Copyright

by

Rohan Kadekodi

2023

The Dissertation Committee for Rohan Kadekodi
certifies that this is the approved version of the following dissertation:

Transparently achieving high performance for applications on Persistent Memory

Committee:

Vijay Chidambaram (Supervisor)

Chistopher Rossbach

James Bornholt

Kimberly Keeton

Gregory Ganger

Transparently achieving high performance for applications on Persistent Memory

by

Rohan Kadekodi

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

June 2023

Dedicated to my family for their unconditional love and support.

This wouldn't have been possible without you.

Acknowledgements

Transparently achieving high performance for applications on Persistent Memory

Publication No. ________________________

Rohan Kadekodi, PhD.
The University of Texas at Austin, 2023

Supervisor: Vijay Chidambaram

Persistent Memory (PM) refers to storage class memory that offers non-volatility, low latency, and high bandwidth. Due to its memory semantics, high capacity and non-volatility, PM can be used as storage as well as memory. A number of applications can benefit from PM due to its unique characteristics. Legacy applications that are built for HDDs and SSDs use POSIX system-calls to access data on PM. Newer applications designed specifically to be used on PM access / store data by memory mapping files and performing loads / stores from user-space to PM-resident addresses. Finally, modern data-intensive applications can use high-bandwidth networking to access hundreds of terabytes of data across PM of multiple nodes. This thesis answers the question: *How to achieve high performance for all three classes of applications on PM without application changes?* This is achieved by carefully studying data access patterns and redesigning internals of PM systems; both local and distributed.

We first present SplitFS, a PM file system aimed at accelerating legacy I/O intensive applications using a novel split of responsibilities between a user-space library file system and an existing kernel PM file system. SplitFS handles data operations in user space, and reuses an existing kernel file system for metadata operations. Next, we introduce WineFS, a novel hugepage-aware PM file system aimed at accelerating newer PM applications. WineFS uses an alignment-aware allocation policy and a suitable on-PM layout to preserve hugepages, thus reducing page faults, while achieving high scalability using per-CPU fine-grained journaling. Finally, we present ScaleMem, which provides a distributed DAX-based `mmap()` abstraction to data-intensive PM applications, while transparently managing their data across the PM of multiple servers.

This dissertation showcases techniques for a wide range of applications to achieve high performance on top of faster storage devices. The techniques introduced in this dissertation are not limited to PM, but can be applied to any byte addressable media. With new storage and memory technologies emerging, we believe that the contributions of this thesis can be used as a foundation to build storage systems offering high performance and strong guarantees.

Table of Contents

Acknowledgements **v**

Abstract **vi**

List of Figures **xiv**

List of Tables **xv**

1 Introduction **1**

1.1 Accelerating legacy applications with SplitFS 4
1.2 Accelerating modern PM applications with WineFS 5
1.3 Accelerating big data applications with ScaleMem 6
1.4 Contributions 8
1.5 Overview 9

2 Background **11**

2.1 Storage Class Memory 11
2.2 Intel Optane DC Persistent Memory 12
2.2.1 Characteristics of PM 12
2.2.2 PM Operating Modes 14
2.2.3 PM Performance profile 15
2.3 Direct Access (DAX) 16
2.4 File systems on PM 17
2.4.1 POSIX API 17
2.4.2 File system crash-consistency guarantees 18
2.4.3 Crash consistency mechanisms 19
2.4.4 Existing PM file systems 21
2.5 Memory mapping on PM 22
2.5.1 Page Faults 22
2.5.2 Hugepages 23

2.6 Summary 25

3 Motivation **27**

3.1 How PM is accessed 27

3.1.1 System-call interface 27

3.1.2 Memory-mapped interface 28

3.2 Limitations of existing file systems 28

3.2.1 Software Overheads of POSIX system-call interface . 29

3.2.2 Performance overheads of Memory-mapped interface 30

3.3 Enabling remote DAX memory mappings 33

3.4 Summary 35

4 Achieving high performance for legacy I/O intensive applications **36**

4.1 Rethinking division of responsibilities between user-space and the kernel 37

4.2 SplitFS: a user-kernel hybrid PM file system 38

4.3 SplitFS Goals 38

4.4 SplitFS Modes and Guarantees 39

4.5 SplitFS Design 41

4.5.1 Handling reads, Overwrites and Appends 46

4.6 SplitFS Implementation 47

4.6.1 Tunable Parameters 50

4.6.2 Security 50

4.7 Discussion 51

4.8 SplitFS Evaluation 52

4.8.1 Experimental Setup 53

4.8.2 Workloads 54

4.8.3 Correctness and recovery 55

4.8.4 SplitFS system call overheads 56

4.8.5 SplitFS performance breakdown 58

4.8.6 Performance on different IO patterns 59

4.8.7 Reducing software overhead 61

4.8.8 Performance on data intensive workloads 62

4.8.9 Performance on metadata intensive workloads 65

4.8.10 Resource consumption 66
4.9 Conclusion 66

5 Achieving high performance for modern PM applications 68
5.1 Building a hugepage-aware file system 68
5.2 WineFS: A hugepage-aware file system that ages gracefully 70
5.3 WineFS Goals 70
5.4 WineFS Design 71
5.4.1 Overview 71
5.4.2 Guarantees 73
5.4.3 Hugepage Awareness 73
5.4.4 Ensuring good performance for applications using POSIX system calls 76
5.5 WineFS Implementation 78
5.6 Discussion 83
5.7 WineFS Evaluation 85
5.7.1 Experimental Setup 86
5.7.2 Crash Consistency & POSIX Compliance 87
5.7.3 Read and Write Throughput 88
5.7.4 Performance for memory-mapped access mode 91
5.7.5 Performance for system-call access mode 95
5.7.6 Scalability 97
5.7.7 NUMA-awareness 97
5.7.8 Resource Consumption 99
5.7.9 Summary 99
5.8 Conclusion 100

6 Supporting and accelerating big data PM applications 101
6.1 Need for a remote DAX `mmap()` abstraction 102
6.2 ScaleMem: Enabling Distributed DAX Memory-Mapping for Persistent Memory 103
6.3 ScaleMem Goals 104
6.3.1 ScaleMem Setup and Usage Scenarios 104
6.4 ScaleMem Design 105
6.4.1 Overview 107

6.4.2 ScaleMem Mechanisms 109
6.4.3 ScaleMem Mechanisms for POSIX system calls . . 112
6.4.4 ScaleMem Policies 113
6.5 ScaleMem Implementation 115
6.6 Discussion 117
6.7 ScaleMem Evaluation 119
6.7.1 ScaleMem Remote page fault cost. 121
6.7.2 ScaleMem Techniques 122
6.7.3 Microbenchmarks 125
6.7.4 Scalability in capacity 127
6.7.5 Real-world Applications 128
6.7.6 CPU and Memory utilization 131
6.8 Conclusion 132

7 Discussion **133**
7.1 Modifying existing file systems 133
7.1.1 Leveraging ext4 DAX's maturity in SplitFS 134
7.1.2 Modifying ext4 DAX instead of building WineFS . . 135
7.2 Comparing SplitFS and WineFS 136
7.2.1 Design Goals and Trade Offs 137
7.2.2 Using SplitFS with WineFS 138
7.3 Using ScaleMem with WineFS vs ext4 DAX 139
7.4 Applicability to other storage media 140
7.4.1 Block-based storage devices 140
7.4.2 Byte addressable persistent media 142
7.5 Summary 143

8 Related Work **144**
8.1 PM file systems 144
8.2 Hugepage-friendliness and file system aging 146
8.3 Distributed file and memory management 148

9 Future Work **151**
9.1 eBPF for memory management 151
9.2 Leveraging other memory and storage technologies 153
9.3 Building systems that provide stronger guarantees 154

9.4 Summary 154

10 Lessons Learned and Conclusions 156

10.1 Summary 157

10.1.1 Improving performance of legacy I/O intensive applications 157

10.1.2 Improving performance of modern PM applications . 158

10.1.3 Improving performance of big-data applications . . . 159

10.2 Lessons Learned 160

10.2.1 Transitioning from block addressability to byte addressability 160

10.2.2 Designing application-specific policies 161

10.2.3 Using DRAM effectively 162

10.3 Closing Words 162

References 164

List of Figures

2.1 Persistent Memory layout 13
2.2 Memory-mapping overhead 23
2.3 Overhead due to TLB misses 24

3.1 Free space fragmentation 30
3.2 Impact of aging 31
3.3 Memory Manager Bottleneck 34

4.1 SplitFS Overview 42
4.2 *relink* steps 44
4.3 SplitFS techniques 58
4.4 Performance on different IO patterns 59
4.5 Software overhead in applications 61
4.6 Data Intensive Workloads 63
4.7 Metadata Intensive Workloads 63

5.1 WineFS Architecture 71
5.2 Read and write throughput for system calls and memory-mapped access 89
5.3 Performance on aged file systems 90
5.4 Latency distribution for P-ART lookups 93
5.5 Performance of applications using POSIX system calls on clean file systems 94
5.6 Microbenchmark: Scalability 97
5.7 Performance across NUMA nodes 98

6.1 ScaleMem overview 106
6.2 Page Fault Latency Breakdown 121
6.3 Effect of Fast Reallocation 125
6.4 Reads/Writes performance comparison with NFS 126
6.5 ScaleMem scaling with nodes 128

6.6 Application Performance Comparison 129

9.1 Userfaultfd path 152

List of Tables

2.1 PM properties 15

3.1 Software Overhead 29

4.1 SplitFS Modes 39
4.2 SplitFS Techniques 41
4.3 Applications used in evaluation 54
4.4 YCSB Workloads 55
4.5 SplitFS system call overheads 57
4.6 SplitFS vs. Strata 64

5.1 Page faults 91
5.2 Remote NUMA accesses 98

6.1 DAX Features 102
6.2 Code breakdown of ScaleMem 115
6.3 Network IO 122
6.4 ScaleMem Techniques 125
6.5 Applications Network IO 128

Chapter 1

Introduction

Today, in the age of data explosion, exabytes of data is generated everyday [1], and efficient access to the data has become a necessity. Widely used applications such as cloud services [2, 3, 4, 5, 6, 7, 8], mobile applications [9, 10, 11, 12], key-value stores [13, 14, 15, 16, 17, 18, 19, 20, 21, 22] and databases [23, 24, 25, 26, 27, 28, 29, 30] all run in data centers all over the world, and strive to provide efficient access to data. Traditional storage technologies such as HDDs have not kept up with the demands for high bandwidth and low latency access of data. This has led to diversification of storage as well as memory technologies into a variety of offerings, that each presents a new trade-off in the space of capacity, performance and cost. An emerging technology that promises to revolutionalize storage is Persistent Memory (PM).

Persistent Memory (PM) is a form of storage class memory, and exists in various forms such as Phase Change Memory [31], Spin-Torque Transfer RAM [32], Battery-backed DRAM, 3D XPoint [33], Memory-semantic SSDs [34], and so on. In order to accommodate large working sets of data center workloads, newer interconnects have been proposed, such as the Compute eXpress Link (CXL). This enables a single data center server to accommodate terabytes of PM.

The first commercial offering of scalable PM is Intel's Optane DC Persis-

tent Memory. PM is placed on the memory bus like DRAM and is accessed via processor loads and stores. Compared to DRAM, loads on PM have 2–3.7× higher latency and $1/3^{rd}$ bandwidth, while stores have the same latency but $1/6^{th}$ bandwidth [35]. A single machine can be equipped with up to 12 TB of PM. Given its memory semantics, large capacity and low latency, PM can be used as memory as well as storage.

Given an order of magnitude higher cost per GB of PM along with two orders of magnitude lower latency compared to modern SSDs, storage systems must answer the question: *how to get the best performance out of PM?* Legacy storage systems are designed with the assumption that the underlying storage device is a block device and is significantly slower than memory. The systems typically buffer data in DRAM in the page cache, batching updates to storage, to amortize the high cost. However, with PM, which offers similar characteristics to DRAM, buffering of data is unnecessary and leads to high software overhead due to extra copies of the data.

Direct Access (DAX) [36] is a feature introduced in the Linux kernel that removes the extra copies of data in the page cache by performing reads and writes directly to PM. Furthermore, memory mapping a file using DAX causes the PM addresses to be mapped to user space, without copying the data to the page cache. This allows loads and stores to directly reach PM without going through DRAM. PM storage systems commonly rely on DAX for low latency access to PM [37, 38, 39, 40].

The applications that use PM for storing data can be categorized into 2 classes. The first class of applications is legacy I/O intensive applications, that are designed for slower storage media, such as HDDs and SSDs, but

that can benefit from the high performance of PM. Examples of these applications are databases [27, 41], file & mail servers [42, 43], and so on. The second class of applications are newer applications that are built for PM, which take advantage of DAX. Examples of these applications are key-value stores [44, 17, 45], PM-resident indexes [46, 47] and caching services [48, 49]. There are also big data applications that require low latency access to terabytes of data that can span across the PM of multiple nodes. Examples of such applications are graph processing systems [50, 51], ML training frameworks [52], etc.

One way to get the best performance out of PM is to build new application interfaces that are more suited to the characteristics of PM. For example, frameworks such as Persistent Memory Development Kit (PMDK) [53] or Mnemosyne [54] allow applications to create crash-consistent data structures on PM instead of storing data in files. While new applications can benefit from using these frameworks, it becomes infeasible for the legacy applications with millions of lines of code to change the way they interact with storage.

In this dissertation, we answer the question: *how to achieve high performance for applications on PM without application changes?*. We study the software overheads incurred by current storage systems for legacy and new applications, with varied data sets, and come up with solutions that minimize the software overheads, maximizing the performance across all classes of applications.

1.1 Accelerating legacy applications with SplitFS

Legacy applications use POSIX system calls such as `open()`, `close()`, `read()`, `write()` to access their data on PM. Existing file systems add large overheads to each file system operation. The overhead comes from performing expensive operations on the critical path, including allocation, logging, and updating multiple complex structures. For example, consider the common operation of appending 4K blocks to a file (total 128 MB). Current file systems suffer from significant overhead (3.5×–12×) for this simple operation, as compared to writing raw 4KB data to PM.

In this dissertation, we present **SplitFS** [55], a PM file system that seeks to reduce software overhead via a novel *split* architecture: a user-space library file system handles data operations while a kernel PM file system (ext4 DAX) handles metadata operations. We refer to all file system operations that modify file metadata as *metadata operations*. Such operations include `open()`, `close()`, and even file appends (since the file size is changed). The novelty of SplitFS lies in how responsibilities are divided between the user-space and kernel components, and the semantics provided to applications. Unlike prior work like Aerie, which used the kernel only for coarse-grained operations, or Strata, where all operations are in user-space, SplitFS routes *all* metadata operations to the kernel. While FLEX [56] invokes the kernel at a fine granularity like SplitFS, it does not provide strong semantics such as synchronous, atomic operations to applications. At a high level, the SplitFS architecture is based on the belief that if we can accelerate common-case data operations, it is worth paying a cost on the comparatively rarer metadata operations. This is in contrast with in-kernel file systems like NOVA which extensively modify

the file system to optimize the metadata operations.

SplitFS transparently reduces software overhead for reads and overwrites by intercepting POSIX calls, memory mapping the underlying file, and serving reads and overwrites via processor loads and stores. SplitFS optimizes file appends by introducing a new primitive named *relink* that minimizes both data copying and trapping into the kernel. The application does not have to be rewritten in any way to benefit from SplitFS. SplitFS reduces software overhead 17× compared to other PM file systems.

Evaluating SplitFS with micro-benchmarks and real applications, we show that it outperforms state-of-the-art PM file systems like NOVA on many workloads by up to 2×. The design of SplitFS allows users to benefit from the maturity and constant development of the ext4 DAX file system, while getting the performance and strong guarantees of state-of-the-art PM file systems. SplitFS is publicly available at `https://github.com/utsaslab/splitfs`.

1.2 Accelerating modern PM applications with WineFS

Newer PM-native applications memory-map files and access data using loads and stores directly from user-space. The performance of memory-mapped applications depends on the number of page faults incurred while accessing data. Hugepages are an important optimization that reduce the number of page faults, but require files to be placed in 2MB contiguous and aligned physical extents on PM.

When existing PM file systems are aged, file layouts and free space tend to be fragmented. This causes the memory-mapped files to be fragmented and placed in unaligned holes of free-space. This causes up-to 512× more

page faults in the critical path, and reduces performance significantly.

This dissertation presents **WineFS** [57], a novel *hugepage-aware* PM file system designed to ensure that hugepages can almost always be used for memory-mapped files, even when the FS is aged and mostly full. Doing so requires a holistic design that both avoids fragmentation-inducing algorithms and proactively considers hugepage boundaries during allocations. Aging causes minimal performance loss in WineFS even when 90% full.

WineFS is designed, end-to-end, to avoid disruption of hugepage usage. Many inter-related aspects of a file system, from allocation policies to crash-consistency schemes to concurrency, affect its ability to keep using hugepages as it ages. WineFS uses a novel *alignment-aware* allocator that tries to preserve 2MB-aligned 2MB extents that can be mapped using hugepages. Large allocation requests are satisfied using aligned extents, while smaller allocations are satisfied using "holes". WineFS uses journaling for crash consistency, rather than the log-structuring popular in system-call-focused PM file systems [37, 58], to avoid data re-locations that disrupt its carefully planned layouts.

We evaluate WineFS using a variety of micro-benchmarks, macro-bench-marks, and applications. We measure performance on both aged file systems. For applications like RocksDB [13], LMDB [59], and PmemKV [17], that access PM via memory-mapped files, WineFS outperforms other PM file systems on an aged setup by up-to 2×.

1.3 Accelerating big data applications with ScaleMem

Big data PM applications refer to PM-native applications that issue loads and stores to access data on PM using DAX, but whose data does not fit

in the PM capacity of a single server. Unfortunately, the DAX memory-mapping method can only be used to access PM that is locally present. Distributed PM file systems do not support memory mapping, only allowing system calls like `read()` and `write()`. You can memory map remote PM via the NFS distributed file system; however, this does not offer DAX semantics. For example, persisting data requires a system call rather than a simple cache-flush instruction. This limitation is particularly important since it is common in industry to use centralized storage servers to hold data; applications cannot use DAX memory-mapping to read and write such data.

This dissertation introduces the *distributed DAX memory mapping* abstraction, `ddmap()`, for PM. *Unmodified* applications can create DAX memory mappings for PM, regardless of whether it is local or remote in a cluster. DAX semantics are provided for the memory-mapped data; a cache-flush instruction is sufficient to persist data. Applications are provided with the illusion of running on a server with a large amount of PM locally attached to it. We implement the `ddmap()` abstraction in the **ScaleMem** system.

ScaleMem co-designs the file system and memory management subsystems, to efficiently handle `ddmap()` faults. For example, ScaleMem finds that the default Linux kernel mechanism for unmapping and remapping a page does not scale due to its locking scheme; ScaleMem introduces a new primitive termed *Fast Reallocation* that allows unmapping and remapping pages in a more efficient and scalable manner. While handling page faults, ScaleMem tracks hot data at fine granularities by interposing on `memcpy()` and other library calls, and building a fine-grained read-only cache of PM data. In workloads with high degrees of skew and locality, the

fine-grained read cache allows ScaleMem to achieve significantly higher performance.

We evaluate ScaleMem on a variety of microbenchmarks and applications on multiple nodes. We show that ScaleMem is able to adapt to a variety of different workloads with the help of application-specific policies, and outperforms NFS [60] by up-to **60**× on certain workloads while doing 100× lesser network I/O. By intercepting POSIX system calls along with page faults, ScaleMem is able to support complex applications such as RocksDB [13] and LMDB [61], outperforming NFS by up-to 7× on read-heavy workloads. ScaleMem also outperforms PM-native applications such as PmemKV [17] by 2× while providing fine-grained cacheline durability guarantees.

1.4 Contributions

We describe the main contributions of this dissertation.

1. Analysis of software overheads and degradation of performance with age in different PM file systems for legacy I/O intensive applications as well as newer PM applications.

2. Design and implementation of a new file system, SplitFS that accelerates legacy POSIX applications through a novel split of responsibilities between user space and kernel space.

3. Design and implementation of a new huge-page-aware file system, WineFS that ages gracefully, with the help of alignment-aware allocations and a suitable on-PM layout, along with high scalability through per-CPU journaling.

4. Introduction of the distributed DAX memory mapping (`ddmap()`) abstraction for PM, which allows unmodified applications to create DAX memory mappings for PM, regardless of whether it is in a local or remote server in a cluster.

5. Implementation of `ddmap()` abstraction in the ScaleMem system, which co-designs the file system and memory management layers, providing the same guarantees to applications as that of local DAX memory mappings.

1.5 Overview

We present an overview of the remaining chapters in the dissertation.

1. **Background.** In Chapter 2, we provide background on Persistent Memory along with its characteristics and modes of operation. We discuss the different modes in which PM is accessed, via system calls and memory mapping using Direct Access. We discuss the existing file systems that are designed for PM. We discuss the different crash consistency mechanisms used by PM file systems, along with their trade-offs.

2. **Motivation.** In Chapter 3, we motivate this dissertation by understanding about the software overheads incurred by existing storage systems on PM. We discuss how the software overheads impact application performance when file systems are newly created as well as when aged. Finally, we discuss the lack of support by PM systems for applications whose data does not fit in the PM capacity of a single server.

3. **Solutions.** The next three chapters describe our new systems for accelerating legacy and new PM applications In Chapter 4, we describe SplitFS, a new PM file system for legacy applications that uses a novel split of responsibilities for accelerating data operations from a user library, while relying on a mature kernel file system for metadata operations. In Chapter 5, we describe WineFS, a new in-kernel file system that is targeted towards newer PM applications by reducing page faults through alignment-aware allocations and a suitable on-PM layout. In Chapter 6, we describe ScaleMem which implements the `ddmap()` abstraction for providing native load and store access to PM applications whose data does not fit in the PM capacity of a single server.

4. **Related Work.** In Chapter 8, we describe prior work on PM file systems, distributed file systems and distributed memory management. We discuss work related to our specific techniques, and describe efforts similar to ScaleMem in scaling application data sets to multiple servers.

5. **Future Work, Lessons Learned, and Conclusions.** In Chapter 9, we describe avenues in which our work could be extended. We describe how we can overcome limitations of our work, and apply it to other contexts, along with its applicability to other storage media. In Chapter 10, we highlight lessons learnt and summarise our work.

Chapter 2

Background

In this chapter, we provide the background required for various aspects of this dissertation. First, we discuss the different kinds of storage class memory technologies (§2.1) and the performance characteristics of Intel Optane DC Persistent Memory (PM) (§2.2). We then talk about the support of PM in the Linux Kernel (§2.3). We then discuss the existing file systems that have been designed for PM, along with their properties (§2.4). Finally, we talk about memory mapping on PM (§2.5).

2.1 Storage Class Memory

Storage class memory (SCM) refers to non-volatile memory technologies such as Phase Change Memory (PCM), Spin-Torque Transfer RAM (STT-RAM), resistive RAM (ReRAM), 3D XPoint and Byte-Addressable SSDs. SCM is also referred to as Non-Volatile Main Memory. Battery-backed DRAM has been commercially available for a long time from different vendors, while Intel Optane DC persistent memory that uses 3D XPoint technology made its debut in 2019. While the different NVMM technologies differ in their characteristics, they share a common goal of providing fast byte-addressable access to data, along with persistence across power cycles.

NVMMs typically occur in the form of non-volatile DIMMs (NVDIMMs)

and are attached to the main-memory bus, alongside DRAM. Attaching NVMMs to the main memory bus allows them to be accessed with very low latencies, similar to DRAM. Due to their byte-addressable nature, NVMMs can be accessed using processor load and store instructions, and exhibit high random access performance, in contrast to secondary storage devices like HDDs and SSDs. In this way, NVMMs blur the gap between volatile memory and non-volatile storage, offering the best of both worlds in terms of latency and durability of data.

2.2 Intel Optane DC Persistent Memory

Intel Optane DC Persistent Memory Module (which we refer to as PM) is the first scalable, commercially available NVMM. Compared to existing storage devices such as HDDs and SSDs, it offers lower latencies and higher read/write bandwidth, and offers a load-store interface instead of a block-based interface. Compared to DRAM, it has lower cost per GB, higher density and is able to persist data across power cycles.

The introduction of PM has enabled this thesis to understand the trade-offs and characteristics of NVMM technologies, and allowed us to build sophisticated systems software, thus getting the best out of NVMM. In the rest of this thesis, we use the terms PM and NVMM interchangeably, to refer to Intel Optane DC Persistent Memory Module.

2.2.1 Characteristics of PM

PM uses the 3D XPoint NVMM technology to offer byte-addressable persistence to data, and occurs in the form of NVDIMMs. The NVDIMMs are attached to the main-memory bus, alongside DRAM, as shown in Fig-

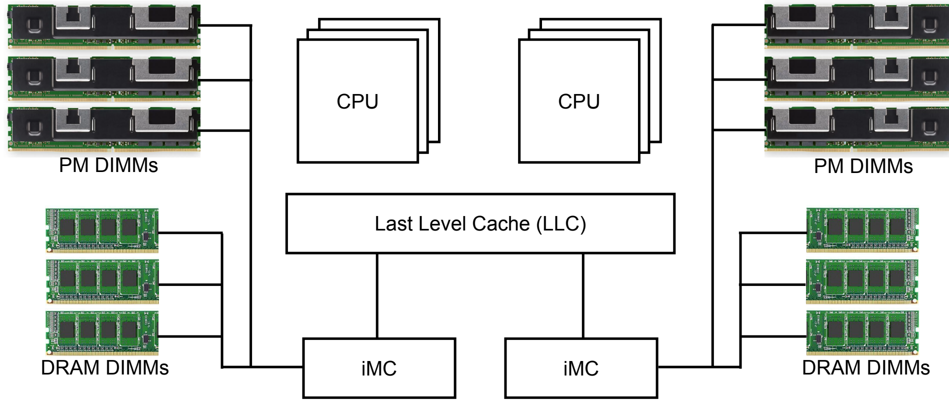

Figure 2.1: **Persistent Memory layout.** This figure shows the layout of PM in CascadeLake Servers. PM is placed alongside DRAM on the memory bus, and is managed using an integrated Memory Controller.

ure 2.1. PM is supported on Intel's CascadeLake processors. This processor contains multiple processor dies, each of which has its own memory bus, and comprises of separate NUMA nodes. The integrated memory controller (iMC) contains channels that supports PM NVDIMMs as well as DRAM volatile DIMMs. A single processor can contain up-to 9 TB of PM across multiple NUMA nodes.

PM guarantees durability of data in the iMC with the help of *Asynchronous DRAM Refresh (ADR)*. ADR is a feature where the power supply signals the different components of the processor about an imminent power failure. The write pending queues (WPQ) use the residual power for flushing data in the memory subsystem. Therefore, once the data is in the WPQs, it is guaranteed persistence. The ADR domain does not include the processor caches, which require explicit flushing of data to the WPQs for guaranteed durability.

Intel processors offer programmers with instructions to control store ordering. Data can be written to PM using normal store instructions such as `mov` which are cached in the CPU caches, or streaming instructions such

as `movnti`, which directly enter the WPQs. The `clflush` and `clflushopt` instructions can be used to flush the processor caches to memory, while `clwb` instruction can be used to write-back data, without evicting them. All these instructions are non-blocking, and can be re-ordered by the processor. To ensure ordering, programmers must use the `sfence` or `mfence` instructions, which ensure that all previous cachelines are written to memory.

2.2.2 PM Operating Modes

PM primarily operates in two modes: Memory Mode and AppDirect Mode. In both these modes, PM NVMDIMMs can be configured to interleave at 4KB granularities for load balancing. With six NVDIMMs, an access larger than 24KB will end up accessing all the NVDIMMs.

Memory Mode. Memory mode uses PM to expand DRAM capacity. It combines an NVDIMM with a conventional DRAM DIMM on the same memory channel that serves as a direct-mapped cache for the NVDIMM. In this mode, PM does not guarantee durability of data. Memory mode is used for increasing the size of heap-based applications such as in-memory databases, indexes or caches.

AppDirect Mode. AppDirect mode exposes PM in the form of block devices, similar to other storage media. This allows users to mount PM as a block device, and to store data in the block device. In this mode, PM guarantees durability of data. DRAM is not used as a cache for PM in this mode. The AppDirect mode is used for providing high-performance storage, such as key-value stores, file systems, persistent graphs and so on.

In this thesis, we focus on PM as a storage device, and use the AppDirect

Medium	Read Latency	Write Latency	Read BW	Write BW
DRAM	100 ns	100 ns	100 GB/s	100 GB/s
PM	300 ns	100 ns	6.6 GB/s	2.2 GB/s
SSD	10 μs	10 μs	2.2 GB/s	0.9 GB/s
HDD	10 ms	10 ms	0.1 GB/s	0.1 GB/s

Table 2.1: **PM properties.** This table shows the latency and bandwidth of a single PM NVDIMM in comparison to other memory and storage technologies.

mode with the interleaved NVDIMM setup.

2.2.3 PM Performance profile

PM offers a unique and complex performance profile that is different from DRAM, and depends on multiple factors such as number of threads and access size and access pattern. Loads and stores that are smaller then 256B cause amplification at the PM media, and have lower performance [35]. Accessing PM with more threads causes head-of-line blocking at the queues, which reduces its performance [35]. We now look at the latency and bandwidth profile of PM. Table 2.1 shows the performance profile of PM in comparison with other memory and storage technologies.

