe management command line interface. Retrieved August 22, 2024 from https://github.com/linux-nvme/nvme-cli

[16] proc_stat(5) - Linux manual page. Retrieved July 18, 2025 from https://www.man7.org/linux/man-pages/man5/proc_stat.5.html

[17] open(2) - Linux manual page. Retrieved July 30, 2025 from https://man7.org/linux/man-pages/man2/open.2.html

[18] perf_event_open(2) - Linux manual page. Retrieved August 8, 2025 from https://man7.org/linux/man-pages/man2/perf_event_open.2.html

[19] PMDK: Persistent Memory Development Kit. Retrieved August 11, 2025 from https://github.com/pmem/pmdk

[20] What's New In Python 3.3. Retrieved from https://docs.python.org/3/whatsnew/3.3.html

[21] ioctl_ficlonerange(2) - Linux manual page. Retrieved from https://man7.org/linux/man-pages/man2/ioctl_ficlone.2.html

[22] BusyBox. Retrieved from https://busybox.net/

[23] Soramichi Akiyama and Takahiro Hirofuchi. 2017. 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*, June 2017. ACM, Washingon DC USA, 1–8. https://doi.org/10.1145/3095770.3095773

[24] Lukas Alt, Anara Kozhokanova, Thomas Ilsche, Christian Terboven, and Matthias S. Mueller. 2024. An Experimental Setup to Evaluate RAPL Energy Counters for Heterogeneous Memory. In *Proceedings of the 15th ACM/SPEC International Conference on Performance Engineering*, May 2024. ACM, London United Kingdom, 71–82. https://doi.org/10.1145/3629526.3645052

[25] Thomas E Anderson, Simon Peter, Marco Canini, Jongyul Kim, Dejan Kostic, Youngjin Kwon, Waleed Reda, Henry N Schuh, and Emmett Witchel. 2022. Assise: Performance and Availability via Client-local NVM in a Distributed File System. *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (November 2022), 1011–1027. Retrieved from https://www.usenix.org/conference/osdi20/presentation/anderson

[26] Jean-Philippe Aumasson and Daniel J. Bernstein. 2012. SipHash: A Fast Short-Input PRF. *Progress in Cryptology - INDOCRYPT 2012 7668*, 489–508. https://doi.org/10.1007/978-3-642-34931-7_28

[27] Jens Axboe. Flexible I/O Tester. Retrieved July 30, 2025 from https://github.com/axboe/fio

[28] Fabrice Bellard. 2005. QEMU, a fast and portable dynamic translator. In *Proceedings of the annual conference on USENIX Annual Technical Conference (ATEC '05)*, April 2005. USENIX Association, USA, 41.

[29] Alex Bennée. 2022. QEMU TCG Plugins. Retrieved from https://qemu.eu/doc/8.0/devel/tcg-plugins.html

[30] Florian Bernd and Joel Höner. zydis | The ultimate X86 & X86-64 disassembler library. Retrieved from https://zydis.re/

[31] Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. 2003. The Zettabyte File System. (2003).

[32] Ilia Bozhinov. 2022. Reducing Synchronous Write Latency With a PMEM Write Cache in the Device Mapper Layer. Doctoral dissertation. Retrieved from https://os.itec.kit.edu/downloads/2022_BA_Bozhinov_DMWriteCache.pdf

[33] Neil Brown. 2017. A block layer introduction part 1: the bio layer. *LWN.net* (October 2017). Retrieved October 9, 2025 from https://lwn.net/Articles/736534/

[34] CLEAResult. 80 PLUS certification specifications and ratings. Retrieved July 17, 2025 from https://www.clearesult.com/80plus/program-details#program-details-table

[35] The const generics project group. 2021. Const generics MVP hits beta!. Retrieved August 4, 2025 from https://blog.rust-lang.org/2021/02/26/const-generics-mvp-beta/

[36] Jonathan Corbet. 2019. Some slow progress on get_user_pages(). *LWN.net* (April 2019). Retrieved August 1, 2025 from https://lwn.net/Articles/784574/