Latency. PM offers asymmetric read/write latencies. When performing 8-byte load instructions with a single thread, the sequential read latency is around 170ns [35], while the random read latency is around 300ns [35]. This difference between sequential and random read latency is because small random reads lead to read-amplification at the NVDIMMs, due to the 256B internal block size of PM. Hence, the read latency of PM is 2-3× higher than DRAM.

The latency of writes on PM for random or sequential writes is approx-

imately 100ns [35]. This is because stores do not write to NVDIMMs in the critical path, but go through the WPQs at the processor, which is the same for DRAM.

Bandwidth. The bandwidth of reads and writes on PM depends on the number of threads and access sizes and number of NVDIMMs. The maximum read bandwidth of a single NVDIMM is 6.6 GB/s [35], while the maximum write bandwidth of a single NVDIMM is 2.3 GB/s [35]. A server with the maximum number of PM NVDIMMs, the maximum read bandwidth is 1/3rd that of DRAM, while the maximum write bandwidth is 1/6th compared to DRAM.

Effect of NUMA on PM. NUMA-effects play a big role in the performance of PM. Remote NUMA read latencies are up-to 1.79× higher than local NUMA reads, while remote NUMA write latencies are up-to 2.5× higher than local NUMA writes [62]. Remote PM can achieve 59% and 62% of local PM read and write bandwidth [62].

The bandwidth difference increases when accessing data in a mixed read/write manner. In a mixed read/write access pattern, remote bandwidth is up-to 3× lower than local bandwidth.

2.3 Direct Access (DAX)

The introduction of PM has resulted in addition of new features in the Linux kernel to take advantage of its low-latency storage. An important feature introduced in the Linux kernel to help users access PM is called Direct Access (DAX) [36], that bypasses the page cache while accessing data on PM.

The page cache is usually used to buffer reads and writes to files. For

PM, which is a byte-addressable media, the page cache pages cause unnecessary copies of the original storage. The DAX code removes the extra copy by performing reads and writes directly to the storage device.

2.4 File systems on PM

File systems that are designed for low-latency persistent storage offered by PM have the potential to impact a large number of applications. There is active research on building native PM file systems [37, 63, 64, 40, 58] as well as extending existing file systems meant for other storage media, to run efficiently on PM. [38, 39].

2.4.1 POSIX API

PM file systems (similar to HDD/SSD-based file systems) typically implement the well-known POSIX API, which contains operations to manage files. We refer to all file system operations that modify file metadata as *metadata operations*. Such operations include `open()`, `close()`, `rename()`, `unlink()`, etc. We refer to operations that are involved in reading or writing file data as *data operations*. These operations are `read()` and `write()`. Appending data to a file is considered as a metadata operation, since it changes the size of the file.

All POSIX-compliant file systems are expected to provide *failure-atomic metadata operations*. This implies that in the event of a system crash, the file system should reflect either the old version of the file metadata in its entirety, or the new version of the file metadata in its entirety. For guaranteeing durability of data, POSIX includes a primitive called `fsync()`. The `fsync()` call guarantees that when the call returns, all dirty data as-

sociated with the file (passed as an argument to `fsync()`) is persistent on PM.

2.4.2 File system crash-consistency guarantees

PM file systems strive to provide strong consistency and durability guarantees to applications, which can be broken down into the following classes.

Metadata consistency File systems that provide metadata consistency are guaranteed to recover to a consistent state after a crash with respect to its metadata. However, the file data can remain inconsistent, and can contain garbage values. This guarantee is provided by all POSIX-compliant PM file systems.

Synchronous Data Operations In synchronous data operations, when a `read()` or `write()` is complete, the data is guaranteed to be durable and recoverable when the call returns. In other words, applications do not need a subsequent `fsync()` for data durability.

Synchronous Metadata Operations Synchronous metadata operations provide immediate durability of metadata of a files, and guarantee that the metadata of a file is persisted on successful completion of any file system operation.

Data consistency Data consistency implies that all data operations are failure atomic. This implies that if there is a system crash during a `write()` operation while updating file data, the file system is guaranteed to recover to a state that contains the new data in its entirety, or contains the old data in its entirety.

2.4.3 Crash consistency mechanisms

PM file systems use different mechanisms such as journaling, copy-on-write and log structuring for achieving failure atomicity of data and metadata. Each of these mechanisms offer different performance trade offs.

Journaling. Journaling file systems reserve a specific region in the PM partition for journaling updates to file metadata and (optionally) file data. Every operation that modifies the metadata of a file involves creating a transaction in the journal. In the event of a crash, the journal is replayed to bring the file system to a consistent state. Journaling can occur in two forms. *Undo journaling* or write-ahead logging involves writing old values of the metadata (or data) to the journal before updating the values in-place. In the event of a crash, incomplete journal transactions are rolled back, and the old values are copied from the journal to the file-system data structures, to bring the file system to a consistent state. *redo journaling* involves writing new values in the journal, and a subsequent checkpointing phase where the new values are copied from the journal to the file system data structures.

Journaling requires writing data twice: once to the journal and once to the target location. Data journaling is expensive, since the entire data has to be written twice, causing high write amplification. As a result, PM file systems typically journal the metadata and write data directly to the target location.

copy-on-write or shadow paging. Shadow paging file systems never update data in-place. Any update to a data involves writing the new version to another location, flushing the newly written data to PM, and updating the metadata to point to the new location atomically. The old data is then

reclaimed and added to the free list of blocks. In the event of a crash, the metadata points to the old version of the data, and ensures that the file system is in a consistent state. The file metadata pointing to the file data is often kept in a tree structure, which uses transactions to atomically point to the new location.

Updating file data using copy-on-write also suffers from write amplification: copying the old data to a new location, and then updating the data in the new location.

Log structuring. Log structured file systems (LFS) were designed to exploit the sequential access performance of hard disk drives or SSDs. These file systems buffer random writes in memory and convert them into larger, sequential writes to disk, making the best of hard disks' sequential access performance. Log structured file systems optionally maintain file metadata in a log structured manner as well, by using version control. Stale metadata is reclaimed in the background to avoid random accesses in the critical path. Although log structuring makes more sense for HDDs and SSDs, there are PM file systems that manage metadata in a log structured fashion, to avoid performing small random updates.

LFS performance relies on being able to write data sequentially, in free contiguous regions on PM. To ensure a consistent supply of such regions, LFS constantly clean and compact the log to reclaim space occupied by stale (meta)data. Log cleaning and compacting incurs from read and write amplification, which degrades the performance of LFS. Apart from this, LFS also write data out-of-place, suffering from problems similar to shadow paging.

2.4.4 Existing PM file systems

Researchers, companies and open-source communities have designed and implemented a number of PM file systems. We divide PM file systems into two groups. Native PM file systems are designed especially for PM. They exploit the byte-addressability of PM storage and can dispense with many of the optimizations (and associated complexity) that blockbased file systems implement to hide the poor performance of disks. BPFS [63] is the first native PM file system that we are aware of. BPFS is a shadow paging file system that provides metadata and data consistency. BPFS proposes a hardware mechanism to enforce store durability and ordering and uses short-circuit shadow paging to reduce shadow paging overheads in common cases. PMFS [40] is a lightweight PM file system that bypasses the block layer and file system page cache to improve performance. PMFS uses journaling for metadata updates. SCMFS [64] utilizes the operating system's virtual memory management module and maps files to large contiguous virtual address regions, making file accesses simple and lightweight. SCMFS does not provide any consistency guarantee of metadata or data. Aerie [65] implements the file system interface and functionality in user space to provide low-latency access to data in PM. Aerie journals metadata but does not support data atomicity or mmap operation. Strata [58] is a "cross media" file system that runs partly in userspace. It provides strong atomicity and high performance.

Adapted PM file systems (or just "adapted file systems") are block-based file systems extended for PM. xfs DAX [39] and ext4 DAX [38] have modes in which they become adapted file systems. xfs DAX and ext4 DAX are the state-of-the-art adapted PM file systems in the Linux kernel. They

add DAX support to the original file systems so that data page accesses are bypassing the page cache, but metadata update still go through the old block-based journaling mechanism. So far, adapted file systems have been built subject to constraints that limit how much they can change to support PM. For instance, they use the same on-“disk” format in both block-based and DAX modes, and they must continue to implement (or at least remain compatible with) disk-centric optimizations. Existing PM file systems usually store efficient data structures such as B+tree and radix tree in PM to manage inodes and files. However, persistent data structures are difficult to implement and usually incur large performance overheads due to cache flush and memory ordering operations

2.5 Memory mapping on PM

A memory map operation (performed via the `mmap()` system call) maps one or more pages in the process virtual address space to extents on PM. For example, consider virtual addresses `4K` to `8K-1` are mapped to bytes `0` to `4K-1` on file `foo`. Bytes `0` to `4K-1` in `foo` then correspond to bytes `10*4K` to `11*4K -1` on PM. A store instruction to virtual address `5000` would then translate to a store to byte `40964` on PM. Thus, PM can be accessed via processor loads and stores without the interference of software; the virtual memory subsystem is in charge of translating virtual addresses into corresponding physical addresses on PM.

2.5.1 Page Faults

The process of setting up the mapping between the process virtual address space and corresponding physical addresses on PM happens on a page fault.

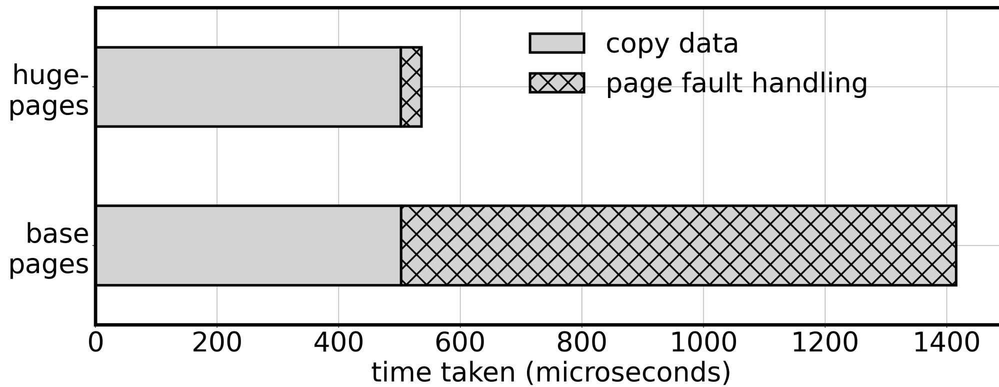

Figure 2.2: **Memory-mapping overhead.** This figure shows the time taken to memory-map and write a 2MB file, with and without huge pages. With huge pages, most of the time goes towards copying data. Without huge pages, two thirds of total time goes towards handling page faults and setting up page tables. Note that using huge pages makes writing the file 2× faster.

Page faults occur on the first access of a page, and trap into the kernel, which sets up page table entries and TLB mapping for the corresponding virtual address. By default, the size of a page is 4KB, and accessing a 1GB file incurs more than 200K page faults.

2.5.2 Hugepages

To reduce the overheads of page faults, modern processors support *hugepages*, which are 2MB or 1GB in size. Hugepages reduce the number of page faults by 500×. We run an experiment where we memory-map and write to a 2MB file, with and without hugepages. Figure 2.2 shows the results: that mapping with hugepages can reduce the overall time taken by 2×, by reducing the time taken to handle page faults.

Conditions for obtaining hugepages. Despite the benefits of hugepages, it is challenging for applications using the memory-mapped access mode to reliably get hugepages. In order to get a hugepage on a page fault, the

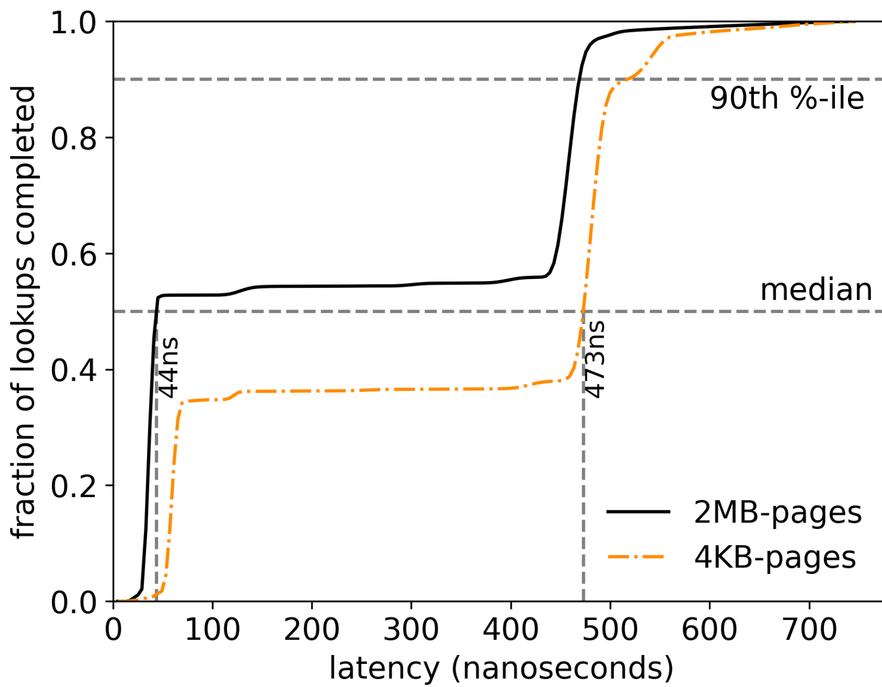

Figure 2.3: **Overhead due to TLB misses.** This figure shows the CDF of latencies when reading random elements of a large PM array that has been memory-mapped and pre-faulted. Using huge pages reduces the number of TLB misses. On a TLB misses, page table entries are fetched and cached in the processor caches, reducing cache space for the application. The median latency is 10× higher when using base pages rather than huge pages, as the array element that is read has been knocked out of the processor cache by page table entries.

underlying file must be placed on 2MB aligned physical blocks and must not be fragmented. Even a single byte offset from alignment forces the operating system to fall back to base pages with a high page-fault cost.

Pre-faulting pages. A natural question that arises is: can we simply pre-fault all of the pages outside the critical path. First, this is simply not possible for a number of applications (such as LMDB [59]) as they use sparse mappings, allocating space on demand when they get a page fault. Pre-faulting would lead to unacceptable space overhead for these applications. Second, even if all the pages are pre-faulted, hugepages still provide a performance benefit by reducing TLB misses. We ran an experiment where we memory-mapped a file containing a large array and randomly read elements in the array. The entire file was pre-faulted, so there were no page faults in the critical path. Figure 2.3 shows the results. There is a 10× reduction in median latency when using hugepages, corresponding to whether the array element that was read was in the processor cache or not. When using base pages, the array element was kicked out to make space for page table entries when handling TLB misses. Thus, even when all pages are pre-faulted, using hugepages to map files on PM improves performance.

2.6 Summary

In this chapter, we introduced the background material essential for this dissertation. We discussed about Intel Optane DC Persistent Memory (PM) and its characteristics. We then looked at support in the Linux kernel for PM in the form of Direct Access (DAX). We then described the different ways in which file systems are designed for PM; along with their crash consistency mechanisms. Finally, we discussed memory mapping on

PM in the presence of DAX.

Chapter 3

Motivation

We use the background material presented in Chapter 2 to motivate this dissertation. We studied existing PM file systems that use different data structures and crash consistency mechanisms, providing different trade offs between performance and consistency guarantees for data and file system metadata.

In this chapter, we discuss the different ways in which applications access PM (§3.1) along with software overheads of the file systems for the applications (§3.2). We discuss the limitations of existing file systems and memory managers to support big data applications that use DAX (§3.3).

3.1 How PM is accessed

The low latency and high bandwidth enable a number of legacy and new applications to benefit from PM. We classify the applications into 2 categories based on the way that they access their data on PM.

3.1.1 System-call interface

Traditional applications designed for magnetic hard drives and solid state drives can access PM through POSIX system calls. The applications in this category typically create and delete files using `creat()` and `unlink()`,

and access their data using `read()` and `write()`. Examples of applications in this category are databases such as MySQL [27], SQLite [30] PostgreSQL [41], mail servers and file servers such as [43, 42], key-value stores such as LevelDB [14], etc.

The performance of POSIX system-call applications depends on the overheads of the kernel and file-system layers in accessing application data. The applications in this category issue frequent `read()` and `write()` calls for accessing data, with occasional metadata-related calls for creating, deleting and renaming files. In order to ensure high performance for the applications in this category, PM file systems must incur minimum overheads in data-transfer operations such as `read()` and `write()`.

3.1.2 Memory-mapped interface

Given the performance benefits of memory-mapping, applications designed specifically for PM tend to use this method to access data (key-value stores such as PmemKV [17], Pmem-Redis [45], RocksDB-pmem [44], LMDB [61], caching services such as Memcached [49], Pelikan [48], databases such as Memhive-PostgreSQL [25], data-structure libraries such as PMDK [53]).

The performance of memory-mapped applications depends on the time it takes for setting up page table entries so that a virtual address in the process can point to a location on PM. This overhead directly depends upon whether hugepages are used to map the underlying file.

3.2 Limitations of existing file systems

In this section, we discuss the limitations of existing file systems in achieving high performance for both the classes of applications. We discuss soft-

File system	Append (ns)	Overhead (ns)	Overhead (%)
ext4 DAX	9002	8331	1241%
PMFS	4150	3479	518%
NOVA	3021	2350	350%

Table 3.1: **Software Overhead.** The table shows the software overhead of various PM file systems for appending a 4K block. It takes 671 ns to write 4KB to PM.

ware overheads incurred by existing file systems for legacy applications in §3.2.1 and newer PM applications in §3.2.2.

3.2.1 Software Overheads of POSIX system-call interface

Traditional file systems add large overheads to each file-system operation, especially on the write path. The overhead comes from performing expensive operations on the critical path, including allocation, logging, and updating multiple complex structures. The systems community has proposed different architectures to reduce overhead. BPFS [63], PMFS [40], and NOVA [37] redesign the in-kernel file system from scratch to reduce overhead for file-system operations. Aerie [65] advocates a user-space library file system coupled with a slim kernel component that does coarse-grained allocations. Strata [58] proposes keeping the file system entirely in user-space, dividing the system between a user-space library file system and a user-space metadata server. Aerie and Strata both seek to reduce overhead by not involving the kernel for most file-system operations.

Despite these efforts, file-system data operations, especially writes, have significant overhead. For example, consider the common operation of appending 4K blocks to a file (total 128 MB). It takes 671 ns to write a 4

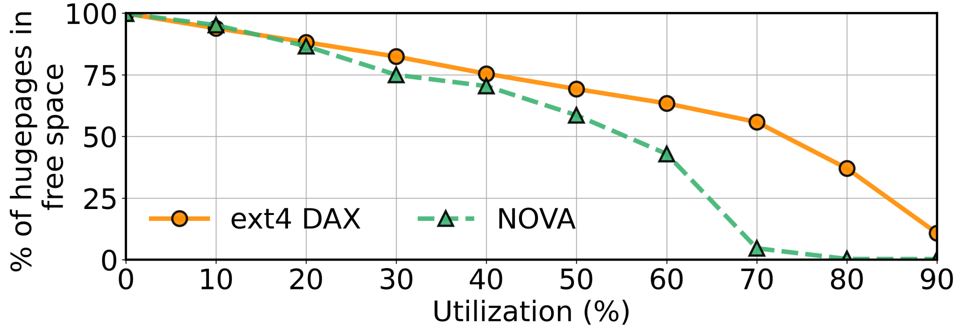

Figure 3.1: **Free space fragmentation.** Free-space becomes increasingly fragmented as utilization increases in aged NOVA and ext4 DAX. At 70% utilization, NOVA has close to zero 2MB aligned and contiguous regions.

KB to PM; thus, if performing the append operation took a total of 675 ns, the software overhead would be 4 ns. Table 3.1 shows the software overhead on the append operation on various PM file systems. We observe that there is still significant overhead ($3.5 - 12.4\times$) for file appends.

3.2.2 Performance overheads of Memory-mapped interface

While it is relatively easier for a freshly-formatted file system to place files on 2MB aligned and unfragmented regions on PM[1], it becomes increasingly difficult to maintain that alignment as the file system ages. File system instances are routinely used for several years at a time [66, 67, 68]. It is well known that as the file system ages, it suffers file and free space fragmentation because of the file creations, deletions and updates causing significant slowdowns [69, 70, 71, 72, 73]. In the context of PM and hugepages, arbitrary free space fragmentation worsens the problem of obtaining aligned and contiguous 2MB extents. Figure 3.1 shows the number of hugepages

[1]PMFS and xfs-DAX cannot get hugepages even when they are clean because unlike ext4-DAX, these file systems completely disregard alignment even for large extents.

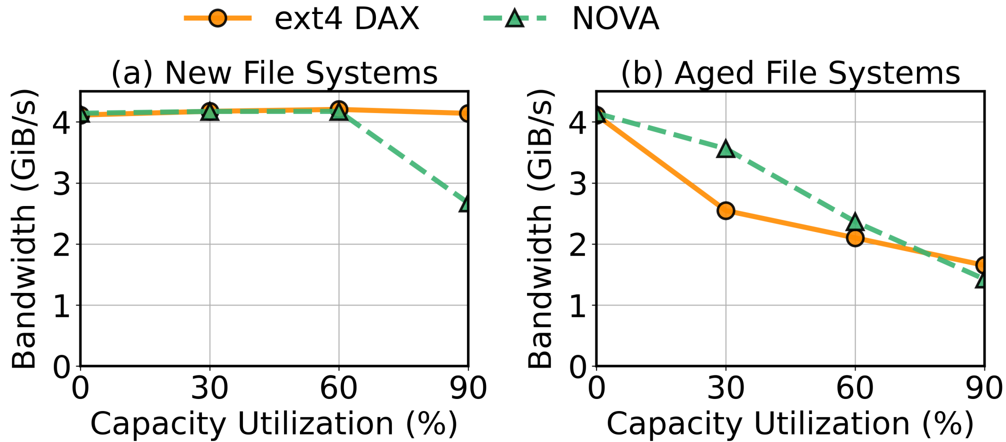

Figure 3.2: **Impact of aging.** Write bandwidth to memory-mapped files for two PM FSs on un-aged (left) and aged (right) file systems stored on Intel Optane PM. For ext4-DAX and NOVA, aging reduces bandwidth by ≈50% even when the FS is only 60% full.

available as a file system is aged. We performed aging using Geriatrix [70], an aging framework to perform aging. In this experiment 100GB file system partitions of ext4-DAX and NOVA were subjected to up to 40TB of file creates and deletes. With increasing utilization both ext4-DAX and NOVA are unable to maintain aligned 2MB extents. In fact at about 70% utilization, NOVA had close to zero 2MB extents left.

We observed that fragmentation of free space due to aging does not impact the performance of applications that access PM through POSIX system calls such as `read()` and `write()`, as PM offers similar bandwidth for sequential and random access of data.

In summary, the performance of applications accessing PM by mmaping files is up-to 2× better than applications accessing PM through POSIX system calls but can degrade significantly with age due to fragmentation of free space, while the performance of applications accessing via POSIX system calls remains constant and independent of age.

Hugepages without file system support. One solution to obtain hugepages is via defragmentation. In this context, defragmentation would mean re-alignment of extents to 2MB boundaries, and not necessarily focusing on its contiguity. One can imagine a user-space utility for defragmenting `mmap`-ed files, which would read the fragmented file, and rewrite it using large allocations. However, without file system support, it is impossible to guarantee that large allocation requests are satisfied using 2MB aligned extents. We observe that ext4-DAX and NOVA do not always use aligned extents when they are available; this is natural since these file systems optimize for locality and contiguity, rather than hugepages. For example, in Figure 3.2 (b), ext4-DAX has 12k aligned extents available at 60% utilization, but ends up using only 3k aligned extents, while the workload requires 8k aligned extents.

One could also imagine a file-system-wide defragmentation utility could be run to reclaim hugepages. However, existing defragmentation utilities do not aim to recover huge-pages. Such utilities would consume PM bandwidth when running in the background. The performance for a given PM file would only improve if it had been defragmented by the utility; this could take a lot of time depending on the size of the file system.

Finally, one could use a file system that always allocates in sizes of 2MB; the `bigalloc` mode of ext4 does this. However, this leads to significant space wastage and internal fragmentation, as a large number of user files tend to be small.

3.3 Enabling remote DAX memory mappings

Big data applications today must handle vast amounts of data in the presence of data explosion. For example, the largest graph datasets that are open source span multiple terrabytes in size [74]. Additionally, the high price of PM today makes it infeasible to be deployed in large capacities in singular servers, but makes it more practical to be shared across multiple servers in a cluster. Achieving high performance with increasing data requires applications to leverage the low-latency access of DAX for data spanning across the PM of multiple servers.

Big data applications typically use DAX by creating large files on a DAX file system, memory mapping the files in userspace, and then issuing direct loads and stores to the files. Examples of such applications are key-value stores such as LMDB [61], Pmem-Redis [45], PmemKV [17], data structure libraries such as Persistent Memory Development Kit (PMDK) [53], indexes such as RECIPE [46], FAST & FAIR [47], and so on.

Handling Remote DAX Page Faults. Using remote DAX requires the virtual memory subsystem and the file system components to work in tandem with each other. Consider that we have a simple application that tries to use DAX memory maps to access a dataset larger than the PM capacity of a single server. In order to serve a DAX page fault, the virtual address of the page must correspond to a physical block of the file on PM. This may involve migrating the block of file from the remote server to the local server, and set up the memory mapping on a page fault. Similarly, if the local PM runs out of space, a block of file must be migrated from the local server to a remote server that has enough PM space, before a new remote block can be brought in the local node.

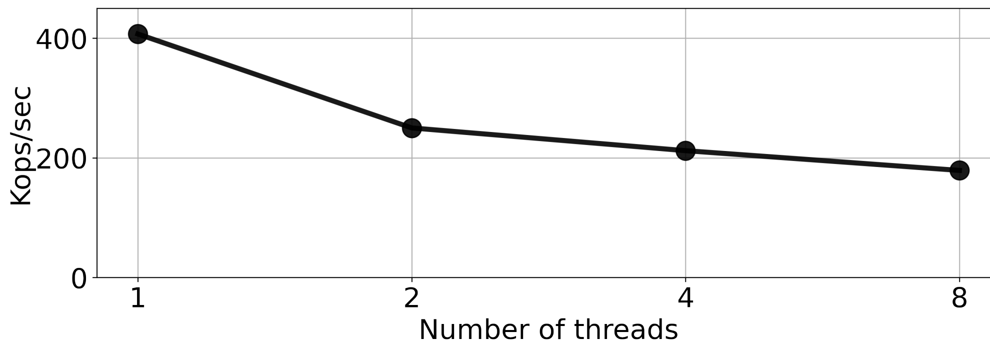

Figure 3.3: **Memory Manager Bottleneck.** Shows the drop in throughput with increasing threads due to lock contention in the memory manager on `mmap()` and `munmap()` operations.

Challenges of Remote DAX memory mappings. Supporting remote DAX memory mappings comes with a unique set of challenges. These challenges are related to resolving bottlenecks in the virtual memory as well as file system.

Migrating blocks between servers due to remote DAX page faults requires frequent unmapping and mapping of file regions. Unfortunately, the Linux memory manager is not equipped to efficiently handle concurrent frequent mappings and unmappings of memory within the same process. Linux maintains metadata corresponding to all the memory mappings of a process, and requires holding exclusive locks on this metadata on every `mmap()` and `munmap()` call. Fig 3.3 shows the cost of each `mmap()` call in the presence of other concurrent unmap and map calls on the same file, which increases as we increase the number of threads.

At the file system layer, frequent migrations of file blocks cause allocations and deallocations that fragment the file as well as the free space. Fragmentation of files also destroys the memory mappings associated with the file, because of lack of co-ordination between the memory manager and file system components. This increases number of page faults and results

in significant slowdowns.

3.4 Summary

We motivated the need for newer systems software for PM for achieving high performance along with stronger consistency and durability guarantees. We presented the different ways in which applications access PM, and saw the shortcomings of current file systems and memory managers in achieving high performance for a wide range of PM applications.

Chapter 4

Achieving high performance for legacy I/O intensive applications

In chapter 3, we discussed how legacy I/O intensive applications access PM using POSIX system calls. We looked at software overheads incurred by existing PM file systems in achieving high performance for legacy applications. In this chapter, we introduce a new design point in PM file systems for accelerating applications using `read()` and `write()` POSIX system calls for accessing data.

In the rest of the chapter, we first discuss division of responsibilities between user-space and kernel to reduce software overhead in file systems (§4.1). We then introduce SplitFS, a file system that lies in user space as well as the kernel, and achieves high performance for legacy applications. We present the goals of SplitFS (§4.3). We present an overview of the design and discuss how SplitFS provides atomic operations at low overhead (§4.5), and discuss its implementation (§4.6). We then discuss the performance of SplitFS in comparison to existing PM file systems on a wide variety of workloads (§4.8).

4.1 Rethinking division of responsibilities between user-space and the kernel

Legacy applications depend on file systems for every data access. In-kernel file systems result in context switches between user-space and the kernel for every data access, and perform expensive operations such as allocations and deallocations in the critical path. As a result of this, the file systems such as NOVA [37], PMFS [40], ext4 DAX [38], xfs DAX [39] suffer from high software overheads for common operations such as file appends. On the other hand, user-space file systems such as Strata [58] and Aerie [65] avoid performing expensive operations in the critical path and avoid frequent context switches, but suffer from high complexity and bugs due to reimplementing the entire VFS layer in user space.

We believe that the right way to achieve high performance combined with stability and maturity is to design a hybrid file system that lies in user space as well as the kernel. Common system calls such as `read()` and `write()`, which are easier to implement and are performance-critical, can be implemented in user space on pre-allocated regions to avoid expensive operations in the critical path. Metadata operations, which are rare and have a number of corner cases, can be implemented in the kernel with the help of a mature in-kernel file system. Such a design offers the best of both worlds from the point of view of performance and stability for legacy applications.

4.2 SplitFS: a user-kernel hybrid PM file system

As a part of this dissertation, we build SplitFS, a PM file system that seeks to reduce *software overhead* via a novel split architecture: a user-space library file system handles data operations while a kernel PM file system (ext4 DAX) handles metadata operations.

4.3 SplitFS Goals

Low software overhead. SplitFS aims to reduce software overhead for data operations, especially writes and appends.

Transparency. SplitFS does not require the application to be modified in any way to obtain lower software overhead and increased performance.

Minimal data copying and write IO. SplitFS aims to reduce the number of writes made to PM. SplitFS aims to avoid copying data within the file system whenever possible. This both helps performance and reduces wear-out on PM. Minimizing writes is especially important when providing strong guarantees like atomic operations.

Low implementation complexity. SplitFS aims to re-use existing software like ext4 DAX as much as possible, and reduce the amount of new code that must be written and maintained for SplitFS.

Flexible guarantees. SplitFS aims to provide applications with a choice of crash-consistency guarantees to choose from. This is in contrast with PM file systems today, which provide all running applications with the same set of guarantees.

Mode	*Sync. Data Ops*	*Atomic Data Ops*	*Sync. Metadata Ops*	*Atomic Metadata Ops*	*Equivalent to*
POSIX	✗	✗	✗	✓	ext4 DAX
sync	✓	✗	✓	✓	Nova-Relaxed, PMFS
strict	✓	✓	✓	✓	NOVA-Strict, Strata

Table 4.1: **SplitFS Modes.** The table shows the three modes of , the guarantees provided by each mode, and current file systems that provide the same guarantees.

4.4 SplitFS Modes and Guarantees

SplitFS provides three different modes: POSIX, sync, and strict. Each mode provides a different set of guarantees. Concurrent applications can use different modes at the same time as they run on SplitFS. Across all modes, SplitFS ensures the file system retains its integrity across crashes.

Table 4.1 presents the three modes provided by SplitFS. Across all modes, appends are atomic in SplitFS; if a series of appends is followed by `fsync()`, the file will be atomically appended on `fsync()`.

POSIX mode. In POSIX mode, SplitFS provides metadata consistency [75], similar to ext4 DAX. The file system will recover to a consistent state after a crash with respect to its metadata. In this mode, overwrites are performed in-place and are synchronous. Note that appends are not synchronous, and require an `fsync()` to be persisted. However, SplitFS in the POSIX mode guarantees atomic appends, a property not provided by ext4 DAX. This mode slightly differs from the standard POSIX semantics: when a file is accessed or modified, the file metadata will not immediately reflect that.

Sync mode. SplitFS ensures that on top of POSIX mode guarantees, operations are also guaranteed to be synchronous. An operation may be considered complete and persistent once the corresponding call returns and applications do not need a subsequent `fsync()`. Operations are not atomic in this mode; a crash may leave a data operation partially completed. No additional crash recovery needs to be performed by SplitFS in this mode. This mode provides similar guarantees to PMFS as well as NOVA without data and metadata checksumming and with in-place updates; we term this NOVA configuration *NOVA-Relaxed*.

Strict mode. SplitFS ensures that on top of sync mode guarantees, each operation is also atomic. This is a useful guarantee for applications; editors can allow atomic changes to the file when the user saves the file, and databases can remove logging and directly update the database. This mode does not provide atomicity across system calls though; so it cannot be used to update two files atomically together. This mode provides similar guarantees to a NOVA configuration we term *NOVA-Strict*: NOVA with copy-on-write updates, but without checksums enabled.

Visibility. Apart from appends, all SplitFS operations become immediately visible to all other processes on the system. On `fsync()`, appends are persisted and become visible to the rest of the system. SplitFS is unique in its visibility guarantees, and takes the middle ground between ext4 DAX and NOVA where all operations are immediately visible, and Strata where new files and data updates are only visible to other processes after the digest operation. Immediate visibility of changes to data and metadata combined with atomic, synchronous guarantees removes the need for leases to coordinate sharing; applications can share access to files as they would

Technique	*Benefit*
Split architecture	Low-overhead data operations, correct metadata operations
Collection of memory-maps	Low-overhead data operations in the presence of updates and appends
Relink + Staging	Optimized appends, atomic data operations, low write amplification
Optimized operation logging	Atomic operations, low write amplification

Table 4.2: **SplitFS Techniques.** The table lists each main technique used in SplitFS along with the benefit it provides. The techniques work together to enable SplitFS to provide strong guarantees at low software overhead.

on any other POSIX file system.

4.5 SplitFS Design

We now provide an overview of the design of SplitFS, and how it uses various techniques to provide the outlined guarantees. Table 4.2 lists the different techniques and the benefit each technique provides.

Split architecture. As shown in Figure 4.1, SplitFS comprises of two major components, a user-space library linked to the application called U-Split and a kernel file system called K-Split. SplitFS services all data operations (e.g., `read()` and `write()` calls) directly in user-space and routes metadata operations (e.g., `fsync()`, `open()`, etc.) to the kernel file system underneath. File system crash-consistency is guaranteed at all times. This approach is similar to Exokernel [76] where only the control operations are handled by the kernel and data operations are handled in user-space.

Collection of mmaps. Reads and overwrites are handled by `mmap()`-ing

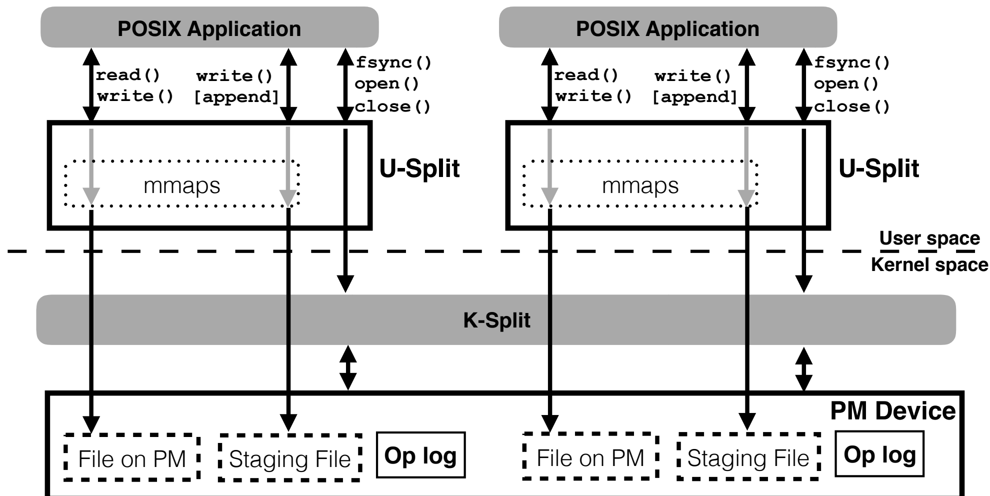

Figure 4.1: **SplitFS Overview.** The figure provides an overview of how SplitFS works. Read and write operations are transformed into loads and stores on the memory-mapped file. Append operations are staged in a staging file and *relinked* on `fsync()`. Other metadata POSIX calls like `open()`, `close()`, etc. are passed through to the in-kernel PM file system. Note that loads and stores do not incur the overhead of trapping into the kernel.

the surrounding 2 MB part of the file, and serving reads via `memcpy()` and writes via non-temporal stores (`movnti` instructions). A single logical file may have data present in multiple physical files; for example, appends are first sent to a staging file, and thus the file data is spread over the original file and the staging file. SplitFS uses a collection of memory-maps to handle this situation. Each file is associated with a number of open `mmap()` calls over multiple physical files, and reads and over-writes are routed appropriately.

Staging. SplitFS uses temporary files called staging files for both appends and atomic data operations. Appends are first routed to a staging file, and are later relinked on `fsync()`. Similarly, file overwrites in strict mode are also first sent to staging files and later relinked to their appro-

priate files.

Relink. On an `fsync()`, all the staged appends of a file must be moved to the target file; in strict mode, overwrites have to be moved as well. One way to move the staged appends to the target file is to allocate new blocks and then copy appended data to them. However, this approach leads to write amplification and high overhead. To avoid these unnecessary data copies, we developed a new primitive called relink. Relink logically moves PM blocks from the staging file to the target file without incurring any copies.

Relink has the following signature: relink(file1, offset1, file2, offset2, size). Relink atomically moves data from offset1 of file1 to offset2 of file2. If file2 already has data at offset2, existing data blocks are de-allocated. Atomicity is ensured by wrapping the changes in an ext4 journal transaction. Relink is a metadata operation, and does not involve copying data when the involved offsets and size are block aligned. When offset1 or offset2 happens to be in the middle of a block, SplitFS copies the partial data for that block to file2, and performs a metadata-only relink for the rest of the data. Given that SplitFS is targeted at POSIX applications, block writes and appends are often block-aligned by the applications.

Optimized Logging. In strict mode, SplitFS guarantees atomicity for all operations. To provide atomicity, we employ an Operation Log and use logical redo logging to record the intent of each operation. Each U-Split instance has its own operation log that is pre-allocated, `mmap()`-ed by U-Split, and written using `movnti` instructions. We use the necessary memory fence instructions to ensure that log entries persist in the correct order. To reduce the overheads from logging, we ensure that in the common case, per

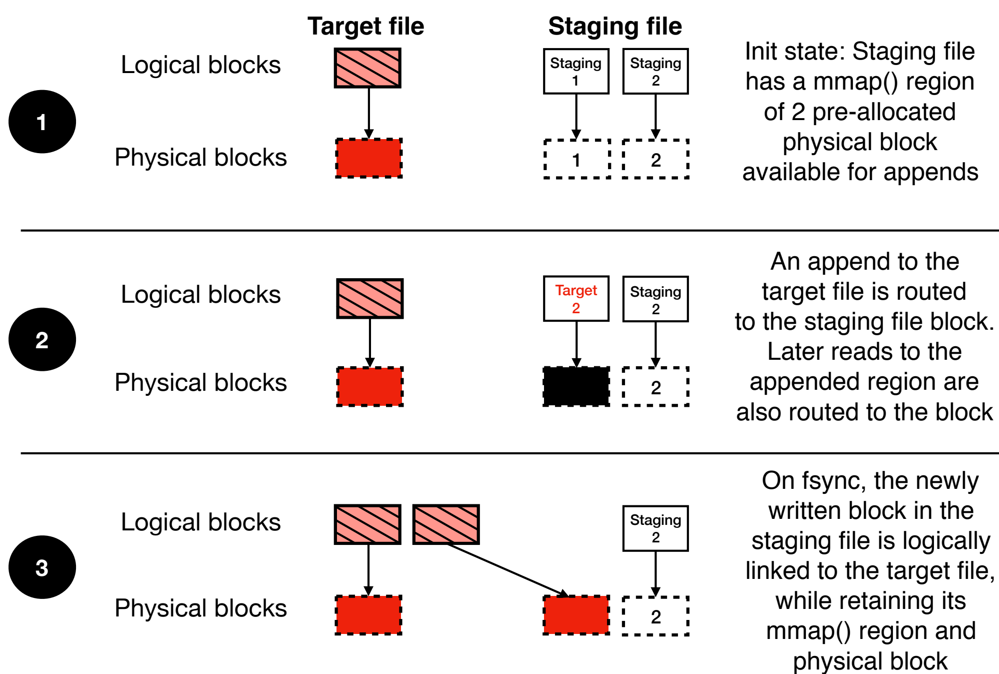

Figure 4.2: ***relink* steps.** This figure provides an overview of the steps involved while performing a relink operation. First, appends to a target file are routed to pre-allocated blocks in the staging file and subsequently on an `fsync()`, they are *relinked* into the target file while retaining existing memory-mapped regions.

operation, we write one cache line (64B) worth of data to PM and use a single memory fence (sfence in x86) instruction in the process. Operation log entries do not contain the file data associated with the operation (e.g., data being appended to a file), instead they contain a logical pointer to the staging file where the data is being held.

We employ a number of techniques to optimize logging. First, to distinguish between valid and invalid or torn log entries, we incorporate a 4B transactional checksum [77] within the 64B log entry. The use of checksum reduces the number of fence instructions necessary to persist and validate a log entry from two to one. Second, we maintain a tail for the log in DRAM and concurrent threads use the tail as a synchronization variable. They use compare-and-swap to atomically advance the tail and write to their respective log entries concurrently. Third, during the initialization of the operation log file, we zero it out. So, during crash recovery, we identify all non-zero 64B aligned log entries as being potentially valid and then use the checksum to identify any torn entries. The rest are valid entries and are replayed. Replaying log entries is idempotent, so replaying them multiple times on crashes is safe. We employ a 128MB operation log file and if it becomes full, we checkpoint the state of the application by calling relink() on all the open files that have data in staging files. We then zero out the log and reuse it. Finally, we designed our logging mechanism such that all common case operations (`write()`, `open()`, etc.) can be logged using a single 64B log entry while some uncommon operations, like `rename()`, require multiple log entries.

Our logging protocol works well with the SplitFS architecture. The tail of each U-Split log is maintained only in DRAM as it is not required

for crash recovery. Valid log entries are instead identified using checksums. In contrast, file systems such as NOVA have a log per inode that resides on PM, whose tail is updated after each operation via expensive clflush and sfence operations.

Providing Atomic Operations. In strict mode, SplitFS provides synchronous, atomic operations. Atomicity is provided in an efficient manner by the combination of staging files, relink, and optimized logging. Atomicity for data operations like overwrites is achieved by redirecting them also to a staging file, similar to how appends are performed. SplitFS logs these writes and appends to record where the latest data resides in the event of a crash. On `fsync()`, SplitFS relinks the data from the staging file to the target file atomically. Once again, the data is written exactly once, though SplitFS provides the strong guarantee of atomic data operations. Relink allows SplitFS to implement a form of localized copy-on-write. Due to the staging files being pre-allocated, locality is preserved to an extent. SplitFS logs metadata operations to ensure they are atomic and synchronous. Optimized logging ensures that for most operations exactly one cache line is written and one `sfence` is issued for logging.

4.5.1 Handling reads, Overwrites and Appends

Reads. Reads consult the collection of mmaps to determine where the most recent data for this offset is, since the data could have been overwritten or appended (and thus in a staging file). If a valid memory mapped region for the offsets being read exists in U-Split, the read is serviced from the corresponding region. If such a region does not exist, then the 2 MB region surrounding the read offset is first memory mapped, added to the

the collection of mmaps, and then the read operation is serviced using processor loads.

Overwrites. Similar to reads, if the target offset is already memory mapped, then U-Split services the overwrite using non-temporal store instructions. If the target offset is not memory mapped, then the 2MB region surrounding the offset is first memory mapped, added to the collection of mmaps, and then the overwrite is serviced. However, in strict mode, to guarantee atomicity, overwrites are first redirected to a staging file (even if the offset is memory mapped), then the operation is logged, and finally relinked on a subsequent `fsync()` or `close()`.

Appends. SplitFS redirects all appends to a staging file, and performs a relink on a subsequent `fsync()` or `close()`. As with overwrites, appends are performed with non-temporal writes and in strict mode, SplitFS also logs details of the append operation to ensure atomicity.

4.6 SplitFS Implementation

We implement SplitFS as a combination of a user-space library file system (9K lines of C code) and a small patch to ext4 DAX to add the relink system call (500 lines of C code). SplitFS supports 35 common POSIX calls, such as pwrite(), pread(), fread(), readv(), ftruncate(), openat(), etc; we found that supporting this set of calls is sufficient to support a variety of applications and microbenchmarks. Since PM file systems PMFS and NOVA are supported by Linux kernel version 4.13, we modified 4.13 to support SplitFS. We now present other details of our implementation.

Intercepting POSIX calls. SplitFS uses LD_PRELOAD to intercept

POSIX calls and either serve from user-space or route them to the kernel after performing some book-keeping tasks. Since SplitFS intercepts calls at the POSIX level in glibc rather than at the system call level, SplitFS has to intercept several variants of common system calls like `write()`.

Relink. We implement relink by leveraging an ioctl provided by ext4 DAX. The ioctl swaps extents between a source file and a destination file, and uses journaling to perform this atomically. The ioctl also deallocates blocks in the target file if they are replaced by blocks from the source file. By default, the ioctl also flushes the swapped data in the target file; we modify the ioctl to only touch metadata, without copying, moving, or persisting data. We also ensure that after the swap has happened, existing memory mappings of both source and destination files are valid; this is vital to SplitFS performance, as it avoids page faults. The ioctl requires blocks to be allocated at both source and destination files. To satisfy this requirement, when handling appends via relink, we allocate blocks at the destination file, swap extents from the staging file, and then deallocate the blocks. This allows us to perform relink without using up extra space, and reduces implementation complexity at the cost of temporary allocation of data.

Handling file open and close. On file open, SplitFS performs `stat()` on the file and caches its attributes in user-space to help handle later calls. When a file is closed, we do not clear its cached information. When the file is unlinked, all cached metadata is cleared, and if the file has been memory-mapped, it is un-mapped. The cached attributes are used to check file permissions on every subsequent file operation (e.g., `read()`) intercepted by U-Split.

Handling fork. Since SplitFS uses a user-space library file system, special care needs to be taken to handle `fork()` and `execve()` correctly. When `fork()` is called, SplitFS is copied into the address space of the new process (as part of copying the address space of the parent process), so that the new process can continue to access SplitFS.

Handling execve. `execve()` overwrites the address space, but open file descriptors are expected to work after the call completes. To handle this, SplitFS does the following: before executing `execve()`, SplitFS copies its in-memory data about open files to a shared memory file on /dev/shm; the file name is the process ID. After executing `execve()`, SplitFS checks the shared memory device and copies information from the file if it exists.

Handling dup. When a file descriptor is duplicated, the file offset is changed whenever operations are performed on either file descriptor. SplitFS handles by maintaining a single offset per open file, and using pointers to this file in the file descriptor maintained by SplitFS. Thus, if two threads dup a file descriptor and change the offset from either thread, SplitFS ensures both threads see the changes.

Staging files. SplitFS pre-allocates staging files at startup, creating 10 files each 160 MB in size. Whenever a staging file is completely utilized, a background thread wakes up and creates and pre-allocates a new staging file. This avoids the overhead of creating staging files in the critical path.

Cache of memory-mappings. SplitFS caches all memory-mappings it creates in its collection of memory mappings. A memory-mapping is only discarded on `unlink()`. This reduces the cost of setting up memory mappings in the critical path on read or write.

Multi-thread access. SplitFS uses a lock-free queue for managing the staging files. It uses fine-grained reader-writer locks to protect its in-memory metadata about open files, inodes, and memory-mappings.

4.6.1 Tunable Parameters

SplitFS provides a number of tunable parameters that can be set by application developers and users for each U-Split instance. These parameters affect the performance of SplitFS.

`mmap()` size. SplitFS supports a configurable size of `mmap()` for handling overwrites and reads. Currently, SplitFS supports `mmap()` sizes ranging from 2MB to 512MB. The default size is 2 MB, allowing SplitFS to employ huge pages while pre-populating the mappings.

Number of staging files at startup. There are ten staging files at startup by default; when a staging file is used up, SplitFS creates another staging file in the background. We experimentally found that having ten staging files provides a good balance between application performance and the initialization cost and space usage of staging files.

Size of the operation log. The default size of the operation log is 128MB for each U-Split instance. Since all log entries consist of a single cacheline in the common case, SplitFS can support up to 2M operations without clearing the log and re-initializing it. This helps applications with small bursts to achieve good performance while getting strong semantics.

4.6.2 Security

SplitFS does not expose any new security vulnerabilities as compared to an in-kernel file system. All metadata operations are passed through to the

kernel, which performs security checks. SplitFS does not allow a user to open, read, or write a file to which they previously did not have permissions. The U-Split instances are isolated from each other in separate processes; therefore applications cannot access the data of other applications while running on SplitFS. Each U-Split instance only stores book-keeping information in DRAM for the files that the application already has access to. An application that uses SplitFS may corrupt its own files, just as in an in-kernel file system.

4.7 Discussion

We reflect on our experiences building SplitFS, describe problems we encountered, how we solved them, and surprising insights that we discovered.

Page faults lead to significant cost. SplitFS memory maps files before accessing them, and uses `MAP_POPULATE` to pre-fault all pages so that later reads and writes do not incur page-fault latency. As a result, we find that a significant portion of the time for `open()` is consumed by page faults. While the latency of device IO usually dominates page fault cost in storage systems based on solid state drives or magnetic hard drives, the low latency of persistent memory highlights the cost of page faults.

Huge pages are fragile. A natural way of minimizing page faults is to use 2 MB huge pages. However, we found huge pages fragile and hard to use. Setting up a huge-page mapping in the Linux kernel requires a number of conditions. First, the virtual address must be 2 MB aligned. Second, the physical address on PM must be 2 MB aligned. As a result, fragmentation in either the virtual address space or the physical PM prevents huge pages from being created. For most workloads, after a few thousand files were

created and deleted, fragmenting PM, we found it impossible to create any new huge pages. Our collection-of-mappings technique sidesteps this problem by creating huge pages at the beginning of the workload, and reusing them to serve reads and writes. Without huge pages, we observed read performance dropping by 50% in many workloads. We believe this is a fundamental problem that must be tackled since huge pages are crucial for accessing large quantities of PM.

Staging writes in DRAM. An alternate design that we tried was staging writes in DRAM instead of on PM. While DRAM staging files incur lower allocation costs than PM staging files, we found that the cost of copying data from DRAM to PM on `fsync()` overshadowed the benefit of staging data in DRAM. In general, DRAM buffering is less useful in PM systems because PM and DRAM performance is similar.

Legacy applications need to be rewritten to take maximum advantage of PM. We observe that the applications we evaluate such as LevelDB spent a significant portion of their time ($60-80\%$) performing POSIX calls on current PM file systems. SplitFS is able to reduce this percentage down to 46-50%, but further reduction in software overhead will have negligible impact on application runtime since the majority of the time is spent on application code. Applications would need to be rewritten from scratch to use libraries like `libpmem` that exclusively operate on data structures in `mmap()` to take further advantage of PM.

4.8 SplitFS Evaluation

In this section, we use a number of microbenchmarks and applications to evaluate SplitFS in relation to state-of-the-art PM filesystems like ext4

DAX, NOVA, and PMFS. While comparing these different file systems, we seek to answer the following questions:

- How does SplitFS affect the performance of different system calls as compared to ext4 DAX? (§4.8.4)
- How do the different techniques employed in SplitFS contribute to overall performance? (§4.8.5)
- How does SplitFS compare to other file systems for different PM access patterns? (§4.8.6)
- Does SplitFS reduce file-system software overhead as compared to other PM file systems? (§4.8.7)
- How does SplitFS compare to other file systems for real-world applications? (§4.8.8 & §4.8.9)
- What are the compute and storage overheads incurred when using SplitFS? (§4.8.10)

We first briefly describe our experimental methodology before addressing each of the above questions.

4.8.1 Experimental Setup

We evaluate the performance of SplitFS against other PM file systems on Intel Optane DC Persistent Memory Module (PMM). The experiments are performed on a 2-socket, 96-core machine with 768 GB PMM, 375 GB DRAM, and 32 MB Last Level Cache (LLC). We run all evaluated file systems on the 4.13 version of the Linux kernel (Ubuntu 16.04). We run each experiment multiple times and report the mean. In all cases,

Application	Description
TPC-C [78] on SQLite [79]	Online transaction processing
YCSB [80] on LevelDB [81]	Data retreival & maintenance
Set in Redis [23]	In-memory data structure store
Git	Popular version control software
Tar	Linux utility for data compression
Rsync	Linux utility for data copy

Table 4.3: **Applications used in evaluation.** The table provides a brief description of the real-world applications we use to evaluate PM file systems.

the standard deviation was less than five percent of the mean, and the experiments could be reliably repeated.

4.8.2 Workloads

We used two key-value stores (Redis, LevelDB), an embedded database (SQLite), and three utilities (tar, git, rsync) to evaluate the performance of SplitFS. Table 4.3 lists the applications and their characteristics.

TPC-C on SQLite. TPC-C [78] is an online transaction processing benchmark. It has five different types of transactions each with different ratios of reads and writes. We run SQLite v3.23.1 [79] with SplitFS, and measure the performance of TPC-C on SQLite in Write-Ahead-Logging (WAL) mode.

YCSB on LevelDB. The Yahoo Cloud Serving Benchmark [80] has six different key-value store benchmarks, each with different read/write ratios. Table 4.4 shows the details of the different workloads of YCSB. We run all the YCSB workloads on the LevelDB key-value store [81]. We set the `sstable` size to 64 MB as recommended in Facebook's tuning guide [82].

Redis. We set 1M key-value pairs in Redis [23], an in-memory key-value

Workload	***Description***	***Represents***
Load A	100% writes	Insert data for workloads A–D and F
A	50% reads, 50% writes	Session recording recent actions
B	95% reads, 5% writes	Browsing and tagging photo album
C	100% reads	Caches
D	95% reads (latest values), 5% writes	News feed or status feed
Load E	100% writes	Insert data for Workload E
E	95% Range queries, 5% writes	Threaded conversation
F	50% reads, 50% Read-modify-writes	Database workload

Table 4.4: **YCSB Workloads.** The table describes the six workloads in the YCSB suite. Workloads A–D and F are preceded by Load A, while E is preceded by Load E.

store. We run Redis in Append-Only-File mode, where it logs updates to the database in a file and performs `fsync()` on the file every second.

Utilities. We also evaluate the performance of SplitFS for tar, git, and rsync. With git, we measured the time taken for `git add` and `git commit` of all files in the Linux kernel ten times. With rsync, we copy a 7 GB dataset of 1200 files with characteristics similar to backup datasets [83] from one PM location to another. With tar, we compressed the Linux kernel 4.18 along with the files from the backup dataset.

4.8.3 Correctness and recovery

Correctness. First, to validate the functional correctness of SplitFS we run various micro-benchmarks and real-world applications and compare the resulting file-system metadata such as the number of files and contents of each file, to the one obtained with ext4 DAX. We observe that the file-system metadata obtained with ext4 DAX and SplitFS are equivalent,

validating how SplitFS handles POSIX calls in its user-space library file system.

Recovery times. Crash recovery in POSIX and sync modes of SplitFS does not require anything beyond allowing the underlying ext4 DAX file system to recover. In strict mode, however, all valid log entries in the operation log need to be replayed on top of ext4 DAX recovery. This additional log replay time depends on the number and type of valid log entries in the log. To estimate the additional time needed for recovery, we crash our real-world workloads at random points in their execution and measure the log replay time. In our crash experiments, the maximum number of log entries to be replayed was 18,000 and that took about 3 seconds on emulated PM (emulation details in §4.8.8). In a worst-case micro-benchmark where we perform cache-line sized writes and crash with 2M (128MB of data) valid log entries, we observed a log replay time of 6 seconds on emulated PM.

4.8.4 SplitFS system call overheads

The central premise of SplitFS is that it is a good trade-off to accelerate data operations at the expense of metadata operations. Since data operations are more prevelant, this optimization improves overall application performance. To validate this premise, we construct a micro-benchmark similar to FileBench Varmail [84] that issues a variety of data and metadata operations. The micro-benchmark first creates and appends 16KB to a file (as four appends, each followed by an `fsync()`), closes it, opens it again, read the whole file as one read call, closes it, then opens and closes the file once more, and finally deletes the file. The multiple open and close

System call	Strict	Sync	POSIX	ext4 DAX
open	2.09	2.08	1.82	1.54
close	0.78	0.69	0.69	0.34
append	3.14	3.09	2.84	11.05
fsync	6.85	6.80	6.80	28.98
read	4.57	4.53	4.53	5.04
unlink	14.60	13.56	14.33	8.60

Table 4.5: **SplitFS system call overheads.** The table compares the latency (in us) of different system calls for various modes of SplitFS and ext4 DAX.

calls were introduced to account for the fact that their latency varies over time. Opening a file for the first time takes longer than opening a file that we recently closed, due to file metadata caching inside U-Split. Table 4.5 shows the latencies we observed for different system calls and they are reported for all the three modes provided by SplitFS and for ext4 DAX on which SplitFS was built.