[37] Spencer Desrochers, Chad Paradis, and Vincent M. Weaver. 2016. A Validation of DRAM RAPL Power Measurements. In *Proceedings of the Second International Symposium on Memory Systems*, October 2016. ACM, Alexandria VA USA, 455–470. https://doi.org/10.1145/2989081.2989088

[38] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable Reverse Engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, December 2015. ACM, Los Angeles CA USA, 1–11. https://doi.org/10.1145/2843859.2843867

[39] Eelco Dolstra. 2006. The purely functional software deployment model. Doctoral dissertation. Retrieved from https://edolstra.github.io/pubs/phd-thesis.pdf

[40] Srikar Dronamraju. Uprobe-tracer: Uprobe-based Event Tracing — The Linux Kernel documentation. Retrieved from https://www.kernel.org/doc/html/v6.12/trace/uprobetracer.html

[41] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*, 2014. ACM, New York, NY, USA, 15:1–15:15. https://doi.org/10.1145/2592798.2592814

[42] Michael J Eager. 2012. *Introduction to the DWARF Debugging Format*. Retrieved from https://dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf

[43] Andrew Fasano. 2022. TCG Plugins vs PANDA Plugins. Retrieved from https://github.com/panda-re/panda/wiki/TCG-Plugins-vs-PANDA-Plugins

[44] UEFI Forum. 2025. Advanced Configuration and Power Interface (ACPI) Specification. Retrieved from https://uefi.org/sites/default/files/resources/ACPI_Spec_6.6.pdf

[45] Xinwei Fu, Wook-Hee Kim, Ajay Paddayuru Shreepathi, Mohannad Ismail, Sunny Wadkar, Dongyoon Lee, and Changwoo Min. 2021. Witcher: Systematic Crash Consistency Testing for Non-Volatile Memory Key-Value Stores. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, October 2021. ACM, Virtual Event Germany, 100–115. https://doi.org/10.1145/3477132.3483556

[46] João Gonçalves, Miguel Matos, and Rodrigo Rodrigues. 2023. Mumak: Efficient and Black-Box Bug Detection for Persistent Memory. In *Proceedings of the Eighteenth European Conference on Computer Systems*, May 2023. ACM, Rome Italy, 734–750. https://doi.org/10.1145/3552326.3587447

[47] Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. 2020. Automatic Core Specialization for AVX-512 Applications. In *Proceedings of the 13th ACM International Systems and Storage Conference*, May 2020. ACM, Haifa Israel, 25–35. https://doi.org/10.1145/3383669.3398282

[48] Brendan Gregg. 2024. The Return of the Frame Pointers. Retrieved March 7, 2025 from https://www.brendangregg.com/blog/2024-03-17/the-return-of-the-frame-pointers.html

[49] Hao Guo and Youyou Lu. 2025. Achieving Low-Latency Graph-Based Vector Search via Aligning Best-First Search Algorithm with SSD. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, 2025. Retrieved from https://www.usenix.org/conference/osdi25/presentation/guo

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

[51] Adnan Hasnat and Shoaib Akram. 2025. SPIRIT: Scalable and Persistent In-Memory Indices for Real-Time Search. *ACM Transactions on Architecture and Code Optimization* 22, 1 (March 2025), 1–26. https://doi.org/10.1145/3703351

[52] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process isolation for High-Throughput data plane libraries. In *2019 USENIX annual technical conference (USENIX ATC 19)*, July 2019. USENIX Association, Renton, WA, 489–504. Retrieved from http://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[53] Brook Heisler. *Criterion.rs*. Retrieved August 4, 2025 from https://bheisler.github.io/criterion.rs/book/

[54] Tejun Heo and Florian Mickler. 2010. Workqueue — The Linux Kernel documentation. Retrieved August 1, 2025 from https://www.kernel.org/doc/html/v6.12/core-api/workqueue.html