We make three observations based on these results. First, data operations on SplitFS are significantly faster than on ext4 DAX. Writes especially are 3–4× faster. Second, metadata operations (*e.g.,* `open()`, `close()`, etc.) are slower on SplitFS than on ext4 DAX, as SplitFS has to setup its own data structures in addition to performing the operation on ext4 DAX. Third, as the consistency guarantees provided by SplitFS get stronger, the syscall latency generally increases. This increase can be attributed to the additional work SplitFS has to do (*e.g.,* logging in strict mode) for each system call to provide stronger guarantees. Overall, SplitFS achieves its objective of accelerating data operations albeit at the expense of metadata operations.

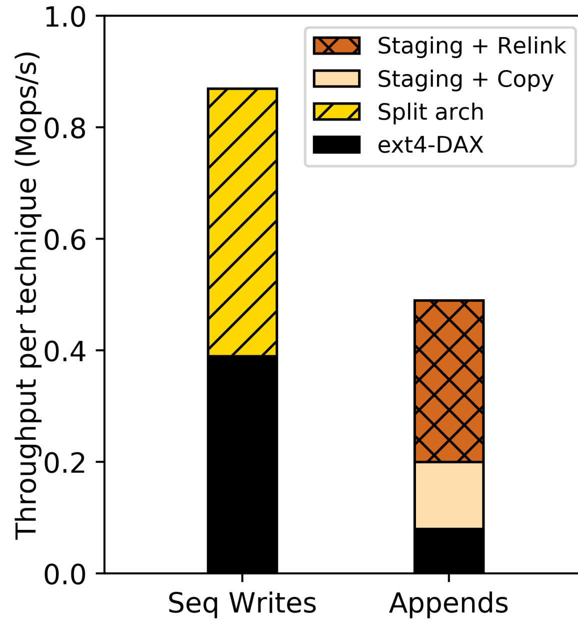

Figure 4.3: **SplitFS techniques.** This figure shows the contributions of different techniques to overall performance. We compare the relative merits of these techniques using two write intensive microbenchmarks; sequential overwrites and appends.

4.8.5 SplitFS performance breakdown

We examine how the various techniques employed by SplitFS contribute to overall performance. We use two write-intensive microbenchmarks: sequential 4KB overwrites and 4KB appends. An `fsync()` is issued every ten operations. Figure 4.3 shows how individual techniques introduced one after the other improve performance.

Sequential overwrites. SplitFS increases sequential overwrite performance by more than 2× compared to ext4 DAX since overwrites are served from user-space via processor stores. However, further optimizations like handling appends using staging files and relink have negligible impact on this workload as it does not issue any file append operations.

Appends. The split architecture does not accelerate appends since with-

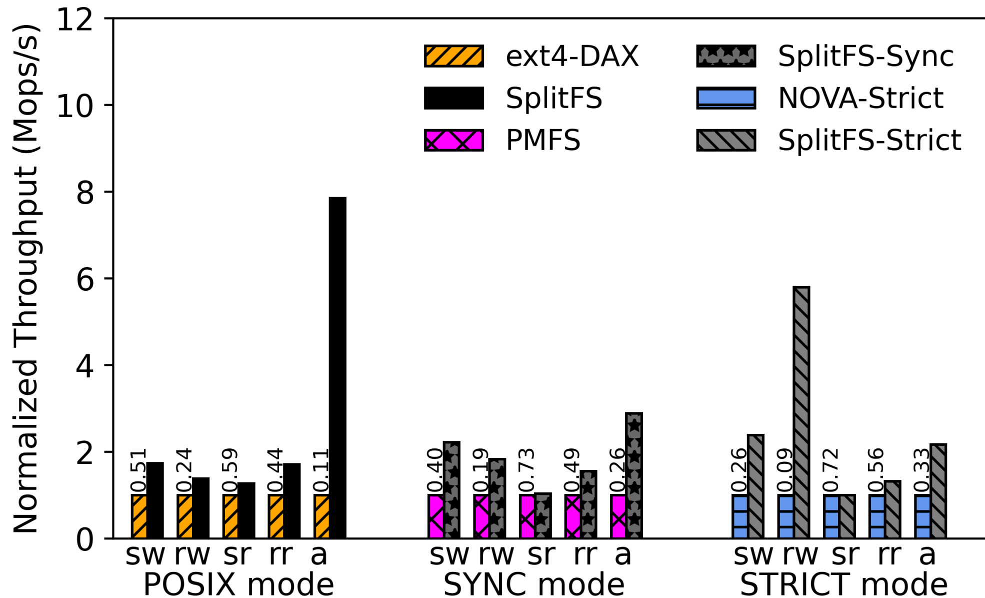

Figure 4.4: **Performance on different IO patterns.** This figure compares SplitFS with the state-of-the-art PM file systems in their respective modes using micrbenchmarks that perform five different kinds of file access patters. The y-axis is throughput normalized to ext4 DAX in POSIX mode, PMFS in sync mode, and NOVA-Strict in Strict mode (higher is better). The absolute throughput numbers in Mops/s are given over the baseline in each group.

out staging files or relink all appends go to ext4 DAX as they are metadata operations. Just introducing staging files to buffer appends improves performance by about 2×. In this setting, even though appends are serviced in user-space, overall performance is bogged down by expensive data copy operations on `fsync()`. Introducing the relink primitive to this setting eliminates data copies and increases application throughput by 5×.

4.8.6 Performance on different IO patterns

To understand the relative merits of different PM file systems, we compare their performance on microbenchmarks performing different file IO patterns: sequential reads, random reads, sequential writes, random writes,

and appends. Each benchmark reads/writes an entire 128MB file in 4KB operations. We compare file systems providing the same guarantees: SplitFS-POSIX with ext4 DAX, SplitFS-sync with PMFS, and SplitFS-strict with Nova-strict and Strata. Figure 4.4 captures the performance of these file systems for the different micro-benchmarks.

POSIX mode. SplitFS is able to reduce the execution times of ext4 DAX by at least 27% and as much as 7.85× (sequential reads and appends respectively). Read-heavy workloads present fewer improvement opportunities for SplitFS as file read paths in the kernel are optimized in modern PM file systems. However, write paths are much more complex and longer, especially for appends. So, servicing a write in user-space has a higher payoff than servicing a read, an observation we already made in Table 4.5.

Sync mode. Compared to PMFS, SplitFS improves the performance for write workloads (by as much as 2.89×) and increases performance for read workloads (by as much as 56%). Similar to ext4 DAX, SplitFS's ability to not incur expensive write system calls translates to its superior performance for the write workloads.

Strict mode. NOVA, Strata, and SplitFS in this mode provide atomicity guarantees to all operations and perform the necessary logging. As can be expected, the overheads of logging result in reduced performance compared to file systems in other modes. Overall, SplitFS improves the performance over NOVA by up to 5.8× on the random writes workload. This improvement stems from SplitFS's superior logging which incurs half the number of log writes and fence operations than NOVA.

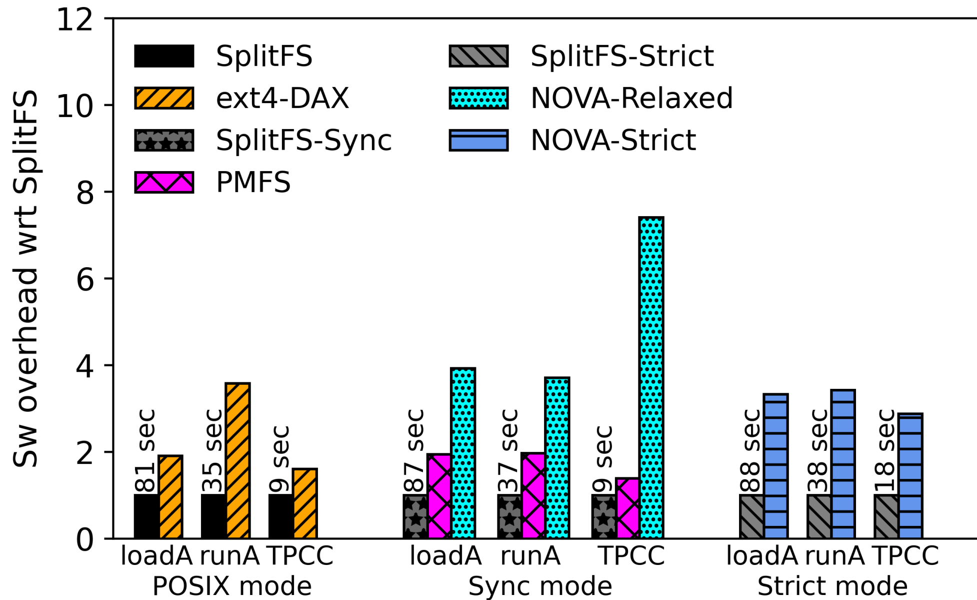

Figure 4.5: **Software overhead in applications.** This figure shows the relative file system software overhead incurred by different applications with various file systems as compared providing the same level of consistency guarantees (lower is better). The numbers shown indicate the absolute time taken to run the workload for the baseline file system.

4.8.7 Reducing software overhead

The central premise of SplitFS is that it is possible to accelerate applications by reducing file system software overhead. We define *file-system software overhead* as the time taken to service a file-system call minus the time spent actually accessing data on the PM device. For example, if a system call takes 100 s to be serviced, of which only 25 μs were spent read or writing to PM, then we say that the software overhead is 75 μs. In addition to avoiding kernel traps of system calls, the different techniques discussed in §4.5 help SplitFS reduce its software overhead. Minimizing software overhead allows applications to fully leverage PMs.

Figure 4.5 highlights the relative software overheads incurred by different file systems compared to SplitFS providing the same level of guaran-

tees. We present results for three write-heavy workloads, LevelDB running YCSB Load A and Run A, and SQLite running TPCC. ext4 DAX and NOVA (in relaxed mode) suffer the highest relative software overheads, up to 3.6× and 7.4× respectively. NOVA-Relaxed incurs the highest software overhead for TPCC because it has to update the per-inode logical log entries on overwrites before updating the data in-place. On the other hand, SplitFS-sync can directly perform in-place data updates, and thus has significantly lower software overhead. PMFS suffers the lowest relative software overhead, capping off at 1.9× for YCSB Load A and Run A. Overall, SplitFS incurs the lowest software overhead.

4.8.8 Performance on data intensive workloads

Figure 4.6 summarizes the performance of various applications on different file systems. The performance metric we use for these data intensive workloads (LevelDB with YCSB, Redis with 100% writes, and SQLite with TPCC) is throughput measured in Kops/s. For each mode of consistency guarantee (POSIX, sync, and strict), we compare SplitFS to state-of-the-art PM file systems. We report the absolute performance for the baseline file system in each category and relative throughput for SplitFS. Despite our best efforts, we were not able to run Strata on these large applications; other researchers have also reported problems in evaluating Strata [56]. We evaluated Strata with a smaller-scale YCSB workload using a 20GB private log.

Overall, SplitFS outperforms other PM file systems (when providing similar consistency guarantees) on all data-intensive workloads by as much as 2.70×. We next present a breakdown of these numbers for different

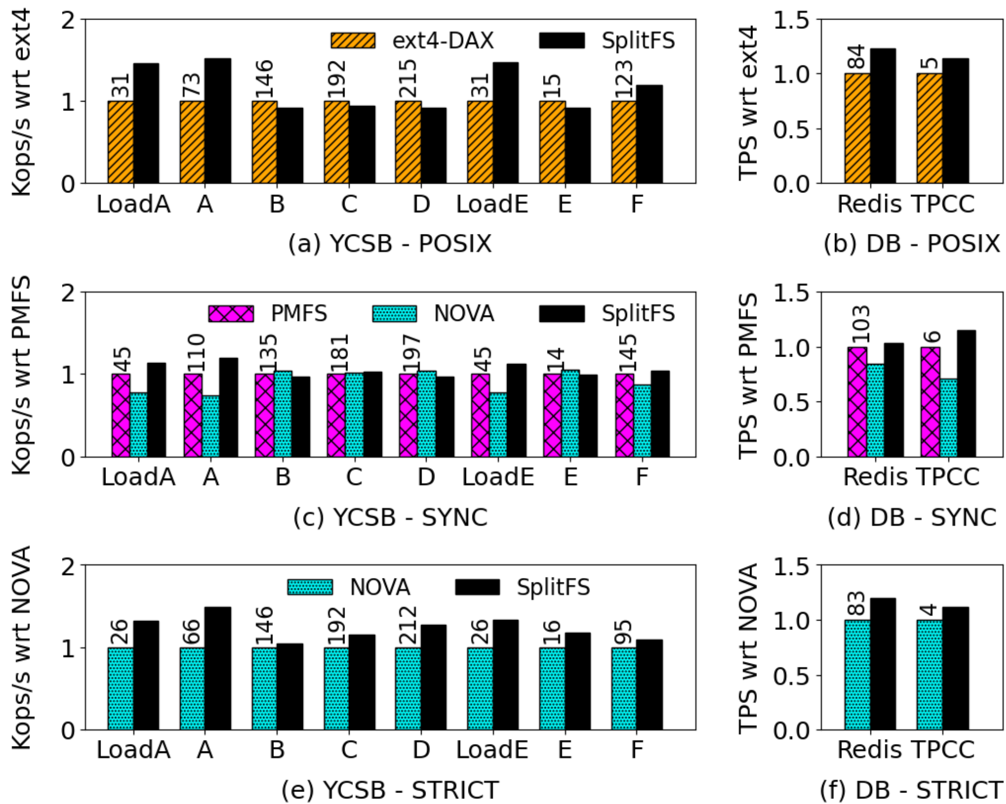

Figure 4.6: **Data Intensive Workloads.** This figure shows the performance of data intensive applications (YCSB, Redis, and TPCC) with different file systems, providing three different consistency guarantees, POSIX, sync, and strict. Overall, SplitFS beats all other file systems on all data intensive applications (in their respective modes). The numbers indicate the absolute throughput in Kops/s for the base file system.

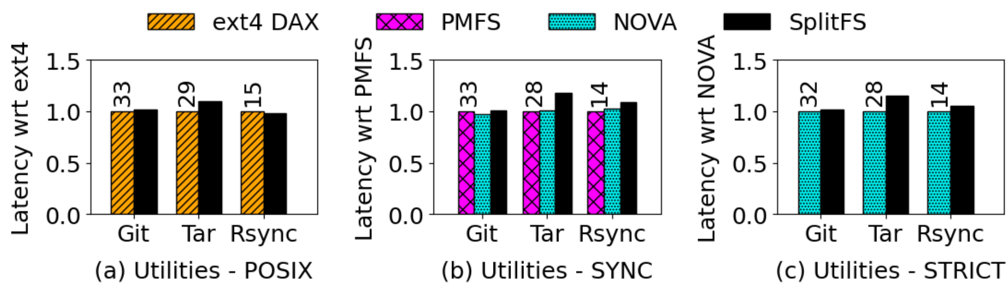

Figure 4.7: **Metadata Intensive Workloads.** This figure shows the performance of metadata instensive utilities (git, tar, and rsync) with different file systems, providing three different consistency guarantees, POSIX, sync, and strict. Overall, SplitFS incurs minor performance degradation on metadata heavy workloads. The numbers indicate latency in seconds for the base file system.

Workload	Strata	
Load A	29.1 kops/s	1.73×
Run A	55.2 kops/s	1.76×
Run B	76.8 kops/s	2.16×
Run C	94.3 kops/s	2.14×
Run D	113.1 kops/s	2.25×
Load E	29.1 kops/s	1.72×
Run E	8.1 kops/s	2.03×
Run F	73.3 kops/s	2.25×

Table 4.6: **SplitFS vs. Strata.** This table compares the performance of Strata and SplitFS strict running YCSB on LevelDB. We present the raw throughput numbers for Strata and normalized SplitFS strict throughput w.r.t Strata. This is the biggest workload that we could run reliably on Strata.

guarantees.

POSIX mode. SplitFS outperforms ext4 DAX in all workloads. Write-heavy workloads like RunA (2×), LoadA (89%), LoadE (91%), Redis (27%), etc. benefit the most with SplitFS. SplitFS speeds up writes and appends the most, so write-heavy workloads benefit the most from SplitFS. SplitFS outperforms ext4 DAX on read-dominated workloads, but the margin of improvement is lower.

Sync and strict mode. SplitFS outperforms sync-mode file systems PMFS and NOVA (relaxed) and strict-mode file system NOVA (strict) for all the data intensive workloads. Once again, it's the write-heavy workloads that show the biggest boost in performance. For example, SplitFS in sync mode outperforms NOVA (relaxed) and PMFS by 2× and 30% on RunA and in strict mode outperforms NOVA (strict) by 2×. Ready-heavy workloads on the other hand do not show much improvement in performance.

Comparison with Strata. We were able to reliably evaluate Strata (employing a 20 GB private log) using LevelDB running smaller-scale YCSB workloads (1M records, and 1M ops for workloads A–D and F, 500K ops for workload E). We were unable to run Strata on Intel DC Persistent Memory. Hence, we use DRAM to emulate PM. We employ the same PM emulation framework used by Strata. We inject a delay of 220ns on every `read()` system call, to emulate the access latencies of the PM hardware. We do not add this fixed 220ns delay for writes, because writes do not go straight to PM in the critical path, but only to the memory controller. We add bandwidth-modeling delays for reads as well as writes to emulate a memory device with $1/3^{rd}$ the bandwidth of DRAM, an expected characteristic of PMs [35]. While this emulation approach is far from perfect, we observe that the resulting memory access characteristics are in line with the expected behavior of PMs [58]. SplitFS outperforms Strata on all workloads, by 1.72×–2.25× as shown in Table 4.6.

4.8.9 Performance on metadata intensive workloads

Fig 4.7 compares the performance of SplitFS with other PM file systems (we only show the best performing PM file system) on metadata-heavy workloads like git, tar, and rsync. These metadata-heavy workloads do not present many opportunities for SplitFS to service system calls in user-space and in turn slow metadata operations down due to the additional bookkeeping performed by SplitFS. These workloads represent the worst case scenarios for SplitFS. The maximum overhead experienced by SplitFS is 13%.

4.8.10 Resource consumption

SplitFS consumes memory for its file-related metadata (e.g. to keep track of open file descriptors, staging files used). It also additionally consumes CPU time to execute background threads that help with metadata management and to move some expensive tasks off the application's critical path.

Memory usage. SplitFS uses a maximum of 100MB to maintain its own metadata to help track different files, the mappings between file offsets and `mmap()`-ed regions, etc. In strict mode, SplitFS additionally uses 40MB to maintain data structures to provide atomicity guarantees.

CPU utilization. SplitFS uses a background thread to handle various deferred tasks (e.g. staging file allocation, file closures). This thread utilizes one physical thread of the machine, occasionally increasing CPU consumption by 100%.

4.9 Conclusion

This chapter presents SplitFS, a PM file system built using the split architecture. SplitFS handles data operations entirely in user-space, and routes metadata operations through the ext4 DAX PM file system. SplitFS provides three modes with varying guarantees, and allows applications running at the same time to use different modes. SplitFS only requires adding a single system call to the ext4 DAX file system. Evaluating SplitFS with micro-benchmarks and real applications, we show that it outperforms state-of-the-art PM file systems like NOVA on many workloads. The design of SplitFS allows users to benefit from the maturity and constant devel-

opment of the ext4 DAX file system, while getting the performance and strong guarantees of state-of-the-art PM file systems. SplitFS is publicly available at `https://github.com/utsaslab/splitfs`.

Chapter 5

Achieving high performance for modern PM applications

In this chapter, we target a second class of applications that access PM using loads and stores on memory mapped files from user space. We saw in Chapter 3 that the performance of these applications depends on whether the applications are able to get hugepages for the memory mapped files. We also studied the impact of file systems on getting hugepages and the limitations of existing file systems to reliably get hugepages as they age.

In this chapter, we first discuss the need for a hugepage-aware PM file system for accelerating modern PM applications (§5.1). We introduce WineFS, a PM file system that relies on a novel allocation policy and on-PM layout to achieve hugepage awareness. We then present the goals of WineFS (§5.3). We then discuss the overview of the design of WineFS (§5.4). We discuss how WineFS achieves hugepage awareness (§5.4.3) and high scalability (§5.4.4), along with its implementation (§5.5). Finally, we evaluate WineFS against existing file systems (§5.7).

5.1 Building a hugepage-aware file system

The performance of applications that issue loads and stores on memory mapped files depends on whether the files can be mapped using hugepages.

Getting hugepages involves placing file extents on hugepage aligned boundaries on PM. Due to misalignment and fragmentation of the free space and file extents with frequent allocations and deallocations, the free space contiguity must be maintained by periodic defragmentation of files. We believe that it is a better approach to be proactive about conserving hugepages, rather than reactively defragmenting files. We require support from the file system to conserve aligned extents. Current PM file systems do not optimize for this goal. Some file systems such as Strata [58] and NOVA [37] make it harder to map files using hugepages due to their log-structured nature. NOVA could be modified to become hugepage-aware, but would require non-trivial changes to its design which would reduce its performance. For example, dedicating on-PM regions for the per-file journals would increase the load on garbage collection and its interference with foreground threads. Changing the copy-on-write granularity of NOVA to the size of hugepages to avoid fragmentation would result in increased write as well as space amplification.

Mature file systems such as ext4 DAX [38] or xfs-DAX [39] have allocators that care more about contiguity than alignment, which makes them sacrifice hugepages as part of their design. Additionally, in order to achieve high performance for a wide range of legacy as well as newer PM applications, the mature file systems would have to change several fundamental components such as incorporating a hugepage-aware allocator, devising an on-PM layout that avoids fragmentation of free space, supporting low-cost data atomicity and adding fine-grained low-cost journaling for crash consistency.

Designing a hugepage-aware PM file system requires revisiting all as-

pects of file system design from the lens of hugepage-awareness, rather than tweaking one or two aspects of an existing file system.

5.2 WineFS: A hugepage-aware file system that ages gracefully

We build WineFS, a hugepage-aware PM file system that ages gracefully. WineFS uses a novel allocation policy that takes into account alignment of free space and contiguity of free space, reliably getting hugepages for memory mapped files even when aged. WineFS uses a suitable on-PM layout and a hybrid crash consistency mechanism that avoids fragmentation while providing strong consistency and durability guarantees for data and metadata.

5.3 WineFS Goals

- WineFS must be POSIX-compliant, and should not require any changes in the application.
- WineFS must try to provide hugepage-sized extents aligned at the hugepage boundary for files that are memory-mapped.
- WineFS must not sacrifice performance of applications that use POSIX system calls to access PM.
- WineFS must not sacrifice performance when the file system is new, either for memory-mapped or system-call access to PM.
- The design of WineFS must seek to preserve hugepages wherever possible.

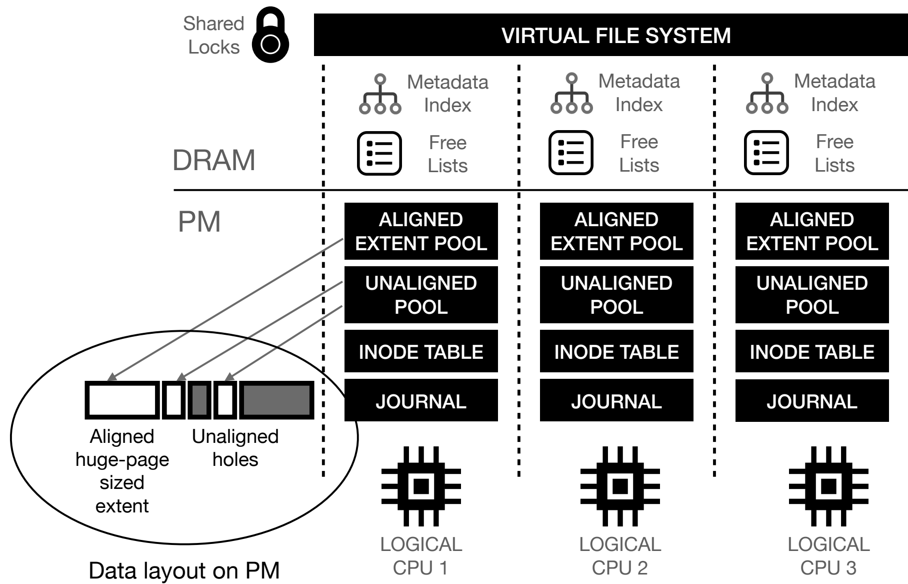

Figure 5.1: **WineFS Architecture.** The figure shows the main components of WINEFS. WINEFS partitions the file system per logical CPU for concurrency. Each logical CPU has its own journal, inode table, and free lists for aligned extents and holes. WINEFS uses DRAM indexes for metadata for efficient directory and resource lookups. The shared locks in the VFS layer help WINEFS coordinate its multiple journals.

- WINEFS must provide strong guarantees such as atomic, synchronous operations, similar to the strict mode of SPLITFS.

5.4 WineFS Design

We provide an overview of the design of WINEFS, and describe the design choices that lead to hugepage awareness and graceful aging.

5.4.1 Overview

WINEFS achieves these goals through a set of design choices. WINEFS achieves hugepage-awareness through:

- A novel alignment-aware allocator that satisfies large allocation requests using aligned extents, and smaller requests using unaligned holes.
- Using a PM data layout with contained fragmentation: metadata structures and journals that are updated in-place.
- Using journaling for crash consistency as it preserves data layout, even at the cost of writing metadata twice to PM.
- Using per-CPU metadata structures (rather than per-file) for obtaining concurrent updates.
- Using data journaling to atomically update files with aligned extents, and using copy-on-write to atomically update holes.

WineFS ensures good performance for applications that access PM via POSIX system calls through a *second* set of design choices. The key observation is this second list is chosen such that it composes well with first list. For example, this would not be the case if WineFS had used log-structuring for accelerating metadata updates. The design choices:

- WineFS uses fine-grained journaling optimized for PM.
- WineFS uses DRAM metadata indexes to accelerate operations such as directory lookups.

Figure 5.1 presents an overview of the WineFS architecture and how all these design choices come together.

5.4.2 Guarantees

WineFS can be run in two modes: strict mode and relaxed mode. The default is the strict mode. The mode can be changed using mount options.

Strict Mode. All file system operations, both data operations and metadata operations, are atomic and synchronous. Upon completion of each `write()` system call, the data involved is guaranteed to be durable. NOVA, SplitFS-strict, and Strata provide the same guarantees.

Relaxed Mode. All metadata operations such as `rename()` are atomic and synchronous. Data operations are not atomic, and may be partially completed on a crash. ext4 DAX, xfs, and PMFS provide the same guarantees.

5.4.3 Hugepage Awareness

We now describe in detail the design choices that make WineFS a hugepage-aware PM file system.

Data Layout: Controlled Fragmentation. WineFS uses controlled fragmentation in designing its data layout on PM. Typically, metadata structures cause significant fragmentation as they tend to be small. For example, NOVA has a per-file log that causes fragmentation, using up an aligned extent. WineFS tries to control the fragmentation caused by metadata structures, by assigning dedicated locations for metadata structures. The metadata structures are updated in-place within these locations. The space in these locations is recycled for other metadata structures.

Concurrency: Per-CPU data pool and metadata structures. Closely related to the data layout is how WineFS achieves concurrency. NOVA

achieves high concurrency by providing a per-file log. While this provides high concurrency, it also fragments the free space and uses up aligned extents. WineFS chooses a different design: per-CPU journal, data structures, and data and metadata pools. The PM free space is divided by the number of logical CPUs. Each logical CPU gets its own journal, inode table, and pool of aligned extents and unaligned holes. Each CPU maintains free lists in DRAM that keep track of free inodes and extents in the CPU's own pool of inodes and extents. In the common case, an allocation request arising at a logical CPU can be handled locally, without any communication with other logical CPUs. Since the unit of parallelism is a logical CPU, rather than a file, this design provides high concurrency without incurring fragmentation. Experimental results show that WineFS scales as well as NOVA, while being more hugepage-friendly.

A natural question that arises is: how are the per-CPU journals coordinated? WineFS uses the Virtual File System (VFS) layer for coordination. The namespace is shared across all logical CPUs, and VFS provides shared locks for directory inodes, while WineFS holds locks for file modification operations such as `write()` or `fallocate()`. An inode can only be locked by one logical CPU at a time. WineFS ensures that all file system operations that require journaling also grab inode locks, implying that a file can only be part of one per-CPU journal at a given time. Different files can be locked by different CPUs and journaled concurrently.

Allocation: Alignment-Aware Allocation. WineFS uses a novel alignment-aware allocator. The allocator splits the entire partition into aligned hugepage-sized extents. Incoming allocation requests are split into requests of hugepage sizes or below. Large allocation requests (that are

hugepage-sized) are satisfied using an aligned extent. Small allocation requests, less than the size of a hugepage, are satisfied by small, unaligned holes. If required, a single aligned extent is broken up to satisfy small allocation requests.

The allocator uses two data structures in DRAM to help with allocation. One is a linked list of aligned extents, and the other is a red-black for finding unaligned extents quickly. These data structures are written to PM on unmount. On a crash, they are re-initialized by scanning the set of used inodes in the file system (similar to NOVA).

The allocator uses the following policy to decide which extent to utilize for an allocation request. It always tries to satisfy the request locally, at the same logical CPU where the request was made. If this is not possible, it picks from other logical CPUs in a greedy manner. It chooses a aligned extent from the logical CPU that has the most free aligned extents. It chooses an unaligned extent from the logical CPU with the most free unaligned extents.

Whenever an unaligned extent is deleted, the allocator tries to merge it with its nearby extents. If the extents can be merged into an aligned extent, they are; and the resulting aligned extent is tracked in the aligned extent pool.

Crash Consistency: Journaling. WineFS chooses to use journaling for updating file system metadata in an atomic fashion. There is a fundamental trade-off between using journaling and copy-on-write or log-structuring. Journaling results in writing metadata twice, once to the journal and once in-place. However, journaling preserves the data layout. In contrast, copy-on-write or log-structuring require only a single write to PM for metadata.

However, copy-on-write will write metadata throughout the partition, varying its location. The single write nature of log-structuring is attractive, which is why it has been adopted in NOVA and Strata. However, we believe that trading off an extra write for preserving the data layout is the right trade-off given how small metadata is, and how important hugepages are for performance. In this respect, WineFS makes a fundamentally different decision than NOVA and Strata.

Data Atomicity: Hybrid Techniques. WineFS provides atomic data updates by default. WineFS uses different techniques to update a file atomically, depending upon how the file extents are allocated. WineFS uses data journaling to update aligned extents, preserving their data layout. As a result, atomically updating a file will not cause it to lose its hugepages. WineFS uses copy-on-write to update unaligned extents, with the extents being written to new unaligned holes provided by the alignment-aware allocator. In this manner, WineFS strikes a balance between incurring the extra write for preserving data layout (when it matters), and using copy-on-write when preserving the data layout does not matter.

5.4.4 Ensuring good performance for applications using POSIX system calls

The design decisions described so far will preserve huge pages and help obtain good performance for applications using memory-mapped files. WineFS also seeks to obtain good performance for applications using POSIX system calls to access PM. The techniques WineFS uses to achieve this are well-known. The challenge lies in adopting techniques that compose well with the huge-page-aware design decisions, such that good performance is

obtained for applications using either memory-mapped files or system calls.

Fine-grained journaling. Similar to other PM file systems such as PMFS and SplitFS, WineFS optimizes journaling for PM. WineFS uses a per-CPU, fine-grained, undo journal. Each log entry is only a cache line in size. All metadata operations in WineFS are synchronous, so the journal entries are immediately persisted. Since metadata operations are synchronous, reclaiming journal space can be done immediately once the operation completes.

WineFS uses undo journaling instead of the redo journaling used in systems like ext4 DAX or SplitFS. In undo journaling, the old data is first copied to the journal, and then the new data is updated in place. If there is a crash, the data is rolled back to the old version using the journal. While undo journaling and redo journaling are functionally equivalent, their performance characteristics differ. Redo journaling has lower latency for writing transactions (no lock to write to the journal), but higher latency for updating data in place (need to get a global lock or set of locks). In contrast, undo journaling has higher latency for writing transactions (need to get locks to update data in place), but does not incur any delay once the transaction is committed; the log entries can be discarded. WineFS employs undo journaling as it reduces tail latency and provides more deterministic performance.

DRAM indexes. WineFS uses red-black trees for traversing directory entries and for maintaining inode free-lists in the per-CPU allocation group, similar to NOVA. The DRAM indexes help in fast metadata operations, as opposed to PMFS that does sequential scanning of directory entries and inode free-lists, causing significant slowdowns.

5.5 WineFS Implementation

WineFS is implemented based on the PMFS code base (6K LOC), and implemented in Linux kernel 5.1. We choose PMFS to build on, as PMFS is a journaling file system and has the on-disk layout that helps WineFS achieve all its goals. It is totally 10K LOC. The following optimizations have been added to the PMFS codebase: (a) PM-optimized per-CPU journaling: 1K LOC, (b) alignment-aware allocator and hugepage handling on page faults: 1K LOC, (c) auxiliary metadata indexes: 700 LOC, (d) NUMA-awareness: 300 LOC, (e) Crash recovery: 1K LOC, and (f) hybrid data atomicity mechanism: 500 LOC. We describe some of the additional implementation details below.

Alignment-aware Allocation. The allocator uses two pools to help with allocation. One is a pool of free aligned extents, and the other is a pool of free unaligned extents. These pools are written to PM on unmount. On a crash, they are re-initialized by scanning the set of used inodes in the file system (similar to NOVA).

Aligned extent pool. WineFS maintains a linked list of free aligned extents in each logical CPU. On getting a hugepage-sized allocation request, WineFS removes an extent from the head of the linked list and uses the extent for satisfying the allocation request. Whenever a free aligned extent is deleted, it is added to the tail of the linked list of the corresponding logical CPU.

Unaligned extent pool. WineFS re-uses the implementation of red-black trees in the linux kernel to keep track of free unaligned extents in each logical CPU. The red-black tree is keyed based on block offsets of the free extents. WineFS uses a first-fit approach to allocate an unaligned extent

for small allocation requests. Whenever an unaligned extent is deleted, the allocator tries to merge it with its nearby extents. If the extents can be merged into an aligned extent, it is merged and tracked in the aligned extent pool.

Journaling. Each thread in WineFS starts a transaction in its per-CPU journal. Once a transaction is started at a per-CPU journal, it continues there even if the thread is migrated away. Each journal transaction contains transaction-entries 64B in size, along with a start and commit entry to mark the start and end of the transaction.

Transaction entries. Each transaction entry contains the following metadata persisted on PM:

- shared transaction ID: This is an atomic counter shared by the per-CPU journals, which increments on every transaction create. As a result, a transaction ID is unique across all the per-CPU journals.
- per-CPU wraparound-counter: Each per-CPU journal contains a wraparound counter, incremented every time that the journal is wrapped around.
- transaction entry type: This provides information about the type of log entry, it is either START, COMMIT or DATA. The START and COMMIT entries are used to mark the start and end of a journal transaction, while the DATA entry is used to store system-call specific entries.

Reclaiming journal space. All operations in WineFS are immediately durable, allowing WineFS to reclaim the space in per-CPU journals that is occupied by the committed transactions. Every journal transaction reserves the maximum number of log entries that it requires in the per-CPU journal

before starting the transaction. Across all system calls, the maximum number of log-entries required are 10, occupying 640 bytes in the journal. If there is not enough space in the journal, the thread waits till enough space is reclaimed before starting the journal transaction.

Handling concurrent updates to shared files. When multiple threads try to modify the same directory by creating files in a shared directory, the VFS locks the directory inode, and only one of the threads is allowed to proceed at any given time. This VFS locking for all the shared metadata updates ensures that there is only a single uncommitted transaction for any file/directory on a crash.

Handling thread migrations. WineFS creates a journal transaction in its respective per-CPU journal. If the OS scheduler migrates the thread to another CPU after creating a journal transaction, WineFS still ensures that the migrated thread uses the per-CPU journal in which the transaction was created, for the duration of the transaction.

Journal Recovery. During recovery, WineFS has to recover multiple per-CPU journals. Note that transaction IDs are global across the per-CPU journals. WineFS rolls back journal entries across per-CPU journals based on the transaction ID order. The per-CPU wraparound-counter helps WineFS in identifying the valid journal entries during recovery. WineFS rolls-back all the transactions that contain the START log entry but that don't have the COMMIT log entry. WineFS ignores all committed transactions.

Minimizing remote NUMA accesses Given that PM will be deployed on multiple NUMA nodes, it is important that the PM file system try to minimize remote NUMA accesses. WineFS uses a number of mechanisms

to try to reduce remote NUMA accesses.

The NUMA-awareness strategy of WineFS builds on the insight that remote writes are more expensive than remote reads [85, 35]. Thanks to temporal locality, if writes are routed to the local NUMA node, reads of the newly written data in the near future will also be local. We recognize that it is challenging to prevent all remote accesses, and thus focus on minimizing remote writes.

Determining the home NUMA node for a process. WineFS assigns a *home* NUMA node to each process when the process first creates or writes a file. The assigned home NUMA node is the NUMA node with most free space.

Writes. On each write, WineFS checks if the process is in its home NUMA node. If required, the process is migrated to its home NUMA node, and space is allocated from one of the per-thread allocation groups on that NUMA node. Further allocations and writes continue at the home NUMA node. If the home NUMA node runs out of free space, a new home is selected, and the process is migrated.

Reads. All reads to recently written data will be local since WineFS ensures the writes happen on the home NUMA node. Older reads will be remote; WineFS does not migrate the process to prevent this situation. Thread migrations are expensive, and the process may access data spread out over different NUMA nodes causing it to thrash if it is migrated too often. Instead, WineFS focuses on keeping writes local.

Child process. Children of a process inherit its home NUMA node, under the assumption that they will be accessing data written by the parent process.

Crash Recovery and unmount. On a clean unmount, the data structures maintained in DRAM (e.g., alignment-aware allocator's free list, inode free list) are serialized and stored on PM. On mount, these data structures are deserialized and reconstructed in memory. If there is a crash, WineFS is first recovered to a consistent state using the per-CPU journals as explained above.

Reactively rewriting a file. If WineFS finds on memory-mapping a file that it is fragmented, it adds it to a list to be rewritten. A background thread in WineFS later reads the file and rewrites it using big allocations. A journal transaction is used to atomically delete the old file and point the directory entry to the new file. This situation may arise if an application uses small allocations when writing to a file that will be later memory-mapped. Due to the small allocation requests, WineFS would have allocated unaligned holes to the file. Note that reactive rewriting of files is an extremely rare operation. Applications that use the memory-mapped interface usually perform occasional large allocations in order to avoid frequently trapping into the kernel.

Supporting extended attributes for preserving alignment of files. Once WineFS provides aligned extents to a file, it makes this information persistent by using a special extended attribute. This is useful if a file allocated using aligned extents is later copied over to another partition or file system by an external utility such as `rsync` and `cp`. Ideally, we would want the file to retain aligned extents after the move or copy. Many linux utilities such as `rsync` and `cp` will read and copy extended attributes associated with files. WineFS uses the extended attributes to communicate alignment information of files from one WineFS partition to another (on

the same or different servers) no matter how that file is transferred. For example, if an aligned file is transferred from a WineFS partition on server A to a WineFS parition on server B via `rsync`, the receiving partition will allocate aligned extents (and not holes) to the file by referring to its extended attributes, even though `rsync` typically copies data using small allocations. Moreover, WineFS also supports directory level extended attributes where all files directly within a directory (not its subdirectories) will inherit alignment information from the extended attributes of the parent directory.

5.6 Discussion

Proactive approach is required to maintain hugepages. WineFS shows that by designing the file system to be hugepage-aware, it is possible to preserve hugepages in the face of aging and high utilization. We believe this is the right approach, as it can be implemented at modest additional complexity without sacrificing performance for applications using system calls to access PM. In contrast, reactive approaches like defragmentation provide only temporary relief before the file system becomes fragmented again. The defragmentation utility would need to be run at high frequency to provide benefits equivalent to WineFS. As with any background maintenance task, defragmentation requires IO and steals device bandwidth from the foreground process. We ran an experiment where we read a fragmented 5GB file and rewrote it with aligned extents. In parallel, we also ran a foreground workload that performed `mmap`-ed reads on another file. We observed a slowdown of 25–40% when the defragmentation is going on.

Thoughts on adding hugepage-friendliness to existing file systems.

We initially tried to add hugepage preservation in ext4 DAX by changing the multi-block allocator to provide 2MB aligned extents for large allocations. To accelerate applications using the system-call access mode, we changed the journaling mechanism of ext4 DAX to perform fine-grained journaling instead of relying on the JBD2 journal. With our changes, managed to get hugepages reliably in a clean setup for memory-mapped files. However, the allocator spent a significant amount of time in searching for available aligned extents, degrading performance when aged, compared to the original ext4 DAX. With respect to applications using system calls to access PM, the performance increased due to fine-grained journaling, but still suffered overheads such as ensuring consistency between on-disk versions of DRAM indexes.

NOVA uses log-structured layout for its metadata, and contains a per-inode log in the form of a linked list. Although NOVA tries to allocate aligned extents to large files, it is incapable of preserving hugepages due to extensive free-space fragmentation, as shown in Figure 3.1. NOVA would need to employ frequent (and expensive) garbage collection to retain free-space contiguity and alignment, interfering with foreground application performance.

Our experience shows that hugepage-awareness is an over-arching concern and not something that can be easily added to an existing PM file system. When designing WineFS, we had to incorporate hugepage-awareness in multiple core components of the file system.

Supporting different hugepage sizes. The size of a huge-page is not fundamental to the design of WineFS. We used 2MB hugepages in this work since our test machine had only 2MB hugepages available. While

WineFS currently has one allocator for 2MB hugepages, it can have additional allocators for each hugepage size that can be supported. For example, since modern kernels and devices support 1GB hugepages, WineFS could have two alignment-aware allocators, one for each hugepage size and one hole-filling allocator.

Using different aging profiles. Throughout the paper, we use the Agrawal aging profile [66] to age all file systems. The Agrawal profile contains a mix of large (≥2MB) and small files (<2MB). We also experimented with other profiles, and saw that in some cases, the fragmentation of other file systems is significantly worse compared to the fragmentation seen by the Agrawal profile. For example, in another profile that mimics an HPC environment [86], we see that even with 50% utilization, only 28% of the free-space is aligned and unfragmented in ext4 DAX, while more than 90% of the free-space is aligned and unfragmented in WineFS. Depending upon how the file system is aged, the user might experience more severe performance degradation than what we show in this work.

5.7 WineFS Evaluation

We seek to answer the following questions:

- Does WineFS handle crashes and metadata updates correctly? (§5.7.2)
- What is the read / write throughput of WineFS in an aged setting? (§5.7.3)
- What is WineFS performance for applications accessing PM through memory-mapped files in an aged setting? (§5.7.4)

- What is WineFS performance for benchmarks and applications using system calls to access PM? (§5.7.5)
- Is WineFS scalable? (§5.7.6)

5.7.1 Experimental Setup

We use a two-socket machine with 28 cores, 112 threads, and 500GB of Intel Optane DC Persistent Memory module, with Fedora-30 and Linux 5.1 kernel. We use a single socket on this machine for our evaluation, since the other file systems that we compare against are not NUMA-aware. We evaluate the scalability of WineFS across NUMA nodes in §5.7.7. We compare WineFS with two groups of PM file systems. First, we compare WineFS in relaxed mode with other file systems providing metadata consistency: ext4 DAX [38], xfs DAX [39], PMFS [40], NOVA-relaxed [87], and SplitFS [55]. Second, we compare WineFS in the default (strict) mode with file systems providing both data and metadata consistency: NOVA and Strata [58].

FS aging setup. To reflect PM file systems in the real world, we use the Geriatrix [70] tool to age evaluated file systems. Agrawal et al. [66] is one of the widely cited profiles that is used to measure the performance of aged file systems. We use the Agrawal profile to represent all the file systems aged by 165TB of write activity in a 500GB partition caused by creation and deletion of files, with a mix of small (< 2MB) and large (>= 2MB) files. 56% of the total capacity is occupied by large files, while the rest is occupied by small files.

5.7.2 Crash Consistency & POSIX Compliance

Crash consistency. We use a modified form of the CrashMonkey framework [88] to test whether WineFS recovers correctly from crashes. We use the Automatic Crash Explorer (ACE) to generate workloads with system calls that modify file-system metadata. For each workload, we use CrashMonkey to generate crash states corresponding to all possible re-orderings of in-flight writes inside each system call. The number of in-flight writes inside each system call were low, so CrashMonkey was able to exhaustively test crash states. Finally, we check that WineFS always recovers to a consistent state. This exercise was useful in finding minor bugs in WineFS early in its development. Currently, WineFS passes all the CrashMonkey tests.

As WineFS uses per-CPU journaling, we also check if WineFS is crash-consistent for multi-threaded applications. Note that WineFS shares a single namespace across all its CPUs. It uses the VFS locks to ensure that only one journal transaction (across all CPUs) involves each file. As a result, after a crash, WineFS has at most one per-CPU journal with pending updates for a given file or directory. WineFS recovers multiple per-CPU journals by using the global transaction ID to order different journal entries.

Time to recover. On recovery, WineFS must reconstruct the DRAM data structures such as the alignment-aware free-space allocator and per-CPU inode inuse lists using relevant metadata on PM. WineFS scans the per-CPU inode tables in parallel. Note that the recovery time depends on the number of files, and not the total amount of data in the file system.

By inducing a crash in WineFS with 675GB of data, WineFS recov-

ered in 7.8s. In this experiment, there were 3.5M files in the partition that was recovered.

POSIX compliance. We use the Linux POSIX file system test suite [89] to test if WineFS meets POSIX standards. WineFS passes all the tests. This is important because it means applications will obtain the expected POSIX behavior from WineFS without the need for application modifications.

5.7.3 Read and Write Throughput

We analyze the throughput of WineFS with microbenchmarks capturing sequential/random read/write workloads. We age the file systems as described in the experimental setup and then run experiments.

Performance for memory-mapped access. We memory-map a 50GB file and use `memcpy()` to perform reads and writes in sequential and random order. WineFS has the highest throughput across all aged PM file systems: WineFS outperforms NOVA by 2.6× on sequential and random writes, and by 2.3× on sequential reads, and by 2.7× on random reads, as shown in the Figure 5.2(a). WineFS spends about 3% of the total time on handling page faults while NOVA spends 60% of the time handling faults. WineFS has comparable performance with the best performing PM file systems in a clean, unaged setting since all file systems are able to map files using hugepages.

Performance for system-call access. We start with an empty file and append data at 4KB granularity until it fills 50% of the free space. We perform reads and in-place writes at 4KB granularities in sequential and random order. Overall, WineFS has equal or better throughput compared

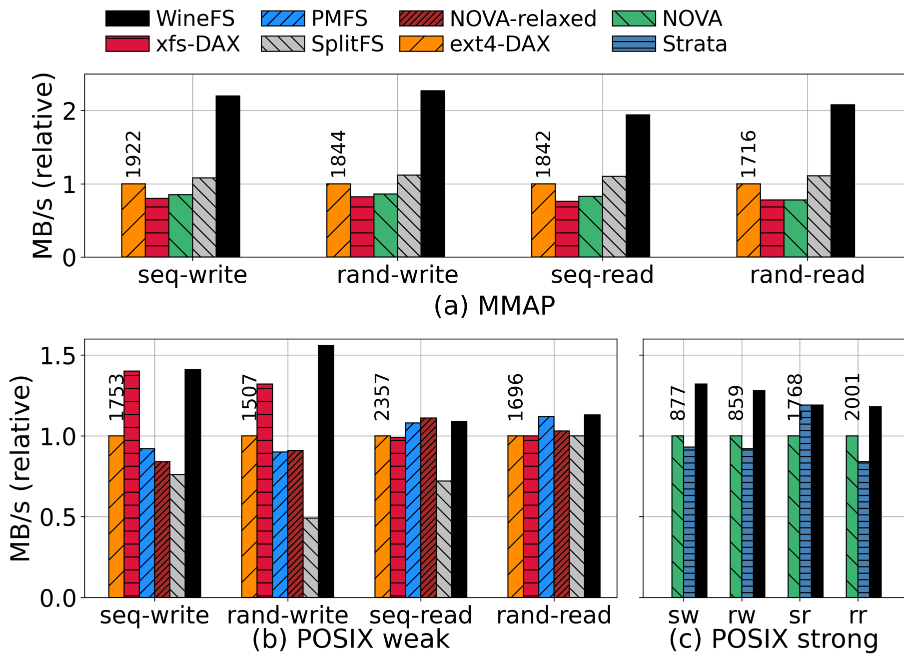

Figure 5.2: **Read and write throughput for system calls and memory-mapped access.** This figure shows the throughput of sequential and random reads and writes in different file systems. There is an `fsync()` after every 10 operations. POSIX strong indicates data consistency while POSIX weak indicates metadata consistency. SplitFS does not perform well on random writes and sequential reads in POSIX weak, because of a large number of memory mappings, and inefficient data structures that are used for indexing into file offsets while handling updates and reads compared to ext4 DAX. ext4 DAX suffers from slowdowns in random reads and performs similar to SplitFS. Across all these workloads, WineFS matches or betters the performance of the best PM file system. Good performance on memory-mapped workloads is due to huge-page-awareness, while good performance on system-call workloads is due to fine-grained journaling and DRAM indexing.

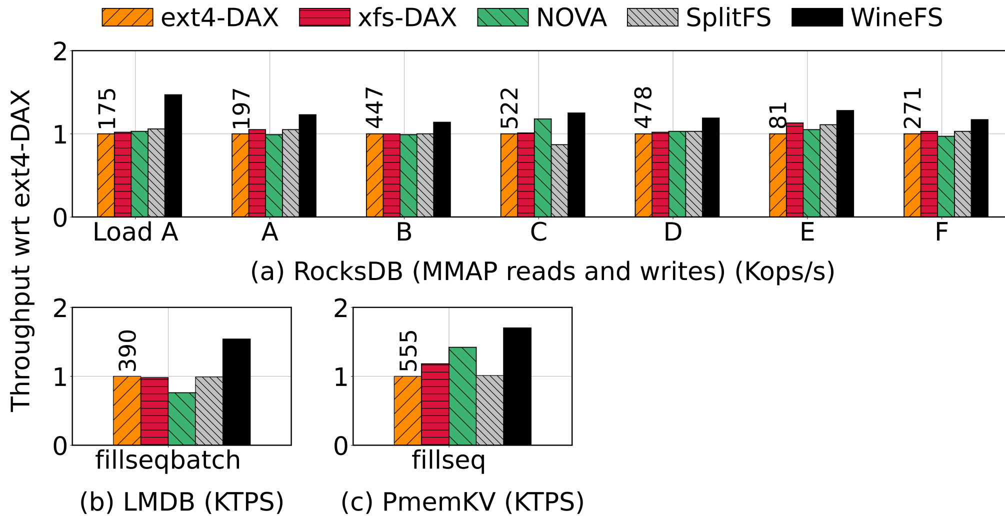

Figure 5.3: **Performance on aged file systems.** This figure shows the performance of different file systems when aged using the Geriatrix tool [70]. At the end of aging, the file systems are 75% full. Overall, WineFS outperforms all other file systems by up-to 70% compared to ext4 DAX in PmemKV and up-to 2× compared to NOVA on LMDB. The file-systems with metadata consistency guarantees are shown in (a), (b) and (c) whereas the file-systems with data as well as metadata consistency guarantees are shown in (d), (e), (f). Note that we do not compare with PMFS, as PMFS was not able to age successfully.

to other file systems on reads and writes, as shown in Figures 5.2(b) and 5.2(c). On writes, WineFS outperforms NOVA by up-to 25%, as NOVA has to add new log entries, invalidate older entries, and update its DRAM indexes for handling overwrites. Further, WineFS and NOVA perform better than Strata on writes as Strata has to perform expensive data copies from its per-process logs to the shared PM region for making data visible to other processes.

Summary. These results show that WineFS achieves excellent read and write throughput, regardless of whether memory-mapped files or system calls are used. It validates that the design choices made to increase memory-mapped and system-call access modes work well together.

	Load A	A	B	C	D	E	F
WineFS	**1.59**	**3.83**	**3.83**	**1.34**	**1.36**	**0.48**	**4.85**
ext4 DAX	11.85×	17.86×	6.20×	10.60×	18.36×	45.38×	14.57×
xfs DAX	28.26×	23.24×	7.04×	11.21×	20.27×	56.38×	17.70×
SplitFS	16.46×	20.52×	6.73×	10.93×	19.94×	50.58×	16.15×
NOVA	32.03×	1.57×	7.65×	1.05×	23.23×	1.15×	22.30×

(a) **YCSB Page Faults (In Millions).**

	fillseqbatch	fillseq
WineFS	**0.06**	**0.01**
ext4 DAX	205×	292×
xfs DAX	280×	455×
SplitFS	208×	296×
NOVA	261×	399×

(b) **LMDB and PmemKV Page Faults (In Millions).**

Table 5.1: **Page faults.** This table shows the number of page faults incurred by various applications on aged file systems. Overall WineFS suffers from the fewest page faults, up-to 450× lower than the other file systems. The software overheads of the applications, along with prefaulting of memory-mapped files in PmemKV limit the improvements for WineFS in terms of end-to-end application performance to 2× (Figure 5.3).

5.7.4 Performance for memory-mapped access mode

We evaluate WineFS using data stores like RocksDB, LMDB, and PmemKV. We continue to use the aged file system setting.

YCSB on RocksDB. We run RocksDB configured to use memory-mapped reads and writes, with hugepages enabled and a memory cap of 64GB. We run the industry-standard YCSB workloads on RocksDB with 60GB dataset consisting of 50M keys and operations. We report RocksDB throughput on all file systems in Figure 5.3(a). WineFS provides the best throughput, outperforming ext4 DAX and NOVA by up to 50% on average across all YCSB workloads. RocksDB incurs the fewest page faults

on WineFS. On other PM file systems, RocksDB incurs up-to 56× higher number of page faults.

LMDB. We run LMDB [59], a btree-based memory-mapped database, with db_bench benchmark's fillseqbatch workload with 50M keys. This workload batches and writes 1KB sized key-value pairs sequentially, which according to LMDB is its best-performing workload [90]. LMDB does on-demand allocations and zeroes pages on page faults by using `ftruncate()` instead of `fallocate()` for the allocations. This reduces space-amplification, but leads to costly page faults. WineFS outperforms ext4 DAX by 54% and NOVA by 2×, as shown in Figure 5.3(b). LMDB running on WineFS incurs 200× and 250× lower page faults in comparison to ext4 DAX and NOVA, as shown in the Table 5.1.

PMemKV. We run PMemKV [17], a key-value store from Intel that uses 128MB memory-mapped files for storing data on PM. We configure PMemKV's cmap concurrent engine to run with 16 threads. We run the write-only fillseq workload that sequentially inserts keys with 4KB-sized values. In this workload, PmemKV creates a PM pool using `fallocate()`, and keeps extending the pool as it gets used up by creating more files and allocating them via `fallocate()`. PMemKV gets the best performance on WineFS, which is 20% higher than on NOVA, 70% higher than on ext4 DAX, and 45% higher than xfs DAX, as shown in the Figure 5.3(c). NOVA does the allocations and zeroing of pages on `fallocate()` while the page fault routine only sets up page tables. On the other hand, ext4 DAX zeroes pages on a page fault and not on `fallocate()`, making page faults more expensive in ext4 DAX. As a result, the performance of NOVA is better than ext4 DAX; even though NOVA suffers from higher number of page

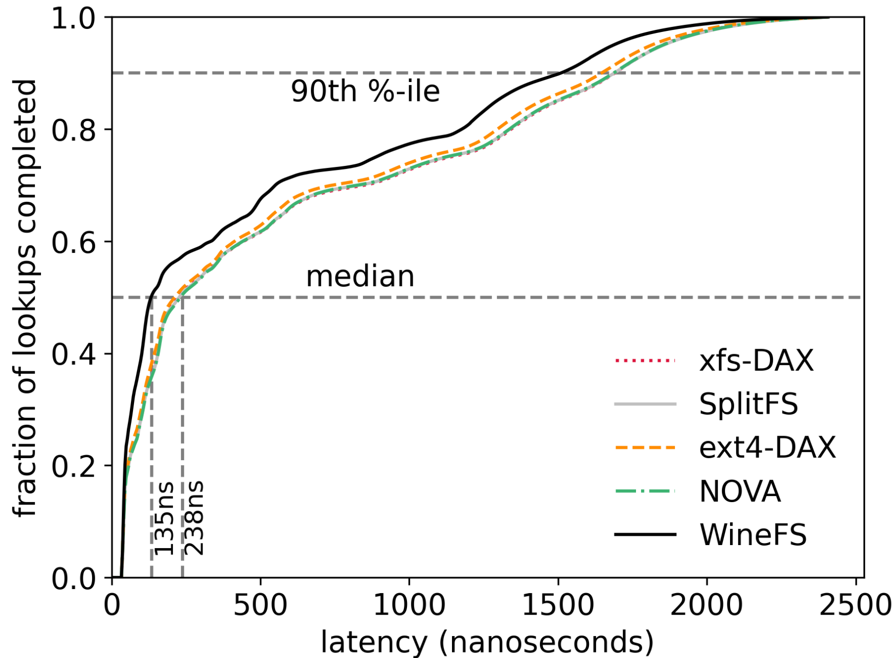

Figure 5.4: **Latency distribution for P-ART lookups.** The figure shows the latency distribution for lookups on the P-ART persistent radix tree. The tree is memory-mapped and pre-faulted before the lookups. WineFS has up-to 60% lower median latency compared to the other PM file systems as WineFS incurs fewer TLB misses and LLC cache misses.

faults compared to ext4 DAX.

Persistent radix tree. We study the performance of the persistent adaptive radix tree, P-ART [46], on WineFS. P-ART creates a PM pool using the vmmalloc library and pre-faults this region during initialization to avoid page faults in the critical path. We insert 60M keys to the index; page-table mappings are setup during inserts. We then perform 60M lookups of a hot-set of 125K unique keys in random order. The lookups don't suffer from page faults as page table entries are already setup. Figure 5.4 shows that the median latency of WineFS is **35%** lower than ext4-DAX and **60%** lower than NOVA. WineFS suffers 400× fewer LLC misses and 2× lower TLB misses compared to the next-best ext4-DAX.