[55] Intel. 2017. Intel® Xeon® Processor Scalable Memory Family Uncore Performance Monitoring Reference Manual. Retrieved August 5, 2025 from https://www.intel.com/content/www/us/en/content-details/671389/intel-xeon-processor-scalable-memory-family-uncore-performance-monitoring-reference-manual.html

[56] Intel. 2018. Power Supply Design Guide for Desktop Platform Form Factors. Retrieved July 17, 2025 from https://www.intel.com/content/dam/www/public/us/en/documents/design-guides/resellers-power-supply-design-guide-changes.pdf

[57] Intel. 2019. Second Generation Intel® Xeon® Scalable Processors Datasheet, Volume Two: Registers. (April 2019). Retrieved from https://www.intel.com/content/dam/www/public/us/en/documents/datasheets/2nd-gen-xeon-scalable-datasheet-vol-2.pdf

[58] Intel. 2020. Persistent Memory FAQ. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/troubleshooting/persistent-memory-faq.html

[59] Intel. 2021. eADR: New Opportunities for Persistent Memory Applications. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html

[60] Intel. 2021. 3rd Gen Intel® Xeon® Processor Scalable Family, Codename Ice Lake Uncore Performance Monitoring Reference Manual. Retrieved August 5, 2025 from https://www.intel.com/content/www/us/en/content-details/679093/3rd-gen-intel-xeon-processor-scalable-family-codename-ice-lake-uncore-performance-monitoring-reference-manual.html

[61] Intel. 2024. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*.

[62] Intel. Fast memcpy with SPDK and Intel® I/OAT DMA Engine. Retrieved August 2, 2025 from https://www.intel.com/content/www/us/en/developer/articles/technical/fast-memcpy-using-spdk-and-ioat-dma-engine.html

[63] Intel. Perfmon Events for 2nd Generation Intel® Xeon® Processor Scalable Family based on Cascade Lake. Retrieved August 5, 2025 from https://perfmon-events.intel.com/cascadelake_server.html

[64] Abdullah Al Raqibul Islam and Dong Dai. 2023. DGAP: Efficient Dynamic Graph Analysis on Persistent Memory. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2023. ACM, Denver CO USA, 1–13. https://doi.org/10.1145/3581784.3607106

[65] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. 2019. Evaluating File System Reliability on Solid State Drives. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019. 783–798. Retrieved May 22, 2025 from https://www.usenix.org/conference/atc19/presentation/jaffer

[66] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, October 2021. ACM, Virtual Event Germany, 804–818. https://doi.org/10.1145/3477132.3483567

[67] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, October 2019. ACM, Huntsville Ontario Canada, 494–508. https://doi.org/10.1145/3341301.3359631

[68] Samuel Kalbfleisch, Lukas Werling, and Frank Bellosa. 2022. Vinter: Automatic Non-Volatile Memory Crash Consistency Testing for Full Systems. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022. 933–950. Retrieved from https://www.usenix.org/conference/atc22/presentation/werling

[69] Samuel Kalbfleisch. 2021. Automatic Non-Volatile Memory Crash Consistency Testing for Full Systems. Doctoral dissertation.

[70] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. Kernel Probes (Kprobes) — The Linux Kernel documentation. Retrieved from https://www.kernel.org/doc/html/v6.12/trace/kprobes.html

[71] The kernel development community. The kernel's command-line parameters — The Linux Kernel documentation. Retrieved August 5, 2025 from https://www.kernel.org/doc/html/v6.12/admin-guide/kernel-parameters.html

[72] The kernel development community. BTT - Block Translation Table — The Linux Kernel documentation. Retrieved June 2, 2025 from https://www.kernel.org/doc/html/v6.12/driver-api/nvdimm/btt.html

[73] The kernel development community. Direct Access for files — The Linux Kernel documentation. Retrieved June 2, 2025 from https://www.kernel.org/doc/html/v6.12/filesystems/dax.html

[74] The kernel development community. Device Mapper — The Linux Kernel documentation. Retrieved August 3, 2025 from https://www.kernel.org/doc/html/v6.12/admin-guide/device-mapper/index.html