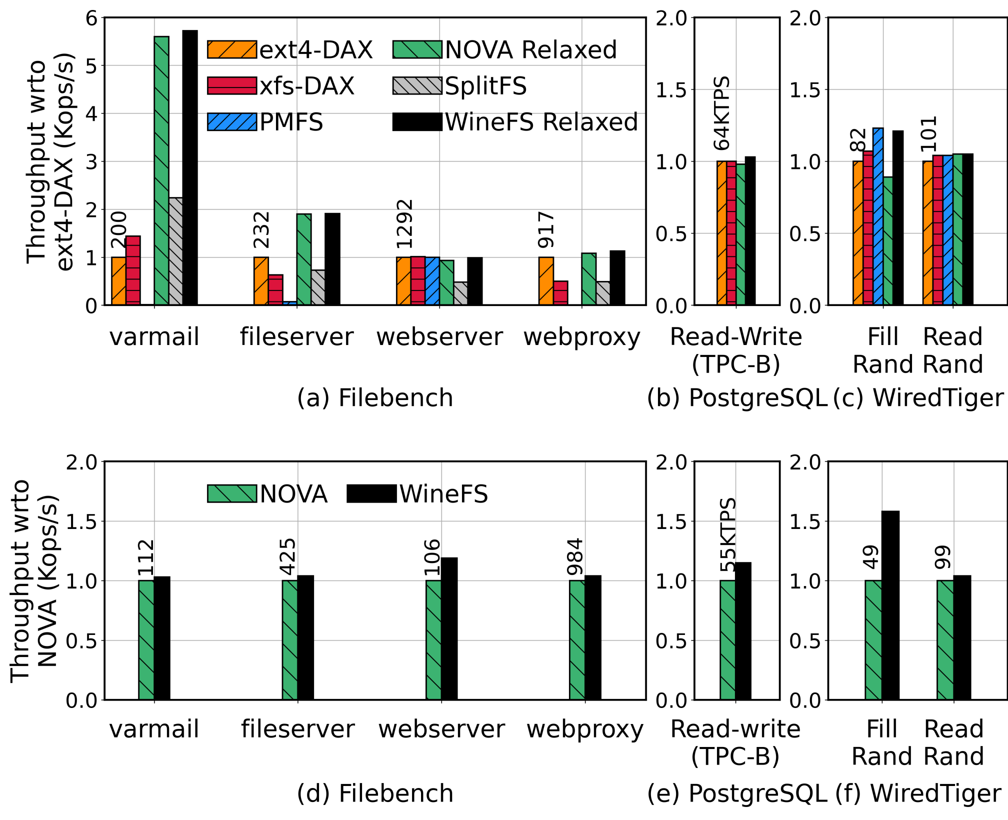

Figure 5.5: **Performance of applications using POSIX system calls on clean file systems.** WineFS has equal or better than the best file system in a clean file system setup. Performance of file systems in the relaxed mode (metadata consistency) is in (a), (b) and (c) and in the strict mode (data + metadata consistency) is in (d), (e) and (f).

Performance on newly created file system. We repeat all the above-mentioned experiments on newly created PM file systems. In general, PM file systems find it easier to map files with hugepages on a new file system. We find that all file systems perform similarly in a clean setup, with WineFS performing up-to 30% better compared to ext4 DAX and 35% compared to NOVA on YCSB Load A. WineFS outperforms PMFS by 80% in LMDB as PMFS does not get hugepages even in a clean file system setup. WineFS outperforms xfs-DAX by up-to 35% for reasons similar to PMFS. Across all these applications, WineFS gets the best performance or matches the performance of the best PM file system. This indicates that the design of WineFS is effective for memory-mapped files even on a newly created file system.

5.7.5 Performance for system-call access mode

We now evaluate WineFS on macro-benchmarks and applications that access PM via system calls. Aging does not impact system call performance on PM. We therefore use newly created file systems for these experiments.

Filebench. We use the Filebench [84, 91] macrobenchmark to evaluate WineFS. We use the varmail, fileserver, webserver, and webproxy benchmarks, with configurations as shown in Table 4.3. These benchmarks emulate the I/O behavior of several real-world applications.

WineFS and NOVA-Relaxed outperform ext4 DAX by up to **5**×. ext4 DAX and xfs perform poorly on varmail due to costly `fsync()` operations and metadata overheads. Although SplitFS outperforms ext4 DAX due to faster appends, it inherits low scalability for `creates` and `deletes` as it relies on ext4 DAX's JBD2 journal. The poor metadata structures, direc-

tory traversals, and inode free lists limit PMFS's performance on metadata-heavy workloads like varmail. WINEFS outperforms existing file systems on other Filebench macrobenchmarks, as shown in Figure 5.5 (a) and (d).

PostgreSQL. We use the PostgreSQL [24] database and run the read-write workload of `pgbench` suite (similar to TPC-B). WINEFS outperforms NOVA by **15%**, as shown in Figures 5.5 (b) and (e). The performance improvements trace back to overwrites. NOVA has to delete per-inode log entries, add new entries for handling overwrites, and update DRAM indexes to reflect the new data. WINEFS only modifies the inode in a journal transaction to point to the newly allocated blocks.

WiredTiger. We use WiredTiger [15], a key-value store that MongoDB uses by default, and run the `FillRandom` and `ReadRandom` workloads of the `db_bench` suite [81] with 1KB sized values. In FillRandom, WiredTiger on WINEFS performs **60%** faster than on NOVA, and outperforms ext4 DAX by **20%**, as shown in Figures 5.5 (c) and (f). WINEFS outperforms NOVA because WiredTiger appends data at unaligned offsets and NOVA forces these appends to a new 4KB page to ensure data atomicity, causing high write amplification. NOVA copies the data in the partial block to the new block and then appends new data. WINEFS continues to append to partially full blocks without having to copy old data like NOVA, while ensuring data atomicity via journaling. In `ReadRandom`, WiredTiger's throughput remains the same across different file systems.

Other utilities. We evaluate WINEFS using kernel compilation, tar, and rsync. Linux kernel compilation (v5.6; using 64 threads) takes similar time across all PM file systems. WINEFS has comparable performance as its competitors across all utilities.

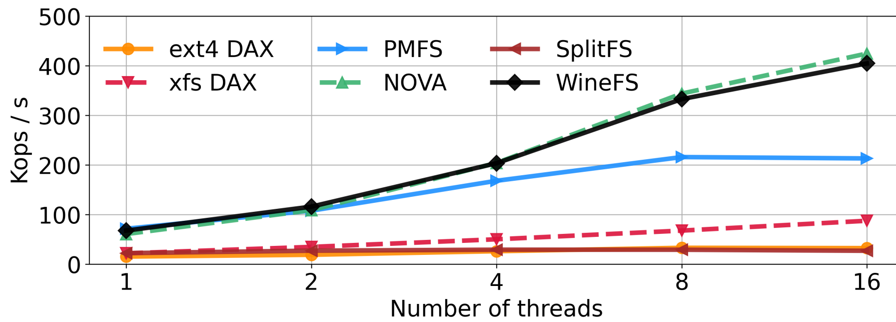

Figure 5.6: **Microbenchmark: Scalability.** WineFS throughput scales with increasing threads on metadata-heavy workloads.

5.7.6 Scalability

We measure the scalability of WineFS using a multi-threaded system call workload: we create a file, append at 4KB granularities, fsync, and unlink in each thread. Figure 5.6 shows the results. WineFS and NOVA have the best scalability. NOVA achieves its scalability through per-inode logs which have the side effect of fragmenting free space (and reducing performance for memory-mapped files). WineFS achieves similar scalability by using per-CPU fine-grained journals that minimize the fragmentation. ext4 DAX and xfs DAX have low scalability as they use a stop-the-world approach on `fsync()` to flush the journal to PM. SplitFS inherits low scalability as it runs atop ext4 DAX. Finally, PMFS scales well due to its fine-grained journaling. All the file systems plateau beyond 16 thread due to the scalability bottlenecks in the VFS layer.

5.7.7 NUMA-awareness

We run a microbechmark using 32 threads to append 100GB of data to thread-private files in different directories. Table 5.2 shows the percentage of remote NUMA writes incurred in different file systems. As each NUMA node has 500GB of free-space, file systems should be able to ensure com-

File system	*Remote Writes*	*File system*	*Remote Writes*
ext4-DAX	45.62%	NOVA	59.17%
xfs-DAX	27%	**Strata**	**0%**
PMFS	43.23%	SplitFS	50.21%
WineFS	**0%**		

Table 5.2: **Remote NUMA accesses.** Percentage of remote NUMA writes by different file systems. Only NUMA-aware WineFS and Strata perform no remote writes.

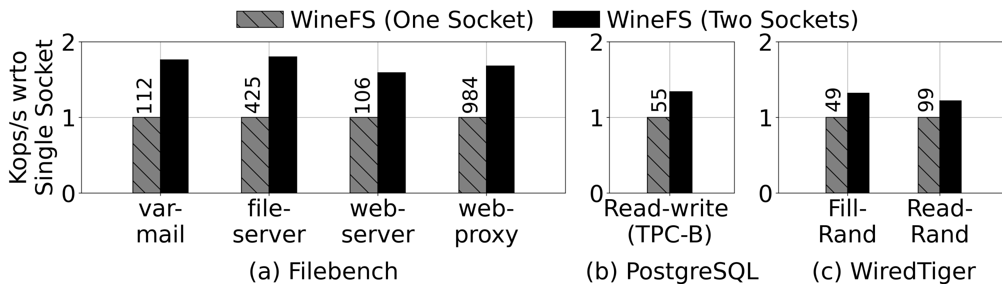

Figure 5.7: **Performance across NUMA nodes.** The performance of WineFS improves by up to 80% on write-heavy workloads when run on two NUMA nodes, due to its NUMA-aware thread migrations.

plete NUMA locality in the best case. WineFS incurs no remote NUMA writes due to its NUMA-aware allocation and thread migration policies. WineFS and Strata are the only file systems that are NUMA-aware. Other file systems incur 25%-60% remote NUMA writes.

We also measure the NUMA-awareness of WineFS on real applications that use POSIX system-calls to access data on PM. We create WineFS on a PM partition spanning across two NUMA nodes and evaluate the performance of WineFS on the workloads mentioned in §5.7.5. WineFS is able to perform up-to 80% better than a single NUMA node, as shown in Figure 5.7. The write-heavy workloads experience higher performance than read-heavy workloads, due to prioritizing writes over reads for local NUMA accesses, as described in §5.5.

5.7.8 Resource Consumption

WineFS consumes memory for its DRAM metadata indexes (e.g., red-black trees used for directory indexing, keeping track of free extents and inode free lists). It also additionally consumes CPU time to execute background activity such as journal space reclamation and retroactive rewriting of files.

Memory usage. WineFS uses a per-directory RB-tree to index the directory entries. The directory entries are hashed and stored, requiring less than 64B of memory per entry. Filling an entire 500GB partition of PM (used in our evaluation) with small 4KB files requires less than 10GB of DRAM for metadata. The memory usage of other DRAM metadata indexes such as the alignment-aware allocator and inode free lists is insignificant compared to the per-directory RB-tree, and can be safely assumed to be less than 1GB.

CPU utilization. WineFS uses a background thread to reclaim space occupied by committed transactions in the per-CPU journals, and uses another background thread in case of reactive re-writing of files to get hugepages. We expect the re-writing of files to be extremely rare, and in the common case, not utilizing a thread.

5.7.9 Summary

Overall, we show that WineFS achieves its goals (§5.3). It conserves hugepages and provides excellent performance for applications using memory-mapped files. It does not sacrifice performance for applications using system calls to access PM, performing equal to or better than the state-of-the-art PM file systems. It provides atomic, synchronous data and metadata

operations. It achieves these properties while being POSIX-compliant and not requiring any changes to the application.

5.8 Conclusion

This chapter presents WineFS, a hugepage-aware PM file system. WineFS demonstrates that it is possible to design a file system that achieves good performance for applications accessing PM via either memory-mapped files or system calls. WineFS revisits a number of file-system design choices in the light of hugepage-awareness. The design of WineFS allows it to resist aging, offering the same performance in the aged and unaged setting. WineFS is publicly available at `https://github.com/utsaslab/winefs`.

Chapter 6

Supporting and accelerating big data PM applications

In this chapter, we target the third class of applications called big data PM applications. These applications access large amounts of data using loads and stores on memory mapped files such that the data may not fit in the PM capacity of a single node. In Chapter 3, we saw that existing memory management systems and distributed file systems fail to provide DAX properties to big data PM applications, and compromise on the durability and consistency of data.

To enable big data applications on PM, this chapter introduces a new abstraction called *Distributed DAX memory mappings* (`ddmap()`) (§6.1). We then introduce ScaleMem, a system that implements the `ddmap()` abstraction. We first discuss the goals of ScaleMem and its target use cases (§6.3). We then present an overview of the design of ScaleMem and its mechanisms and policies for providing `ddmap()` (§6.4), along with the implementation of ScaleMem (§6.5). Finally, we evaluate ScaleMem with other systems that provide distributed memory mappings and show that ScaleMem outperforms the existing systems while providing stronger durability and consistency guarantees (§6.7).

System	**Distributed FS (e.g. NFS)**	**Far-Memory (e.g. Fastswap)**	**HotPot**
File-backed mapping	✓	✗	✓
Scalability	✓	✓	✗
Cacheline Durability	✗	✗	✓
Consistency	✗	✓	✓
Unmodified App Support	✓	✓	✗
Distributed File Mapping	✗	✗	✗

Table 6.1: **DAX Features.** The table shows the different features that are important for remote DAX memory mappings, and different systems that provide a subset of the features.

6.1 Need for a remote DAX `mmap()` abstraction

DAX offers important advantages for data intensive applications, with respect to performance by avoiding software overheads in the critical path, durability of data at cacheline granularity, and resource utilization, by avoiding extra copies of data in DRAM. Applications that are built for DAX rely on these characteristics of DAX and its durability guarantees. For example, applications such as PmemKV [17] and Pmem-Redis [45] use `clwb` for writing their data to PM at cacheline granularity from user space, rather than using expensive `msync()` that goes into the kernel. When run with large datasets, it is important that these applications get the same guarantees that they expect out of DAX memory mappings.

Existing systems that offer direct load/store access to large datasets are distributed file systems such as NFS [60], far memory systems such as Fastswap [92] and Infiniswap [93], and HotPot [94]. Table 6.1 shows the

list of features that are required for DAX along with different systems that offer remote memory mappings.

NFS allows unmodified applications at the clients to access memory mapped files in the server through page fault handling. However, NFS does not honor the characteristics of DAX, and copies data in the page cache of the client on page faults, causing data loss for certain applications such as PmemKV [17] that rely on the durability of data using `clwb` on DAX memory mappings. Fastswap and Infiniswap both modify the swap subsystem to enable access to remote DRAM instead of local disks, but are only restricted to heap-based applications. Applications that use memory-mapped files do not use the swap subsystem and cannot take advantage of Fastswap and Infiniswap. HotPot uses an intuitive API similar to local memory mapping for increasing the total available memory capacity to applications, but requires application modifications for issuing commit points. Furthermore, HotPot requires the entire dataset of applications to fit in the PM of a single server, which poses limitations on the application dataset size.

6.2 ScaleMem: Enabling Distributed DAX Memory-Mapping for Persistent Memory

As a part of this dissertation, we build ScaleMem, a system that implements the `ddmap()` abstraction. ScaleMem allows unmodified applications to access data using native loads and stores on memory mapped files, without worrying about the physical server on which their data lies. ScaleMem provides DAX memory mappings for distributed PM files, allowing applications to transparently scale in capacity to the available PM

across multiple nodes.

6.3 ScaleMem Goals

No application modifications. ScaleMem should be POSIX compliant, and support applications issuing POSIX system calls along with loads / stores on memory mapped files. ScaleMem should not require changing the application code.

Scalability across multiple nodes. ScaleMem should be able to support PM in multiple nodes and should allow applications to use large datasets spanning multiple nodes.

Strong consistency and durability guarantees. ScaleMem should allow applications to issue native loads and stores to their data on PM using DAX, and should allow cacheline-level durability of data using processor instructions like `clwb` and `sfence`. ScaleMem should provide immediate visibility and strong consistency guarantees for the data of applications that is similar to running the applications on a single node.

Flexibility. ScaleMem should use policies that provide high performance a diverse set of applications that access data in different patterns, without interfering with other concurrently running applications.

High PM utilization. ScaleMem should ensure high utilization of PM, and should allow free PM capacity in each node to be used for storing data of remote applications.

6.3.1 ScaleMem Setup and Usage Scenarios

ScaleMem is targeted towards applications that run on a single server,

but access large amounts of data that cannot fit in the PM capacity of one server. These applications are different from distributed applications where the compute is spread across multiple connected servers.

ScaleMem offers remote DAX memory mappings for such applications, and allows them to use loads and stores for the memory mappings. ScaleMem does not require any changes to the applications. While the applications that benefit the most out of ScaleMem are the applications that use memory mapped files, ScaleMem also supports applications that issue POSIX system calls to access their data. Supporting both modes of data access allows a broad range of applications to be supported with ScaleMem. Since ScaleMem is targeted at single-node applications, it does not provide properties such as fault tolerance for the data. Obtaining such properties is orthogonal to the goals of ScaleMem.

ScaleMem is designed for the common setup where multiple servers in a datacenter are connected to each other and use Remote Direct Memory Access (RDMA) for communication. We use the term "server" and "node" interchangeably. Each node has compute and memory, and uses PM for storing persistent data. Each node can run user applications independently of the other nodes, and can scale the application to use the PM of other connected nodes. This allows higher utilization of expensive PM across all the nodes. All the compute and DRAM for an application lies in the node that runs the application.

6.4 ScaleMem Design

In this section, we discuss the design of ScaleMem, starting with an overview of how ScaleMem manages to achieve the goals mentioned in

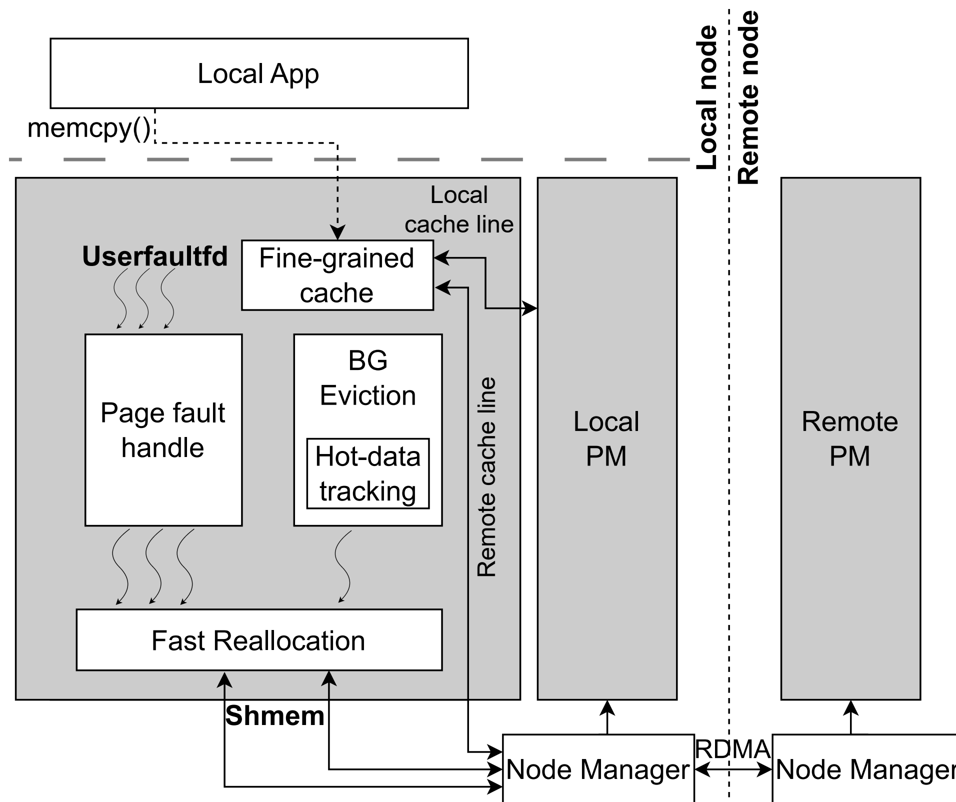

Figure 6.1: **ScaleMem overview.** This figure shows the main components of ScaleMem. ScaleMem uses fine-grained caching, and page fault handling at the App Manager to intercept application data accesses, and services remote page faults through RDMA at the Node Manager. ScaleMem uses background eviction for evicting cold data and reclaiming space in the local node. Fast reallocation is used to allow parallel page fault handling.

§6.3. We then discuss about how ScaleMem handles DAX memory mappings along with loads and stores. Finally, we discuss how ScaleMem handles POSIX system calls such as `read()` and `write()`, supporting a broad range of applications.

6.4.1 Overview

ScaleMem is a new system for supporting distributed DAX-based memory mappings. On a high level, ScaleMem services page faults of DAX-based memory mappings by migrating data to the application server, making the data part of the local file system in the application server. ScaleMem migrates cold data from the application server to a remote server when the local PM free space reaches below a particular threshold.

ScaleMem consists of two components. The App Manager handles memory mappings and DAX page faults, while the Node Manager handles file management and migration of data between nodes. Figure 6.1 shows the overview of ScaleMem. We now discuss the high-level ideas in ScaleMem that motivate its design.

Handling page faults in userspace. ScaleMem completely lies in user space. ScaleMem dynamically links to applications and intercepts system calls made by the application related to the memory management and file system. Being in userspace allows ScaleMem to employ custom policies for managing data specific to the applications it is linked to. ScaleMem registers file-backed memory mappings with `userfaultfd` [95]; the Linux kernel then forwards page fault events that are handled by dedicated threads.

Unifying the memory manager and file system layers. ScaleMem

reduces the bottlenecks that arise because of the lack of co-ordination of the memory manager and file system layers when it comes to remote DAX memory mappings. ScaleMem introduces a new primitive called *fast reallocation* in the kernel which allows in efficient migration of data between nodes while serving page faults.

Avoiding duplicate copies of dirty data across nodes. ScaleMem runs unmodified applications that are designed for single nodes, to use the PM of other nodes for large datasets. The applications expect strong consistency and global visibility of their data across different threads. ScaleMem achieves this by avoiding maintaining duplicate copies of data in different nodes. Every remote DAX page fault *moves* a 4KB block of file from a remote node to the local node. This has an additional benefit of high utilization of PM, and applications can scale up in capacity to the aggregate PM of all the connected nodes.

Tracking hot data at cacheline granularity. ScaleMem caches hot *read-only* data in the local node for minimizing network I/O. However, tracking hot data at page granularities is wasteful and leads to over-estimation of the hot data for data intensive applications like key-value stores and graphs, causing inefficient use of local space. To avoid this problem, ScaleMem intercepts loads and stores in userspace, and caches hot read-only data at cacheline granularity. ScaleMem uses caching only for data that is private to a particular application, and marks the data as read-only in the remote node. This avoids any modifications to the data by applications running on the other nodes.

We will now discuss about the mechanisms in ScaleMem that help it to run *unmodified* applications that issue loads and stores on memory mapped

files, while transparently scaling their data set sizes to the PM capacity of multiple nodes (§6.4.2). We will then talk about how ScaleMem supports POSIX system calls, and handle applications such as RocksDB [13] that issue system calls as well as memory maps, while scaling in capacity to multiple nodes (§6.4.3). Finally, we will discuss about the different policies used in ScaleMem that enable high performance for these applications (§6.4.4).

6.4.2 ScaleMem Mechanisms

ScaleMem has to handle allocations and placement of file blocks, migration of blocks, as well as page faults to transparently support remote DAX memory mappings.

Allocation and Memory Mapping. Every file is assigned a home node, which is the node belonging to the application that created the file. ScaleMem intercepts file allocation calls such as `fallocate()`, `ftruncate()` or file appends in the App Manager, and uses a greedy approach to allocate as much space in the home node of the file as possible. If the entire allocation does not fit in the free space of the home node, the rest of the allocation is serviced by contacting other nodes in a round robin fashion.

Upon `mmap()`, ScaleMem maps the file into virtual memory and registers the memory range for userfaultfd. ScaleMem then tracks the mapping from virtual address to file offset for each page that it manages. This information is used for migrating file blocks between nodes. ScaleMem tries to keep small files in the local node, because small files such as lock files are often short lived and temporary. When the local PM space is low and falls below a certain threshold determined experimentally, ScaleMem

also evicts small files.

Migration. ScaleMem monitors the space utilization periodically using a background thread in each node. As the free space starts getting used up, ScaleMem starts to evict file blocks to remote nodes in the background. The App Manager uses application-specific policies for finding cold blocks to evict. Evicting a cold block involves destroying the memory mapping corresponding to the block by the App Manager, and punching a hole in the file at the corresponding offset by the Node Manager for reclaiming space. Similar to HeMem [96], migration happens periodically and is rate-limited to avoid conflicting with application threads.

Page Fault Handling. ScaleMem handles page faults using userfaultfd [95]. ScaleMem creates a userfaultfd file descriptor and issues ioctls on it to register managed memory with userfaultfd. When ScaleMem intercepts memory allocation calls, it registers the virtual address range with userfaultfd, allowing it to receive page and write-protection faults on this range. ScaleMem uses dedicated page fault handling threads in the App Manager to receive fault events from the kernel. These threads read the userfaultfd file descriptor for page fault events.

In the event of a page fault, the App Manager locates the node of file block. If the block is absent in the local node, the Node Manager is contacted to fetch the block from the remote node. The Node Manager contains metadata for local files for locating the node corresponding to every block. The Node Manager then migrates the block using two-sided RDMA from a remote node, and the remote node punches a hole in its place and thus reclaims space. On receiving the block from the remote node, the local Node Manager writes the block contents to the

corresponding offset in the file, and returns to the App Manager. The App Manager then calls `mmap()` with MAP_POPULATE for setting up page table entries for the block.

Fast Reallocation. ScaleMem introduces novel changes in the Linux kernel to reduce the bottlenecks associated with parallel page fault handling and file fragmentation when dealing with remote DAX page faults and evictions.

Supporting parallel page faults. The design of ScaleMem exposes bottlenecks in the memory manager because of frequent mappings and unmappings due to data migration. Every mapping and unmapping operation requires metadata management in the kernel, which results in obtaining exclusive coarse-grained locks for the entire address space of the process. Fig 3.3 shows the effect of bottlenecks associated with multiple threads creating and destroying mappings.

ScaleMem introduces a new system call in the kernel for low-cost efficient unmapping called `mmap_unpopulate()`. When `mmap_unpopulate()` is invoked, it refrains from modifying the data structure responsible for maintaining memory mapped regions of the process. Instead, it removes the underlying physical memory page mapped to the memory mapped virtual address, which is extremely efficient. This effectively marks the memory mapped page as "stale", and any subsequent access to the virtual address of the memory mapped page results in a page fault, which is efficiently handled by ScaleMem as explained above. The only disadvantage of `mmap_unpopulate()` in contrast with conventional unmapping is that the virtual address range being marked stale is not reclaimed by the process. Note that the process's virtual address space is 2^{48} bytes; practically, this

does not hamper the application in a meaningful way. An alternative is to prevent heavy-weight global locking within the kernel by using fine-grained locks, or lockfree approaches, either of which are extremely complicated in a kernel environment and are likely to make multiple core components of the kernel vulnerable to bugs and breaking changes. We highlight the effect of `mmap_unpopulate()` in §6.7.2.

Reusing holes for small allocations. Migration of cold blocks of files to remote nodes creates holes in the files, causing file and free-space fragmentation in the file system. Instead of merging the small holes to form large free extents, we change the underlying file system to keep holes in a separate queue. We then allocate from the queue when migrating blocks from remote node to the local node on remote page faults. This reduces the bloating of free-space trees in the file system, and requires constant time for addition and removal of free blocks. Note that reusing holes for small allocations has also been explored in WineFS [97].

6.4.3 ScaleMem Mechanisms for POSIX system calls

ScaleMem primarily supports applications that issue loads and stores on DAX memory mappings, by handling page faults in userspace. However, this is not enough. Applications often issue POSIX `read()` and `write()` in addition to loads and stores for accessing data. For example, LMDB [61] and RocksDB [13] use `read()` and `write()` for storing the metadata of the key-value store. For supporting a diverse range of applications, ScaleMem supports POSIX data and metadata calls such as `open()`, `read()`, `write()`, `unlink()` made by the application, and handles them from userspace.

ScaleMem passes all the metadata operations to the local file sys-

tem in each node and maintains metadata in userspace for the files that it manages. Data operations such as POSIX `read()` and `write()` are intercepted by App Manager, and data is fetched from the node containing the corresponding file blocks.

6.4.4 ScaleMem Policies

ScaleMem uses the App Manager to dynamically bind to applications in userspace, and uses application-specific policies to improve the performance of applications. ScaleMem supports different policies for finding hot pages, prefetching pages and find-grained caching of read-only data. ScaleMem has configurable parameters for these policies that can be tuned according to the application access patterns by cloud providers and users. The policies do not interfere with other concurrently running applications on the same node.

Tracking hot blocks. ScaleMem uses an approximate Least Recently Used (LRU) policy by default to evict cold blocks when the free space reaches below a particular threshold. The App Manager uses a background thread to keep track of hot and cold blocks by tracking their accesses. The App Manager uses access and dirty bits of the pages that have mapped the blocks and intercepts POSIX `read()` and `write()` calls, in order to keep a count of accesses made to each block. The hot and cold blocks in separate queues, which are updated periodically by scanning the blocks in each queue. During eviction, a separate migration thread in the App Manager dequeues from the cold block queue and migrates blocks to a remote node. ScaleMem uses *fast reallocation* to remove page table entries corresponding to blocks that are scheduled for eviction. ScaleMem's

estimates of hot and cold blocks are kept fresh by periodically cooling the blocks by halving their access counts.