[75] The kernel development community. Power Capping Framework — The Linux Kernel documentation. Retrieved July 21, 2025 from https://www.kernel.org/doc/html/v6.12/power/powercap/powercap.html

[76] The kernel development community. DMAEngine documentation — The Linux Kernel documentation. Retrieved August 2, 2025 from https://www.kernel.org/doc/html/v6.12/driver-api/dmaengine/index.html

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

[78] Steve Klabnik, Carol Nichols, and Chris Krycho. 2025. *The Rust Programming Language*. Retrieved from https://doc.rust-lang.org/stable/book/

[79] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. 2017. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, October 2017. ACM, Shanghai China, 460–477. https://doi.org/10.1145/3132747.3132770

[80] Philip Lantz, Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. 2014. Yat: A Validation Framework for Persistent Memory Software. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014. 433–438. Retrieved January 30, 2024 from https://www.usenix.org/conference/atc14/technical-sessions/presentation/lantz

[81] Hayley LeBlanc, Shankara Pailoor, Om Saran K R E, Isil Dillig, James Bornholt, and Vijay Chidambaram. 2023. Chipmunk: Investigating Crash-Consistency in Persistent-Memory File Systems. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys '23)*, May 2023. Association for Computing Machinery, New York, NY, USA, 718–733. https://doi.org/10.1145/3552326.3567498

[82] Hayley LeBlanc. 2023. utsaslab/chipmunk: Tool for checking crash-consistency for persistent-memory file systems (EuroSys 23). Retrieved August 12, 2025 from https://github.com/utsaslab/chipmunk

[83] Endian Li, Shushu Yi, Li Peng, Qiao Li, Diyu Zhou, Zhenlin Wang, Xiaolin Wang, Bo Mao, Yingwei Luo, Ke Zhou, and Jie Zhang. 2025. SPDK+: Low Latency or High Power Efficiency? We Take Both. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems*, July 2025. ACM, Boston MA USA, 17–23. https://doi.org/10.1145/3736548.3737824

[84] Zhen Lin, Lingfeng Xiang, Jia Rao, and Hui Lu. 2023. P2CACHE: Exploring tiered memory for In-Kernel file systems caching. In *2023 USENIX annual technical conference (USENIX ATC 23)*, July 2023. USENIX Association, Boston, MA, 801–815. Retrieved from https://www.usenix.org/conference/atc23/presentation/lin

[85] Sihang Liu, Korakit Seemakhupt, Yizhou Wei, Thomas Wenisch, Aasheesh Kolli, and Samira Khan. 2020. Cross-Failure Bug Detection in Persistent Memory Programs. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '20)*, March 2020. Association for Computing Machinery, Lausanne, Switzerland, 1187–1202. https://doi.org/10.1145/3373376.3378452

[86] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. 2019. PMTest: A Fast and Flexible Testing Framework for Persistent Memory

Programs. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, April 2019. ACM, Providence RI USA, 411–425. https://doi.org/10.1145/3297858.3304015

[87] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2025. System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models) Version 1.0. Retrieved from https://gitlab.com/x86-psABIs/x86-64-ABI

[88] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. 2005. Pin: building customized program analysis tools with dynamic instrumentation. *ACM SIGPLAN Notices* 40, 6 (June 2005), 190–200. https://doi.org/10.1145/1064978.1065034

[89] Tobias Mann. 2022. Why Intel killed its Optane memory business. Retrieved August 4, 2025 from https://www.theregister.com/2022/07/29/intel_optane_memory_dead/

[90] Peter Maucher, Lennard Kittner, Nico Rath, Gregor Lucka, Lukas Werling, Yussuf Khalil, Thorsten Gröninger, and Frank Bellosa. 2024. Full-Scale File System Acceleration on GPU. In *Tagungsband des FG-BS Frühjahrstreffens 2024*, 2024. https://doi.org/10.18420/FGBS2024F-03

[91] Peter Maucher. 2022. GPU4FS: A Graphics Processor-Accelerated File System. Master's thesis. Retrieved from https://os.itec.kit.edu/downloads/2022_MA_Maucher_GPU4FS.pdf

[92] Jordan McLeod. 2020. Whole-System Emulation and Analysis with Rust and PANDA. Retrieved from https://panda-re.mit.edu/blog/panda-rs/

[93] Jayashree Mohan, Ashlie Martinez, Soujanya Ponnapalli, Pandian Raju, and Vijay Chidambaram. 2018. Finding Crash-Consistency Bugs with Bounded Black-Box Crash Testing. *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)* (October 2018), 33–50. Retrieved from https://www.usenix.org/conference/osdi18/presentation/mohan

[94] Aris Mpitziopoulos. 2023. Powenetics V2 - Power Measurements Device - Review. Retrieved July 16, 2025 from https://hwbusters.com/psus/powenetics-v2-power-measurements-device-review/

[95] Vijay Nagarajan, Daniel J. Sorin, Mark D. Hill, and David A. Wood. 2020. *A Primer on Memory Consistency and Cache Coherence*. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-031-01764-3

[96] Ian Neal, Ben Reeves, Ben Stoler, Andrew Quinn, Youngjin Kwon, Simon Peter, and Baris Kasikci. 2020. AGAMOTTO: How Persistent is your Persistent Memory Application?. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020. 1047–1064. Retrieved January 30, 2024 from https://www.usenix.org/conference/osdi20/presentation/neal

[97] Aleix Roca Nonell, Balazs Gerofi, Leonardo Bautista-Gomez, Dominique Martinet, Vicenç Beltran Querol, and Yutaka Ishikawa. 2018. On the Applicability of PEBS based Online Memory Access Tracking for Heterogeneous

Memory Management at Scale. In *Proceedings of the Workshop on Memory Centric High Performance Computing*, November 2018. ACM, Dallas TX USA, 50–57. https://doi.org/10.1145/3286475.3286477

[98] Jack O'Connor, Jean-Philippe Aumasson, Samuel Neves, and Zooko Wilcox-O'Hearn. 2021. BLAKE3: one function, fast everywhere. Retrieved from https://blake3.io/

[99] Thomas-Christian Oder. 2023. Fast Persistent Memory Crash Consistency Analysis based on Virtual Machines. Bachelor Thesis. Retrieved from https://os.itec.kit.edu/downloads/2023_BA_Oder_Fast_Crash_Consistency.pdf

[100] Jinyoung Oh and Youngjin Kwon. 2021. Persistent memory aware performance isolation with dicio. In *Proceedings of the 12th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '21)*, August 2021. Association for Computing Machinery, New York, NY, USA, 97–105. https://doi.org/10.1145/3476886.3477517

[101] Shweta Pandey and Arkaprava Basu. 2025. H-Rocks: CPU-GPU accelerated Heterogeneous RocksDB on Persistent Memory. *Proceedings of the ACM on Management of Data* 3, 1 (February 2025), 1–28. https://doi.org/10.1145/3709694

[102] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles CD-ROM*, October 2021. ACM, Virtual Event Germany, 392–407. https://doi.org/10.1145/3477132.3483550

[103] Rittal. 7979402 PDU managed. Retrieved July 17, 2025 from https://www.rittal.com/de-de/products/PG20231215POW101/PG20240419STR003/PRO115366?variantId=7979402

[104] Marc Rittinghaus. 2019. SimuBoost: Scalable Parallelization of Functional System Simulation. Doctoral dissertation. https://doi.org/10.5445/IR/1000097700

[105] Daniel Ritz. 2022. Crash Consistency Testing for Block Based File Systems on NVMe Drives. Bachelor Thesis. Retrieved from https://os.itec.kit.edu/downloads/2022_BA_Ritz_NVMe_Crash_Consistency.pdf

[106] Mendel Rosenblum and John K. Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Trans. Comput. Syst.* 10, 1 (February 1992), 26–52. https://doi.org/10.1145/146941.146943

[107] Michael Roth. 2017. [Qemu-stable] [ANNOUNCE] QEMU 2.9.1 Stable released. Retrieved from https://mail.gnu.org/archive/html/qemu-stable/2017-09/msg00019.html