ScaleMem is able to use different application-specific policies such as First-In-First-Out (FIFO) and Least Frequently Used (LFU) for evicting data, as specified by the users.

Fine-grained caching. For read-heavy workloads with skewed access patterns, ScaleMem uses an application-specific cache for caching read-only hot data in the PM of the local node. ScaleMem intercepts `read()`, `write()` calls as well as `memcpy()` calls made by the applications, and tracks frequently read data at cacheline granularity. This significantly reduces the number of remote page faults in read-heavy workloads, while not compromising on consistency. The applications start with a cache size of 1GB which adaptively increases or decreases in size based on the available free space and the utilization of the cache.

By default, ScaleMem uses the LFU policy for tracking hot cachelines, but it can be configured to use any other policy that is specific to applications' access patterns.

Prefetching. For applications that access data with predictable patterns, ScaleMem allows the option of prefetching blocks and avoiding remote accesses in the critical path. Prefetching can be done in a sequential or a striped manner, depending on the application. While prefetching is kept on by default, applications have the option for turning it off if they do not access data using predictable patterns.

Components	*Lines of Code*
App Manager	
`mmap()` + Page fault handling	2.5K
Background eviction	1K
Space Management	1K
Prefetching + Read-only cache	300
POSIX system calls	1.5K
Other	2K
Node Manager	
File management	2.5K
Data migration	1.5K
RDMA Communication	1K
Other	2K

Table 6.2: **Code breakdown of ScaleMem.** This table gives a breakdown of the code in ScaleMem, in different components in the App Manager and Node Manager.

6.5 ScaleMem Implementation

We implement ScaleMem as a userspace library (17K lines of C code) and a small patch to the Linux kernel (200 LOC) to add the `mmap_unpopulate()` system call, modify the page fault handling in userfaultfd and the FAST_REALLOC flag to `fallocate()` in WineFS [97]. ScaleMem supports 22 common POSIX system calls, such as `read()`, `write()`, `open()`, `unlink()`, etc; we found that supporting this set of calls is sufficient to support a wide variety of applications and microbenchmarks. Furthermore, we rely on the contributions made by HeMem [96] in enabling userfaultfd for DAX. Since this support is implemented in Linux 5.1, we use the same kernel for our implementation.

Intercepting POSIX system calls and glibc calls. ScaleMem uses the syscall_intercept [98] library to intercept POSIX system calls and ei-

ther serve from user-space or route them to the kernel after performing some book-keeping tasks. Furthermore, ScaleMem uses LD_PRELOAD to intercept glibc calls such as `memcpy()` in order to support fine-grained caching (§6.4.4).

Free Space Management. The entire space in a node is managed by the Node Manager service in that node. When starting an application, the App Manager contacts the Node Manager, and the Node Manager reserves space for the App Manager depending on other concurrently running applications on the node. The App Manager then constructs its own free list on top of the allocated space to manage the space according to its own policies. The App Manager uses a red black tree for managing its free space. The App Manager divides the free space into blocks, each of which as a unique ID and corresponds to an offset in the free space. ScaleMem maintains data structures for inodes in userspace, and associates inodes to blocks on allocations. Memory mappings of a block by different threads in an application are maintained in a linked list.

Handling shared files. ScaleMem uses leases to handle files shared by multiple processes. Whenever the Node Manager gets a request for a file from multiple applications, the Node Manager issues a lease in a round-robin fashion to each App Manager, and uses timeouts to avoid starvation.

ScaleMem allows only one node to establish DAX memory-mappings for a page at any given time. If some other node tries to map the same page, ScaleMem waits to get the lease, and then migrates the corresponding page to the new node.

6.6 Discussion

Transparent fine-grained tracking of data. Workloads of data intensive applications such as key-value stores and indexes are typically skewed, and consist of small objects [99], of hundreds of bytes. While designing ScaleMem, we observed that tracking hot data at such small granularities reduces the network I/O by orders of magnitude. However, transparently tracking hot data at granularities below page size is fundamentally limited, because the entire Linux kernel virtual memory subsystem operates on pages. For example, a page fault requires setting up a minimum mapping of 4KB, and tracking hot data using access and dirty bits also is limited to page granularities. Tracking at page granularities leads to over-estimation of the hot data, and results in inefficient utilization of local space.

In ScaleMem, we use LD_PRELOAD to intercept `memcpy()` calls made by the application to track hot data at cacheline granularities. This greatly benefits applications that issue `memcpy()` calls to access their data on PM. We discuss about the benefits of our read-only fine-grained tracking in §6.7.2.

Using one-sided RDMA verbs. ScaleMem uses two-sided RDMA verbs (SEND and RECV) for migrating data between nodes. This is because ScaleMem has to perform active work in local and remote nodes to service page faults as well as evictions. Page faults require punching holes in the remote node, while evictions require allocating blocks in the remote node. We design a mode of ScaleMem that uses one-sided RDMA verbs (READ and WRITE) for a disaggregated setup with large amounts of PM in a remote cluster without compute, called ScaleMem- Passive. In this

mode, ScaleMem never deallocates or allocates space in the remote cluster, and performs one-sided RDMA for reading and writing to the remote PM cluster. However, we stick to using two-sided RDMA verbs in ScaleMem by default, as one of the major goals of ScaleMem is to utilize PM efficiently and work with commonly available hardware.

Overheads of userfaultfd. ScaleMem is a userspace library, and uses userfaultfd to service page faults, allowing applications to transparently scale to the PM of multiple nodes. We find that userfaultfd contributes to a significant overhead in page fault handling, especially when the data is present in the local node, due to multiple context switches between userspace and the kernel for every page fault. Our technique of *fast reallocation* reduces the number of context switches required for userfaultfd by reusing "stale" pages, and not creating new mappings on page faults.

Supporting Hugepages with ScaleMem. Hugepages reduce the number of page faults and increase the TLB reach by 512×. However, in ScaleMem, we observe that while hugepages are very beneficial when the dataset fits in a single node, they increase the network I/O significantly on remote page faults. We instead find that using 4KB base pages with prefetching for sequential access patterns is more beneficial, and achieves the same purpose of reducing the number of page faults in the critical path.

Using ScaleMem with CXL. Use of RDMA is not fundamental to the design of ScaleMem. We use a distributed setup with RDMA in this work to make it accessible for use with commonly available hardware, and due to the lack of clarity with respect to CXL support for PM as a storage device. Using CXL instead of RDMA has the potential to unlock higher performance through the use of hugepages, NUMA affinity of data and

hardware-managed cache coherency.

6.7 ScaleMem Evaluation

We first breakdown the performance of ScaleMem using microbenchmarks that access data in different patterns. We then evaluate the performance of ScaleMem on a variety of applications, where the data does not fit in the PM of a single node.

In our evaluation, we seek to answer the following questions:

- What is the latency breakdown of DAX memory accesses in ScaleMem? (§6.7.1)
- What is the impact of different policies of ScaleMem, for different access patterns? (§6.7.3)
- Does ScaleMem scale in throughput with increasing number of threads? (§6.7.3)
- Does ScaleMem scale in capacity with increasing number of nodes? (§6.7.4)
- How does ScaleMem perform in real-world applications that use DAX memory mappings? How does it compare to other systems that offer similar semantics in terms of consistency guarantees and performance? (§6.7.5)
- Can ScaleMem support complex applications that use POSIX system calls along with DAX memory mappings? (§6.7.5)
- What is the CPU and memory utilization in ScaleMem? (§6.7.6)

NFS: the closest alternative to ScaleMem. While there is no system today that offers the semantics of ScaleMem in terms of DAX memory mappings for remote files, the closest alternative is NFS [60]. NFS offers remote memory mappings via copying the data in the page cache of the clients on remote page faults. Using page cache for serving remote page faults violates the properties of DAX, of writing directly to persistent media, and results in data loss when the applications do not issue `msync()` for their data. Furthermore, Use of unbounded amount of page cache leads to memory becoming full in the client nodes, and leads to significant slowdown of the clients in general.

Semantic differences between ScaleMem and HotPot. HotPot [94] is a distributed shared PM system that exposes a global address space across multiple nodes, and allows applications to access data in this global address space via memory mapping. Despite allowing addressability spanning multiple nodes, HotPot requires that the entire dataset of a process fits in a single node (even if it is fetched remotely). Furthermore, HotPot requires changing the application to use their API, whereas ScaleMem is entirely transparent to the application. Since all of our experiments require accessing data either entirely, or partially stored on a remote machine, and the data size is strictly larger than a single node's capacity, we cannot compare our setup with HotPot.

Experimental setup and competitor systems. Our PM setup is emulated using DRAM. Semantically, this setup acts as a stand-in replacement for any PM hardware. We purposefully choose not to run on Intel Optane DC Persistent Memory. This is because our design is not dependent on the idiosyncrasies of Optane, particularly the degree of parallelism it supports

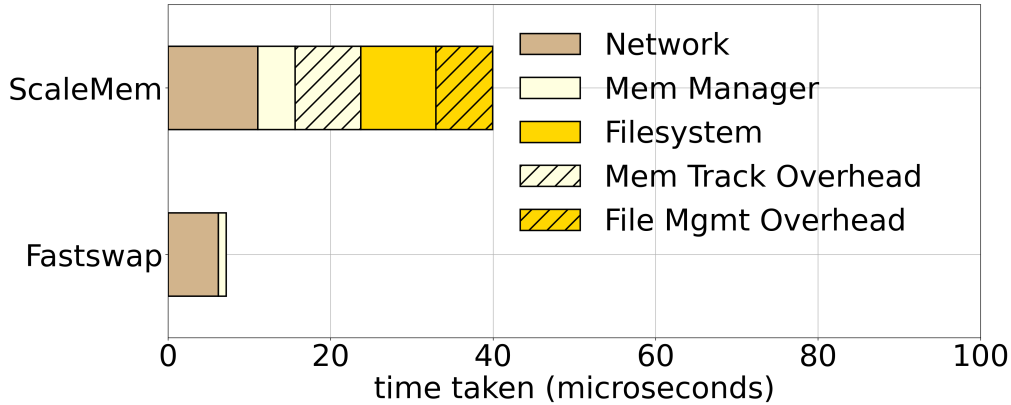

Figure 6.2: **Page Fault Latency Breakdown.** Shows the latency costs broken down by components in Fastswap and . 's cost is higher due to involvement of filesystems rather than just memory.

and its read / write asymmetry. We primarily focus our evaluation against the performance of NFS, since to the best of our knowledge, there is no other system that offers functionality similar to ScaleMem.

6.7.1 ScaleMem Remote page fault cost.

ScaleMem provides a unique abstraction of a distributed DAX memory mapping for applications whose datasets do not fit in the available PM capacity of a single node. In order to provide this abstraction without application modifications, ScaleMem has to build a unified management layer for file system as well as memory management components. This is because DAX memory mappings access data directly on PM, without copying it to DRAM, and hence also involve the file systems. In this section, we evaluate the different costs that ScaleMem has to incur when it comes to servicing a remote page fault on a memory-mapped file that uses DAX.

We run a simple experiment where we access a 4KB page in remotely stored file using `memcpy()`. A remote `memcpy()` invokes the ScaleMem page fault handling routine due to userfaultfd (as discussed in §6.4.2). Fig-

	Read Rand	Read Seq	Read Zipf
ScaleMem	**3.05 GB**	**3.05 GB**	**3.05 GB**
NFS	114.98×	7×	36.8×

Table 6.3: **Network IO.** This table shows the network I/O incurred by NFS and . Overall does much lesser network I/O.

ure 6.2 shows the total latency in servicing a remote page fault, along with a breakdown in different components of ScaleMem. In order to highlight the unique challenges faced by ScaleMem, we issue a 4KB remote `memcpy()` which leads to a page fault on the state-of-the-art far memory system called Fastswap [92]. Figure 6.2 shows the latency breakdown of ScaleMem vs Fastswap. While Fastswap is over 4× faster than ScaleMem, it is important to note that the challenges faced by ScaleMem to provide its unique semantics are not the same as those faced by far-memory systems such as Fastswap. In particular, Fastswap does not support memory mapped files; only supports heap-based (i.e. not file backed) allocations. By supporting neither DAX nor file backing (since it is a heap-based system), Fastswap does not involve the file system, and therefore manages to avoid a lot of complexity that comes with it. As a result, Fastswap cannot provide the strong durability guarantees provided by ScaleMem.

Since ScaleMem incurs high remote page fault cost, its design has several optimizations to lower this cost as is explained in §6.4.4. We now quantify the benefits of these techniques.

6.7.2 ScaleMem Techniques

First we evaluate the effect of the fine-grained caching and background prefetching techniques. Both these techniques result in fewer page faults

and network IO, which in turn improves overall application performance. We run a separate experiment for each technique in ScaleMem, to highlight its advantages.

Fast Reallocation. To measure Fast Reallocation, we sequentially read a 10GB file, half of which is present in the local node, and the local node capacity is set to 5GB (i.e. there is no more space in the local node). We read the file with increasing number of threads ranging from 1 through 8. We disable caching to highlight the effect of Fast Reallocation. We observe that in the absence of Fast Reallocation, the overall throughput only increases by **1.45**× as we increase from 1 to 8 threads. This is attributed to global locks captured by the kernel for the entire memory mapped region. Using Fast Reallocations, we are able to scale in throughput to as high as **4.7**× for 8 threads as shown in Fig 6.3.

Background prefetching. We run the same microbenchmark in which we sequentially `memcpy()` a remote 10GB file using 4KB requests. We measure the total time taken to read the entire file with and without prefetching. We also measure the number of remote page faults incurred by the microbenchmark in reading 10GB of data in each case as shown in Table 6.4.

We observe that prefetching significantly helps in reducing the end-to-end runtime. Prefetching reduces the number of remote page faults by over 97%, which in turn results in reducing the runtime by 2×. Despite almost entirely eliminating number of remote page faults, the relatively modest 2× improvement in runtime is attributed to stalls in the application thread that are waiting for the the prefetched block to be migrated from the remote node before it can be successfully memory mapped locally.

Fine-grained read caching. The effect of caching can be quantified us-

ing a realistic access pattern. Typically, file accesses are not done uniformly at random, but instead follow a heavy-tailed distribution. Thus, we run a microbenchmark in which we perform 200M 1KB `memcpy()` reads using a Zipfian access pattern over 10M values ($\alpha = 0.99$ which is used in YCSB [80]), with and without caching enabled. We measure the number of remote accesses in this experiment, and also report the end-to-end time taken for all 200M read operations. This result is also shown in Table 6.4.

Similar to prefetching, using a fine-grained read cache helps in reducing the runtime by 2.18×. This is because the cache is pre-created, prepopulated and fine-grained as explained in §6.4.4. The cache does not suffer from any file system overheads. Moreover, the fine-grained cache line level accesses captured by the cache allows reading lesser overall data over RDMA compared to a block or file level cache. We observed that data transfer was reduced by 70% compared to fetching 4KB-at-a-time from the remote node. Our cache uses the Least Frequently Used (LFU) algorithm, which is widely considered as the best caching policy for Zipfian workloads, and is able to get a hit rate of 90%.

To conclude, fast reallocation, prefetching and caching significantly reduces the overheads in ScaleMem. Both prefetching and caching and can be activated on a per-application-basis.

Next, we put together techniques evaluated above for a comparison of ScaleMem and NFS on commonly used microbenchmarks, and in §6.7.5 we compare the same systems for real-world application.

	Prefetching (ops/sec)	Caching (ops/sec)
Disabled	14k	257k
Enabled	**1.77**×	**2.1**×

Table 6.4: **ScaleMem Techniques.** This table shows the techniques employed by ScaleMem to reduce remote DAX page faults.

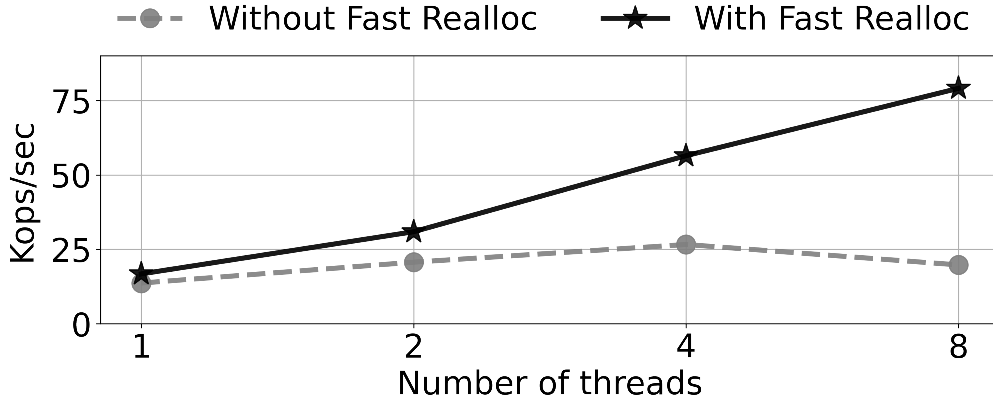

Figure 6.3: **Effect of Fast Reallocation.** ScaleMem, with the help of Fast Reallocation, is able to scale performance with threads.

6.7.3 Microbenchmarks

We analyze the throughput of ScaleMem by running microbenchmarks that capture sequential, uniform and Zipfian read/write workloads. We create a large file such that only half of it fits in a single node, memory map the entire file, and use `memcpy()` to perform reads and writes using different access patterns. We run this experiment using 4 threads. Each `memcpy()` command reads/writes 1KB of data in this experiment. We enable caching and prefetching depending on the access pattern, and report the throughput in terms of ops/s. We compare our performance with NFS, by restricting the DRAM in each node such that the DRAM can use the page cache to cache half the size of the file for a fair comparison.

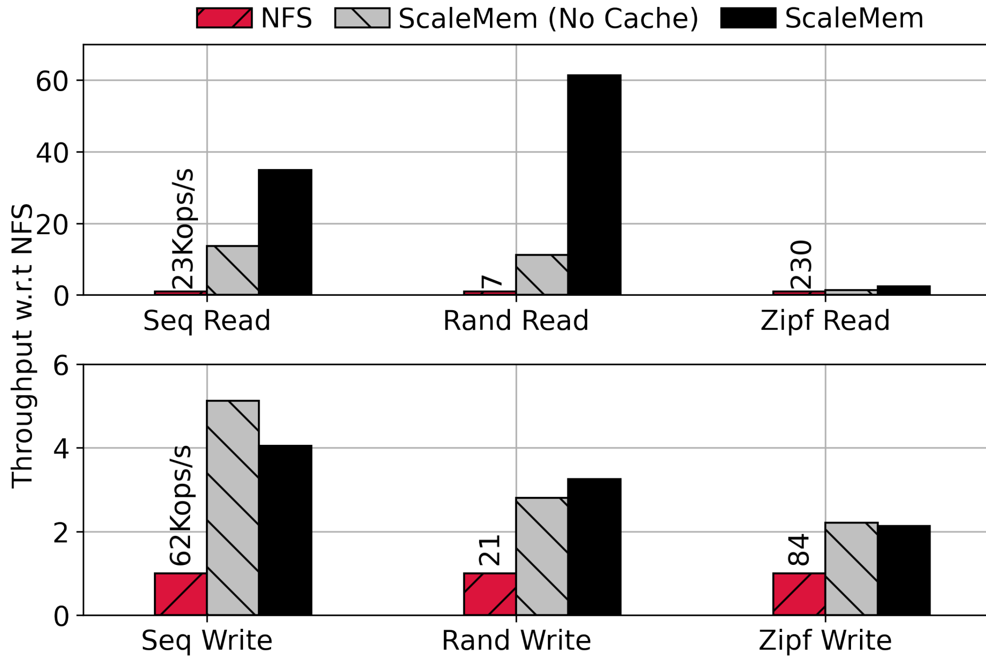

Figure 6.4: **Reads/Writes performance comparison with NFS.** NFS suffers from low performance in zipf and random reads as readahead in NFS leads to high network overhead and thrashing.

Sequential Accesses. For sequential accesses, ScaleMem without read caching performs similar to NFS, since both the systems suffer from the same number of page faults. However, note that ScaleMem provides DAX memory mappings and utilizes significantly lower memory compared to NFS which caches all the remote pages in the page cache of the local node. Furthermore, the durability guarantees provided by ScaleMem is always the same as that for locally present data, even if it fetched as a result of remote page fault. Whereas, for NFS, when it does a remote page fault, the remote data is present exclusively in DRAM until an `msync()` is invoked making it potentially more vulnerable to data loss in the presence of client side crash or power loss. With read caching enabled, ScaleMem outperforms NFS by **2.4**× for reads as shown in Fig 6.4. Writes in Scale-

Mem are **1.3**× faster than NFS, since NFS has to perform more remote I/O to flush the dirty data to the remote node, while ScaleMem uses the local PM.

Uniform Random Accesses. Fine-grained caching and background prefetching work best when data access patterns are not uniformly random. Optimization techniques used by NFS such as sequential readahead on page fault also suffer from poor performance due to thrashing for this access pattern. Nevertheless, ScaleMem still outperforms NFS significantly. Specifically, the ability to turn off prefetching for this workload prevents unnecessary (and useless) data transfers over the network. Fig 6.4 shows that NFS requires almost 115× more I/O compared to ScaleMem, resulting in ScaleMem obtaining over **10**× speedup, mainly due to to caching.

Zipfian Accesses. ScaleMem is able to perform significantly better than NFS in non-uniform, and specifically Zipfian workloads. ScaleMem takes advantage of fine-grained caching, and manages to avoid remote page faults, thus transferring less data over the network. The total network I/O done by NFS is 36.8× higher than ScaleMem. This reflects in the performance improvement, and ScaleMem outperforms NFS by **34**× in this workload. Writes do not benefit from the read cache, but ScaleMem's flexibility allows disabling prefetching and background evictions for writes resulting in a **5**× speedup over NFS for writes as shown in Fig 6.4.

6.7.4 Scalability in capacity

We run an experiment where we check the throughput of ScaleMem as we keep increasing the dataset. We perform multiple experiments, first of which creates a file of size 25GB, second creates a 50GB file that fits equally

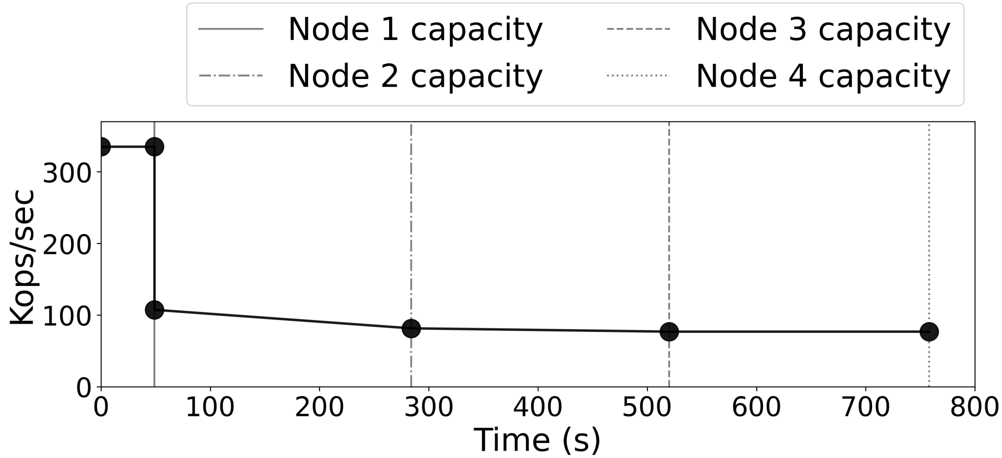

Figure 6.5: **ScaleMem scaling with nodes.** Shows sequential memcpy writes to a file that gradually increases in size expands across 4 nodes. Performance of ScaleMem scales with increasing number of nodes.

	RocksDB	LMDB	PmemKV
ScaleMem	**480.25 GB**	**305.94 GB**	**377.31 GB**
NFS	5.14×	11.14×	1.25×

Table 6.5: **Applications Network IO.** This table shows the network I/O incurred by NFS and ScaleMem. Overall ScaleMem does much lesser network I/O. They show the aggregate network traffic across Load A, Run A and Run C for YCSB and fillseq and randseq for DB Bench.

in 2 nodes, third creates a 75GB that fits in 3 nodes and lastly 100GB that fits in 4 nodes. We then perform `memcpy()` reads in Zipfian access pattern. ScaleMem is able to comfortably scale in capacity to 4 nodes. We report the throughput of ScaleMem as we increase the size of the dataset. We see that ScaleMem gracefully handles increased sizes in datasets, and is able to sustain large workloads. Fig 6.5 shows this result.

6.7.5 Real-world Applications

We evaluate the end-to-end performance of real-world applications that use ScaleMem to scale in PM capacity beyond the size of a single node.

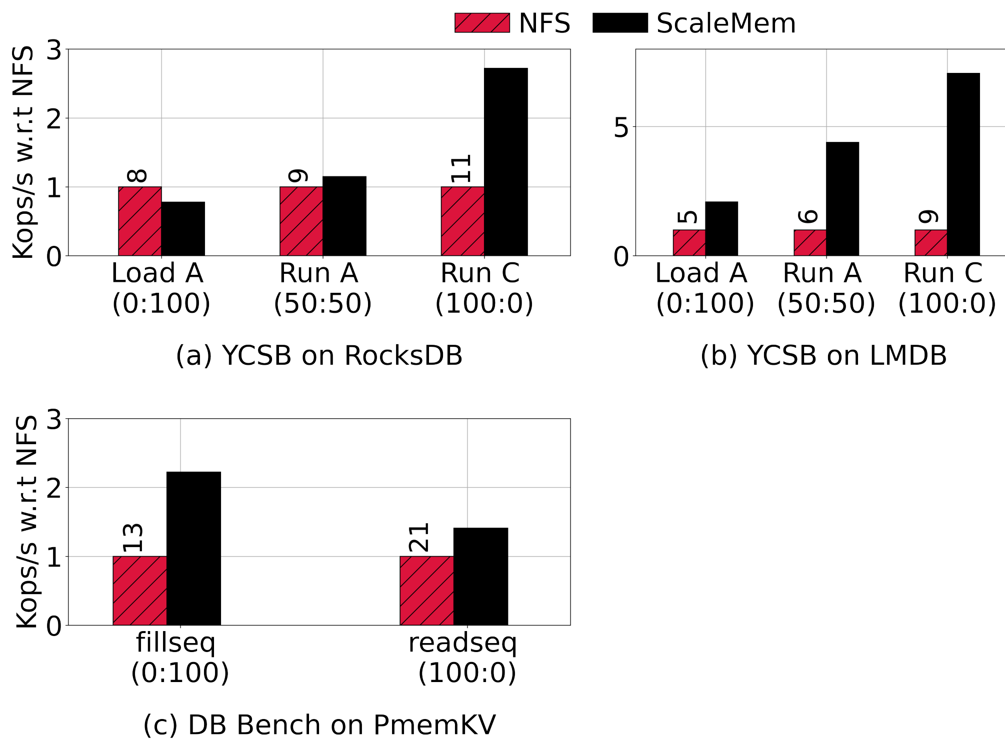

Figure 6.6: **Application Performance Comparison.** This Figure compares the performance of ScaleMem with NFS on 3 real-world applications: RocksDB (a), LMDB (b) and PmemKV (c). ScaleMem outperforms NFS on all the 3 applications due to fast page fault handling using Fast Reallocations and lesser network I/O due to its policies such as fine-grained caching and configurable prefetching.

We reiterate that only ScaleMem can run applications whose data spans across multiple nodes without any modifications, and provides the same consistency and durability guarantees to the applications as if all their data was present on a single node.

We evaluate ScaleMem using production key-value databased, viz. RocksDB [13], LMDB [61] and PmemKV [17].

YCSB on RocksDB. We run RocksDB configured to use memory-mapped file reads and writes, with a memory cap of 25GB. We run the industry-standard YCSB workloads on RocksDB with 50GB dataset consisting of 40M keys and operations. Each node is configured to 25GB of PM. We report RocksDB throughput on ScaleMem and NFS in Figure 6.6. ScaleMem outperforms NFS by up-to **3x** on read-heavy workloads, which have a zipfian distribution, due to its efficiency in utilizing memory and hot data tracking. ScaleMem suffers from 3× fewer page faults than NFS in Run C which is a read-only workload. ScaleMem is slightly slower than NFS on the write-only Load A workload, because it highlights the best case for NFS. NFS is able to read ahead large chunks of data in the page cache for this workload. Although ScaleMem also supports pre-fetching, the cost of a page fault due to DAX in ScaleMem is more than NFS , resulting in better performance for NFS in this workload. network I/O for ScaleMem and NFS is provided in Table 6.5.