[108] Andy M. Rudoff. 2016. Deprecating the PCOMMIT Instruction. Retrieved from https://www.intel.com/content/www/us/en/developer/articles/technical/deprecate-pcommit-instruction.html

[109] Andy Rudoff. 2020. Re: 8 byte atomicity & larger store operations. Retrieved from https://groups.google.com/g/pmem/c/6_5daOuEI00/m/nY_mtKd0CAAJ

[110] Thomas Schmidt. 2022. Achieving Optimal Throughput for Persistent Memory with Per-Process Accounting. Master's thesis. Retrieved from https://os.itec.kit.edu/downloads/2022_MA_Schmidt_PMEM_Accounting.pdf

[111] Christian Schwarz. 2021. Low-Latency Synchronous IO For OpenZFS Using Persistent Memory. Retrieved from https://os.itec.kit.edu/downloads/2021_MA_Schwarz_SyncIOForZFS.pdf

[112] Harshad Shirwadkar, Saurabh Kadekodi, and Theodore Tso. 2024. FastCommit: resource-efficient, performant and cost-effective file system journaling. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024. 157–171. Retrieved May 22, 2025 from https://www.usenix.org/conference/atc24/presentation/shirwadkar

[113] Mohammadreza Soltaniyeh, Gongjin Sun, Xuebin Yao, Amir Beygi, Ramdas Kachare, Dongwan Zhao, Hingkwan Huen, Andrew Chang, Senthil Murugesapandian, and Caroline Kahn. 2025. Revisiting Memory Hierarchies with CMM-H: Use Device-side Caching to Integrate DRAM and SSD for a Hybrid CXL Memory. In *Proceedings of the 17th ACM Workshop on Hot Topics in Storage and File Systems (HotStorage '25)*, July 2025. Association for Computing Machinery, New York, NY, USA, 45–51. https://doi.org/10.1145/3736548.3737828

[114] Yan Sun, Yifan Yuan, Zeduo Yu, Reese Kuper, Ipoom Jeong, Ren Wang, and Nam Sung Kim. 2023. Demystifying CXL Memory with Genuine CXL-Ready Systems and Devices. Retrieved August 14, 2023 from https://arxiv.org/abs/2303.15375v3

[115] Supermicro. 2010. PWS-502-PQ 80 PLUS Verification and Testing Report. Retrieved July 17, 2025 from https://store.supermicro.com/us_en/pub/media/wysiwyg/productspecs/PWS-502-PQ/SUPER_MICRO_PWS-502-PQ_ECOS_1906_500W_Report.pdf

[116] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Mike Nishimoto, and Geoff Peck. 1996. Scalability in the XFS File System. In *Proceedings of the USENIX 1996 Annual Technical Conference*, January 1996. USENIX Association, San Diego, California.

[117] Vishal Verma. 2014. Using the Block Translation Table for sector atomicity. Retrieved May 20, 2025 from https://pmem.io/blog/2014/09/using-the-block-translation-table-for-sector-atomicity/

[118] Haris Volos, Sanketh Nalli, Sankarlingam Panneerselvam, Venkatanathan Varadarajan, Prashant Saxena, and Michael M. Swift. 2014. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*, 2014. ACM, New York, NY, USA, 14:1–14:14. https://doi.org/10.1145/2592798.2592810

[119] Paul Wedeck. 2023. Analyzing Persistent Memory Crash Consistency of WineFS with Vinter. Bachelor Thesis. Retrieved from https://os.itec.kit.edu/downloads/2023_BA_Wedeck_WineFS_Analysis.pdf

[120] Lukas Werling, Yussuf Khalil, Peter Maucher, Thorsten Gröninger, and Frank Bellosa. 2023. Analyzing and Improving CPU and Energy Efficiency of PM File Systems. In *Proceedings of the 1st Workshop on Disruptive Memory Systems*, October 2023. ACM, Koblenz Germany, 31–37. https://doi.org/10.1145/3609308.3625265

[121] Lukas Werling, Thomas-Christian Oder, Lucas Wäldele, Daniel Ritz, and Frank Bellosa. 2024. Improvements in Crash Consistency Testing for Persistent Memory File Systems. In *Tagungsband des FG-BS Frühjahrstreffens 2024*, 2024. Gesellschaft für Informatik e.V., Bochum, Germany. https://doi.org/10.18420/FGBS2024F-01

[122] Matthew Wilcox. 2020. struct page, the Linux physical page frame data structure. Retrieved August 5, 2025 from https://blogs.oracle.com/linux/post/struct-page-the-linux-physical-page-frame-data-structure

[123] Lucas Wäldele. 2023. Crash Consistency Testing for Cross-Media File Systems using Persistent Memory and NVMe. Bachelor Thesis. Retrieved from https://os.itec.kit.edu/downloads/2023_BA_W%C3%A4ldele_Cross-Media_Crash_Consistency.pdf

[124] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. 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)*, March 2022. Association for Computing Machinery, New York, NY, USA, 488–505. https://doi.org/10.1145/3492321.3519556

[125] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, 2016. 323–338. Retrieved January 7, 2019 from https://www.usenix.org/node/194455

[126] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*, 2017. ACM, New York, NY, USA, 478–496. https://doi.org/10.1145/3132747.3132761

[127] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, February 2020. USENIX Association, Santa Clara, CA, 169–182. Retrieved from https://www.usenix.org/conference/fast20/presentation/yang

[128] Yang Yang, Qiang Cao, Jie Yao, Yuanyuan Dong, and Weikang Kong. 2021. SPMFS: A Scalable Persistent Memory File System on Optane Persistent Memory. In *50th International Conference on Parallel Processing (ICPP 2021)*, August 2021. Association for Computing Machinery, New York, NY, USA, 1–10. https://doi.org/10.1145/3472456.3472503

[129] Jifei Yi, Benchao Dong, Mingkai Dong, and Haibo Chen. 2020. On the precision of precise event based sampling. In *Proceedings of the 11th ACM SIGOPS Asia-*

Pacific Workshop on Systems, August 2020. ACM, Tsukuba Japan, 98–105. https://doi.org/10.1145/3409963.3410490

[130] Jianping Zeng, Shuyi Pei, Da Zhang, Yuchen Zhou, Amir Beygi, Xuebin Yao, Ramdas Kachare, Tong Zhang, Zongwang Li, Marie Nguyen, Rekha Pitchumani, Yang Soek Ki, and Changhee Jung. 2025. Performance Characterizations and Usage Guidelines of Samsung CXL Memory Module Hybrid Prototype. https://doi.org/10.48550/arXiv.2503.22017

[131] Yifeng Zhang, Yanqi Pan, Hao Huang, Yuchen Shan, and Wen Xia. 2025. Overcoming the Last Mile between Log-Structured File Systems and Persistent Memory via Scatter Logging. In *Proceedings of the Twentieth European Conference on Computer Systems*, March 2025. ACM, Rotterdam Netherlands, 1009–1025. https://doi.org/10.1145/3689031.3717488

[132] Shengan Zheng, Morteza Hoseinzadeh, and Steven Swanson. 2019. Ziggurat: a tiered file system for Non-Volatile main memories and disks. In *17th USENIX conference on file and storage technologies (FAST 19)*, February 2019. USENIX Association, Boston, MA, 207–219. Retrieved from https://www.usenix.org/conference/fast19/presentation/zheng

[133] Diyu Zhou, Vojtech Aschenbrenner, Tao Lyu, Jian Zhang, Sudarsun Kannan, and Sanidhya Kashyap. 2023. Enabling High-Performance and Secure User-space NVM File Systems with the Trio Architecture. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, October 2023. Association for Computing Machinery, New York, NY, USA, 150–165. https://doi.org/10.1145/3600006.3613171

[134] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. 2022. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022. 179–193. Retrieved from https://www.usenix.org/conference/osdi22/presentation/zhou-diyu