YCSB on LMDB. We run LMDB [61], a btree-based memory mapped database, with the same YCSB workload suite, with a 100GB dataset. We report the throughput of LMDB on Load A which has 100% writes, Run A which performs 50% reads and 50% writes, and Run C which performs 100% reads. The value sizes are approximately 1KB in size. The

throughput numbers are reported in Fig 6.6. ScaleMem again manages to outperform NFS by up-to **7x** in the read-heavy workloads, due to reasons similar to RocksDB. Note that ScaleMem is not able to use the fine-grained cache in this workload because LMDB performs loads / stores on the data instead of `memcpy()`, and loads / stores cannot be intercepted via LD_PRELOAD. Thus, the improvements observed in LMDB are purely because of better tracking of hot pages in ScaleMem compared to NFS, and better utilization of memory. The network I/O for ScaleMem and NFS are provided in Table 6.5.

db_bench on PmemKV. We run PMemKV [17], a key-value store from Intel that uses a large memory-mapped file for storing data on PM. We configure PMemKV's cmap concurrent engine to run with 8 threads. We run the write-only fillseq workload that sequentially inserts keys with 4KB-sized values, followed by the readseq workload that reads the keys. PMemKV gets better performance on ScaleMem, which is **2.2x** higher than on NFS, as shown in the Figure 6.6. ScaleMem manages to use the fine-grained cache effectively in this workload, and suffers from significantly lower network I/O compared to NFS.

6.7.6 CPU and Memory utilization

ScaleMem consumes DRAM for its metadata management in the App Manager and Node Manager components (e.g. using red-black trees for managing free space, maintaining inodes and open files in structures in DRAM, etc). It additionally consumes CPU time to execute parallel page faults and for performing background migration of data.

Memory usage. ScaleMem keeps track of all the blocks in the local

node using an array of structures. Each page entry is 64B in size, and filling an entire partition of 50GB takes up less than 2GB of DRAM. The memory usage of other components such as free-space trees and inodes increases with the number of files, but is low enough to be assumed to be less than 1GB in size.

CPU utilization. ScaleMem uses a configurable number of threads for servicing page faults. The default is set to be 4 threads. Apart from this, ScaleMem performs background tracking and evictions of cold blocks using 2 threads. Furthermore, ScaleMem uses 4 dedicated threads for RDMA, and uses one more thread for the Node Manager service per node. Hence, ScaleMem has to use 11 additional threads in the default case for its background activity and RDMA management, apart from the application threads in the foreground.

6.8 Conclusion

This chapter presents ScaleMem, a system to support distributed DAX memory mapping, that allows *unmodified* applications to create DAX memory mappings for PM, regardless of whether it is local or remote in a cluster. ScaleMem uses *Fast Reallocation*, a novel technique to reduce bottlenecks associated with the memory manager and file system in supporting remote DAX memory mappings. ScaleMem demonstrates that it is possible to scale application datasets while not compromising on durability guarantees using remote DAX memory mappings.

Chapter 7

Discussion

In this chapter, we discuss about alternative solutions to our work in achieving the same goals mentioned in the dissertation and how our systems compare with each other. First, we discuss modifying existing file systems for achieving our goals as opposed to creating new file systems (§7.1). Next, we compare SplitFS and WineFS in terms of their design, and we discuss how they can be used to complement each other (§7.2). We then discuss using ScaleMem with WineFS (§7.3). Finally, we talk about the relevance of our contributions to other storage media (§7.4).

7.1 Modifying existing file systems

In this dissertation, we build two new file systems: SplitFS that targets POSIX system call applications, and WineFS, which targets applications that use memory-mapped files. An alternative way is to modify existing production file systems to achieve the same goals, instead of building new file systems from scratch, while leveraging the maturity of the file systems. In this section, we discuss our attempts in modifying ext4 DAX instead of building new file systems.

7.1.1 Leveraging ext4 DAX's maturity in SplitFS

One of the major goals of SplitFS is to accelerate data operations (`read()` and `write()`), while leveraging the maturity of ext4 DAX for metadata operations (`open()`, `close()`, `rename()`, etc).

Accelerating Data operations from user space. The most common operations in data-intensive applications are `read()` and `write()`. The common case `read()` and `write()` operations are straightforward to implement and test; SplitFS takes advantage of this fact and accelerates the data operations. Data operations are serviced from user space by memory mapping the file regions to the applications' address space and issuing loads and stores from user space, bypassing all software layers. The major gains in the performance of data operations come from avoiding expensive operations such as allocations and context switches in the critical path.

Modifying ext4 DAX to optimize for data operations would involve making changes to the allocation policy and journaling mechanisms, affecting the functionality of metadata operations as well. Moreover, this would not eliminate the overheads of context switching and the OS stack for every data operation.

Leveraging the maturity ext4 DAX for metadata operations. Metadata operations are comparatively rare in data intensive applications. Furthermore, POSIX has a number of corner cases to handle metadata operations. All the reported file system crash consistency and correctness bugs are for metadata operations. SplitFS relies on the maturity of ext4 DAX for these complex and rare operations. The operations are simply forwarded to ext4 DAX in the kernel from user space, and the result of the operation is returned to the applications.

In this way, SplitFS strikes a balance between high performance by accelerating common case data operations and maturity, by relying on the stable and mature ext4 DAX for the complex and rare metadata operations.

7.1.2 Modifying ext4 DAX instead of building WineFS

While designing WineFS, we tried to modify ext4 DAX to achieve hugepage friendliness for `mmap()` applications and high scalability for POSIX system-call applications, without compromising on its maturity. In particular, we tried to change the multi-block allocator as well as the journaling mechanism in ext4 DAX.

Changing the allocation policy of ext4 DAX. ext4 DAX contains an on-PM block bitmap which keeps track of the free-blocks. On top of the bitmap, ext4 DAX maintains an in-memory multi-block allocator to keep track of free extents. An allocation request first results in a scan of the multi-block allocator for the best-fit extent, and then the block bitmap, if the free-space is fragmented.

We changed the in-memory multi-block allocator take into account the physical addresses along with contiguity. The multi-block allocator was modified to allocate 2MB-sized aligned free-space regions for large allocations, getting hugepages for `mmap()`-based applications.

Changing block-based journaling to fine-grained journaling. ext4 DAX maintains a block-based JBD2 journal for crash consistency. The journal suffers from high write amplification, especially for metadata operations that require fine-grained updates. Furthermore, a call to `fsync()` requires flushing the entire journal to PM, and requires a stop-the-world approach, limiting the scalability of ext4 DAX. We changed the block-based

journal to perform fine-grained journaling in ext4 DAX.

Problems. Changing the allocator helped in reliably getting hugepages for `mmap()`-based applications on freshly-formatted ext4 DAX. However, when aged, the allocator became a significant bottleneck, performing multiple scans of the in-memory multi-block buddy allocator as well as the on-PM block bitmap for finding free space.

Changing the data structure of the multi-block allocator and removing the block bitmap would have required changes to the recovery mechanism in ext4 DAX, along with changes to the core data structures of the file system.

Tweaking the JBD-2 journal to perform fine-grained journaling helped in reducing the overhead of `fsync()`, but the scalability was still limited due to expensive locks held by the journal. Changing the journal altogether to remove the locks and to perform per-core journaling would have led to changes to the on-PM layout of ext4 DAX and the way it managed the free-space.

Summary. Making changes to the allocation policy and the journaling mechanism in ext4 DAX introduced necessary changes to multiple core components of the file system, defeating the purpose of using a production-level file system to leverage its maturity.

7.2 Comparing SplitFS and WineFS

SplitFS and WineFS are designed to satisfy different goals: SplitFS is designed to accelerate legacy POSIX system-call applications, while WineFS is designed to accelerate `mmap()`-based applications. Thus, they involve different design decisions. We compare SplitFS and WineFS, and discuss

whether SplitFS can be used with WineFS (instead of ext4 DAX) for its kernel component. We compare SplitFS and WineFS along two axes: design goals, performance.

7.2.1 Design Goals and Trade Offs

The different target applications for SplitFS and WineFS lead to different design goals as well as design trade-offs.

Design goals of SplitFS and WineFS. The main motivation behind designing SplitFS is to remove the software overheads from the critical path of data access, for legacy data-intensive applications. For this purpose, SplitFS was written in user-space to convert system calls into library calls underneath the hood with no context switch overheads for data operations. SplitFS relies on the kernel only for rare metadata operations.

The main motivation behind designing WineFS is to remove the software overheads for `mmap()`-based applications running on PM. These applications are meant to run on byte-addressable media, and issue loads and stores themselves for accessing data, without going through the file system. However, in the presence of DAX, the file systems still impact application performance due to placement of files on PM. WineFS uses a novel on-PM layout and allocation policy to consider physical alignment along with contiguity of free-space, while preserving the contiguity as the file system ages. Since WineFS has to manage physical layout of files on PM, WineFS lies in the kernel.

Design trade-offs in SplitFS and WineFS. Due to the different goals of SplitFS and WineFS, they make different trade-offs. SplitFS leverages the maturity of ext4 DAX for complex metadata operations, but also

inherits the inefficiencies of ext4 DAX in handling metadata operations. SplitFS does not optimize for `mmap()`-based applications, and, as a result, is not able to sustain high performance for such applications with age.

WineFS, on the other hand, suffers from frequent context switches for legacy data-intensive applications that issue POSIX system-call applications for accessing data. WineFS manages to achieve high scalability along with hugepage friendliness, and hence has to build the file system from scratch, without relying on the maturity of ext4 DAX.

7.2.2 Using SplitFS with WineFS

The different classes of applications targeted by SplitFS and WineFS raise an interesting question: *Can* SplitFS *be run on top of* WineFS *to get high performance for legacy POSIX system-call applications as well as `mmap()`-based applications?*.

While it is possible to achieve high performance for a broad range of applications by using WineFS as the kernel component for SplitFS, it is not without limitations. Firstly, one of the main goals of SplitFS is to rely on the maturity of a production file system, and to achieve high performance along with stability. On average, production file systems take a decade to mature and become stable [100, 101, 102, 103]. Since WineFS is a research prototype, it does not promise to provide the stability and maturity of ext4 DAX, which has been developed by thousands of developers for decades, and is actively being maintained even today.

Secondly, using SplitFS may result in fragmentation of free space, which contradicts with the goals of WineFS, and would compromise its

performance. Specifically, the `relink()` system call introduced in SplitFS moves 4KB blocks from one file to another, without moving data. This system call alters the on-PM layout of files, and breaks alignment of files. This causes file and free-space fragmentation, breaking hugepages. The straighforward solution for this is to perform `relink()` for 2MB extents instead of individual blocks, to maintain alignment and contiguity of free space. The downside of this approach is write amplification: Moving 2MB of data when 4KB is dirty will create a large number of journal entries, making the `relink()` operation expensive.

7.3 Using ScaleMem with WineFS vs ext4 DAX

ScaleMem is targeted towards `mmap()`-based applications that run on a single server, but access large amounts of data that cannot fit in the PM capacity of one server. Such applications typically create large files, memory map the files, and issue loads and stores from user-space.

Using ScaleMem with WineFS improves the performance of applications due to two reasons. First, WineFS helps obtain hugepages for such applications, and reduces the number of page faults. Second, WineFS uses efficient data structures compared to ext4 DAX for managing free-space and to allocate blocks. For applications with frequent allocations in the critical path (for e.g. PmemKV [17], LMDB [59], RocksDB [13], etc), fast allocations have a severe impact on the end-to-end performance. PmemKV when run with ScaleMem on top of WineFS provides 2× higher performance compared to ext4 DAX on the write-only fillseq workload due to its efficient allocation policy.

7.4 Applicability to other storage media

In this section, we discuss whether the contributions made in this dissertation can be applied to other storage media. We first discuss whether our techniques apply to block-based secondary storage devices such as HDDs and SSDs. Then we discuss about the application of our techniques to other byte-addressable media such as battery-backed DRAM or PCM.

7.4.1 Block-based storage devices

Block-based secondary storage devices such as HDDs and SSDs offer a different interface compared to byte addressable media. We think that the different contributions made in this paper, while well suited for byte-addressable storage, do not apply to block-based storage devices, due to the following reasons.

Software overheads do not dominate performance in POSIX system-call applications. HDDs have an access latency in the order of milliseconds, while SSDs have an access latency in the order of tens or hundreds of microseconds (Table 2.1). As opposed to this, the latency of a context switch during a system call is in the order of a few nanoseconds. While context switches become important for PM which is a low-latency storage medium, the bottlenecks are shifted to the latency of the hardware when it comes to HDDs or SSDs. As a result, the main contributions of SplitFS for eliminating software overheads in the OS stack would not lead to a significant improvement in overall performance for slower secondary storage media.

File systems do not affect the performance of `mmap()`-based appli-

cations. HDDs and SSDs do not allow direct loads and stores to storage from user-space. As a result of this, the `mmap()`-based applications cannot take advantage of DAX in order to map user-space addresses directly to storage regions. File systems on HDDs and SSDs do not need to worry about the placement of files on storage; data is copied from storage to DRAM on a `mmap()` call, and is handled entirely by the memory management system in Linux. Thus, the contributions made by WineFS in modifying the allocation policies to consider alignment along with contiguity of free-space do not apply to block-based secondary storage media.

Aging on HDDs and SSDs is different from PM. HDDs and SSDs benefit significantly from sequential access patterns, compared to random access. The problem of aging of file systems on HDDs and SSDs is related to contiguity of free-space, not the alignment of free-space. Furthermore, aging on these storage media impacts the performance of POSIX system-call based applications, and not applications that use memory-mapped files. The design decisions made in WineFS to consciously use holes for the POSIX system-call applications (§5.4.3) are contradictory to the properties of HDDs and SSDs, and would lead to significant performance degradations.

Secondary storage devices are not impacted due to NUMA. Secondary storage devices are attached to the I/O bus instead of the memory bus, and are not tied to NUMA nodes. Due to this, the file systems designed for HDDs and SSDs need not be NUMA-aware, and do not need to consider the physical address space and CPU affinities. The trade-offs required to build file systems on HDDs and SSDs are thus different from byte-addressable media.

7.4.2 Byte addressable persistent media

In this dissertation, we build and evaluate systems on top of Intel Optane DC Persistent Memory. However, there are different ways in which byte-addressable persistent memory is offered, such as Phase Change Memory (PCM) [104], Spin-Torque Transfer RAM (STT-RAM) [32], Battery-backed DRAM, and the emerging media such as memory-semantic SSDs [34] and CXL-attached memory expansion [105, 106, 107, 108]. In this section, we discuss the relevance and usefulness of the contributions and techniques of this dissertation in the context of other byte-addressable persistent media.

The contributions of this dissertation assume certain properties of underlying media, which are as follows:

1. **Low access latency.** Access latency should be similar to DRAM, in the order of nanoseconds. Our contributions eliminate the software overheads in the OS stack, where hardware is not the bottleneck.

2. **Support for DAX.** DAX should be supported for the underlying medium, since this dissertation analyses the implications of DAX on memory management and file systems, and proposes solutions.

3. **Fast random access.** Since this dissertation targets byte-addressable media, we assume that the random access performance of the medium is fast and similar to sequential access. In SplitFS and WineFS, we design the file system to take advantage of the fast random access by using up holes to sustain high performance in an aged setup.

We believe that these properties are common across all offerings of byte-addressable persistent media, and we argue that the contributions of this

dissertation are relevant and useful beyond the scope of Intel Optane DC Persistent Memory.

7.5 Summary

In this chapter, we discuss how SplitFS and WineFS compare to each other in terms of their goals, target use cases and design. We then discussed how they could be used together to accelerate legacy applications as well as modern PM applications. We then discussed about WineFS as the local file system for ScaleMem. Finally, we discussed about how this work is relevant generally to byte addressable media and not specific to Intel Optane DC Persistent Memory.

Chapter 8

Related Work

In this chapter, we discuss research and systems that are related to this dissertation. First, we discuss past efforts made in designing PM file systems (§8.1). Then we discuss past work done on hugepage-friendliness and NUMA affinity for PM systems (§8.2). Finally, we discuss distributed file systems, far memory systems and distributed shared memory systems in the context of big data `mmap()`-based applications (§8.3).

8.1 PM file systems

There has been a large body of work on PM file systems and building low-latency storage systems. We briefly describe the work done in PM file systems in this section.

Aerie. Aerie [65] was one of the first systems to advocate for accessing PM from user-space. Aerie proposed a split architecture similar to SplitFS, with a user-space library file system and a kernel component. Aerie used a user-space metadata server to hand out leases, and only used the kernel component for coarse-grained activities like allocation. In contrast, SplitFS does not use leases (instead making most operations immediately visible) and uses splitfs as its kernel component, passing all metadata operations to the kernel. Aerie proposed eliminating the POSIX interface,

and aimed to provide applications flexibility in interfaces. In contrast, SplitFS aims to efficiently support the POSIX interface.

Strata. The Strata [58] cross-device file system is similar to Aerie and SplitFS in many respects. There are two main differences from SplitFS. First, Strata writes all data to a process-private log, coalesces the data, and then writes it to a shared space. In contrast, only appends are private (and only until `fsync`) in SplitFS; all metadata operations and overwrites are immediately visible to all processes in SplitFS. SplitFS does not need to copy data between a private space and a shared space; it instead relinks data into the target file. Finally, since Strata is implemented entirely in user-space, the authors had to re-implement a lot of VFS functionality in their user-space library. SplitFS instead depends on the mature codebase of ext4 DAX for all metadata operations.

Quill and FLEX. Quill [109] and File Emulation with DAX (FLEX) [56] both share with SplitFS the core technique of transparently transforming read and overwrite POSIX calls into processor loads and stores. However, while Quill and FLEX do not provide strong semantics, SplitFS can provide applications with synchronous, atomic operations if required. SplitFS also differs in its handling of appends. Quill calls into the kernel for every operation, and FLEX optimizes appends by pre-allocating data beyond what the application asks for. In contrast, SplitFS elegantly handles this problem using staging files and the relink primitive. While Quill appends are slower than ext4 DAX, SplitFS appends are faster than ext4 DAX appends. At the time of writing this paper, FLEX has not been made open-source, so we could not evaluate it.

Native PM file systems. Several file systems such as SCMFS [64],

BPFS [63], and NOVA [37] have been developed specifically for PM. While each file system tries to reduce software overhead, they are unable to avoid the cost of trapping into the kernel. The relink primitive from SplitFS is similar to the short-circuit paging presented in BPFS. However, while short-circuit paging relies on an atomic 8-byte write, SplitFS relies on ext4's journaling mechanism to make relink atomic.

Kernel By-Pass. Several projects have advocated direct user-space access to networking [110], storage [111, 112, 113, 114], and other hardware features [115, 116, 117]. These projects typically follow the philosophy of separating the control path and data path, as in Exokernel [76] and Nemesis [118]. SplitFS follows this philosophy, but differs in the abstraction provided by the kernel component; SplitFS uses a PM file system as its kernel component to handle all metadata operations, instead of limiting it to lower-level decisions like allocation.

8.2 Hugepage-friendliness and file system aging

We now discuss prior research done in hugepage-friendliness in the context of PM file systems, for `mmap()`-based applications. We also discuss prior research done in reducing journaling overheads for POSIX system-call applications.

Hugepage-friendliness. Prior work studied the high cost of page faults in PM file systems and proposed changes to the memory sub-system [119]; in contrast, WineFS does not require any changes to the memory subsystem. Intel PMDK [53] recommends using ext4 DAX [120] or xfs DAX [39] with 2MB sized blocks, to ensure hugepage-friendliness. However, the downside of this approach is high space amplification for applications with files

smaller than 2MB.

NOVA [37] attempts to allocate hugepage-aligned physical extents, but requires allocation requests to be exact multiples of 2MB. The log-structured design of NOVA fragments free space; NOVA does not seek to prevent this. The free-space allocator in other PM file systems ignores fragmentation and physical alignment, causing a decrease in hugepages. WineFS is the first PM file system that has hugepage-friendliness as a primary design concern and shows that hugepages can be achieved without high space amplification.

Aging in PM file systems. Prior work has studied aging in file systems on magnetic hard drives [69] and SSDs [71, 72, 70]. While prior work has studied aging on emulated persistent memory [70], our work is the first to not only understand the problems that occur with aging on actual PM, but also address it via WineFS.

TLB effects. The correlation of increased performance due to larger TLB reach and coarser TLB mappings on PM was noted by prior work [121], but WineFS is the first to explain the reason behind these observations. Recent work [122] also speaks about the perils of TLB shootdowns, though not in the context of PM.

Fsync overhead. PM file systems like BPFS [63], PMFS [40], NOVA [37], Strata [58], and SplitFS [55] have reduced the overhead of `fsync()`. However, the log-structuring or copy-on-write design of NOVA, Strata, and SplitFS causes fragmentation and reduces hugepages. PMFS uses a single journal that becomes the bottleneck in multi-threaded applications. WineFS uses fine-grained per-CPU undo journal which minimizes `fsync()` overhead without trading off hugepages.

8.3 Distributed file and memory management

We now discuss past research on distributed file and memory systems for PM. We put this research in the context of ScaleMem. While ScaleMem builds on a wealth of research, it is the first system that provides the guarantees of remote DAX-based file memory mappings for applications. Furthermore, ScaleMem is the *only* system that can support applications that issue POSIX system calls as well as DAX-based memory mappings for files without compromising on consistency and durability.

Distributed File Systems Distributed file systems modify the file system component for allowing datasets to scale to multiple nodes. However, most distributed file systems do not manage the virtual memory subsystem to allow for memory mapping of distributed or remote files, and do not support applications that access data using memory-mapped files. Examples of Distributed file systems are Assise [85], Orion [123], Octopus [124], Lustre [125], Colossus/GFS [126] all of which do not offer memory mapping over distributed files.

Some file systems like NFS [60] and Ceph [127] allow for clients to use memory mapped files that lie in the server. These file systems offer memory mapped interface by loading the file blocks in the DRAM page cache of the client nodes. They rely on `msync()` in order to persist the dirty pages on to files on the remote servers. NFS and Ceph do not offer DAX memory-mappings, which directly map PM regions on to the user virtual addresses. This leads to data loss when DAX-based memory mapped applications such as PmemKV [17] which use instructions such as `clwb` and `sfence` to persist their data rather than `msync()` operations. Furthermore, NFS and Ceph end up using an unbounded memory for their page cache in the server and

the client for storing the memory mapped pages.

Far Memory Systems Far Memory systems [128, 129, 92, 93] allow heap-based applications to use the memory of remote machines using RDMA. AIFM [128] and FaRM [129] require application modifications to use specific APIs in order to scale to the memory of remote nodes. Remote regions [130] uses the memory-mapped API for accessing far memory, but require application modifications for data consistency. Fastswap [92] and Infiniswap [93] run unmodified heap-based applications and use the memory of remote nodes instead of local swap disks, by modifying the swap subsystem to perform RDMA to remote memory.

ScaleMem focuses on scaling applications that use file-backed `mmap()` instead of heap memory, especially when application data does not fit in the PM of a single node. This forces ScaleMem to deal with unique challenges in terms of unified management of file systems and memory management when using DAX that are not faced by far memory systems.

Distributed Shared Memory systems There has been a large body of work around distributed shared memory (DSM) [131, 132]. However, without RDMA, networks were a critical bottleneck in traditional DSM systems. Since the introduction of modern networks and protocols such as RDMA over infiniband, there has been renewed interest for DSM systems. Recent DSM systems include Grappa [133], GAM [134], Argo [135], etc. All the these systems require applications to use specific APIs in order to access the underlying DSM. For example, Grappa requires that applications use particular keywords and delegate functions for executing on the node that owns the data. GAM requires applications to use a subset of modified memory-management instructions such as and in order to allocate from

shared memory. Additionally, DSM systems, like far memory systems, are used for heap-based memory applications, and do not use file-backed memory mappings.

In contrast, ScaleMem does not require any modifications to the applications. ScaleMem is able to transparently scale the working set of applications, that use file-backed memory mappings, to the PM capacity of multiple nodes.

Distributed Shared Persistent Memory (HotPot) The closest system to ScaleMem is HotPot [94], which uses Distributed Shared Persistent Memory (DSPM) for increasing the total available memory capacity to applications. HotPot uses an intuitive API, similar to local memory-mapping, but still requires application modifications for issuing commit points. Furthermore, HotPot is focused on scaling out applications through replication and in terms of compute, but requires that the entire application dataset fits in the PM capacity of a single node.

ScaleMem, on the other hand, is focused on increasing the working set sizes of applications, and does not provide replication or fault tolerance. ScaleMem does not require any application modifications, and can use the unmodified `mmap()` interface to applications. Furthermore, ScaleMem supports POSIX system calls along with memory mappings, while HotPot focuses purely on memory mapped applications that do not issue any other system calls to access data.

Chapter 9

Future Work

In this chapter, we outline directions in which our work could be extended in the future. This fall into three main categories: reducing overheads of performing memory management in user-space, building tiered memory systems that leverage other memory and storage media and providing stronger guarantees for user data, such as availability and fault tolerance.

9.1 eBPF for memory management

We implemented the entire memory management layer in user-space in ScaleMem. This allowed ScaleMem to perform application-specific memory hot data tracking as well as prefetching. ScaleMem manages memory in user-space by registering memory to userfaultfd, and then writing a page fault handler routine in user-space for handling page faults. The main drawback of this approach is that page faults become expensive. Figure 9.1 shows a path of a page fault in a memory region registered with userfaultfd. As we can see in the figure, every page fault has to incur multiple context switches from user-space to kernel space.

eBPF is a method by which user-space code can be injected at arbitrary points in the Linux kernel, without requiring any recompilation. Using eBPF for handling page faults would allow user-space policies for tracking

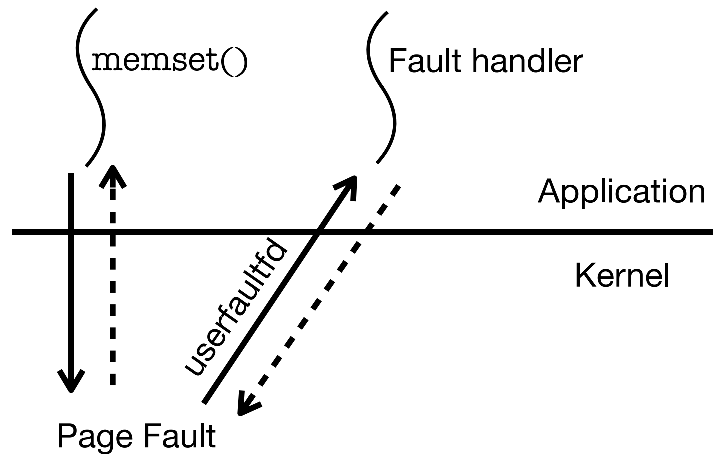

Figure 9.1: **Userfaultfd path.** This figure shows the path of a page fault when userfaultfd is enabled. Page faults serviced through userfaultfd suffer from high overheads due to multiple context switches between user space and kernel.

hot data, while reducing the number of context switches in the page fault routine. This would reduce the overheads of page faults.

ScaleMem suffers from a high number of page faults when the data does not fit in the PM capacity of a single server. Each eviction of a page results in removing its page table entries, causing a page fault on a subsequent access to the page. Reducing the overheads of page faults by using eBPF for memory management has the potential to have significant performance gains.

Handling page faults in the kernel using eBPF also aids in RDMA programming. Using RDMA in the kernel enables privileged features such as registering memory regions for DMA when the data arrives over the network. This saves CPU utilization during remote data handling, and increases the performance.

9.2 Leveraging other memory and storage technologies

The contributions made in this dissertation are in the context of PM used as the only storage device. While we believe that the adoption of byte-addressable persistent storage devices in the future is inevitable, we believe that other cheaper forms of block storage such as HDDs and SSDs will stay relevant. Additionally, there are newer memory technologies and interconnects that are being proposed, such as Compute eXpress Link (CXL) or byte-addressable SSDs, which can co-exist with other byte-addressable media. We believe that it is important to build tiered systems that can take advantage of the different characteristics of the different technologies in a single system.

In modern datacenters, memory is one of the costliest resources [136] and consumes the most amount of power. Furthermore, memory is heavily underutilized – it is always provisioned for the peak, which is bursty in nature. Adding memory and storage tiering to distributed memory management systems such as ScaleMem will enable us to use cheaper forms of memory for cold data, and buy lesser amounts of expensive DRAM, and ensure high utilization. This has the potential to save millions of dollars, along with reducing the carbon footprint of datacenters. SplitFS and WineFS can also be extended to include SSDs and HDDs as their lower tiers, similar to Strata [58], such that the hot data stays in PM while the cold data is compacted and stored in a sequential format on block-based secondary storage devices.

Adding multiple memory and storage technologies will make hot data tracking vital for performance. Depending on the performance character-

istics and the density of the technology, the data can be kept such that the hottest data lies in the best performing tier. We believe that this is an exciting area of exploration in the future, which has the potential to fundamentally impact memory management of datacenters.

9.3 Building systems that provide stronger guarantees

ScaleMem allows applications to extent their working data set sizes to multiple nodes, while providing the same load/store interface and guarantees of visibility of data, thus, not requiring applications to be modified in any way. However, ScaleMem suffers from unavailability of data due to server or network failures.

We think that it is important to enable distributed DAX-based memory management systems such as ScaleMem to provide reliability and availability of user data. Existing work in the context of far memory has been proposed by Google in the Carbink system [137]. ScaleMem can use techniques proposed by Carbink to enable replication of hot data, and erasure coding for the cold data, and provide high fault tolerance and availability of data, without compromising significantly on the performance.

9.4 Summary

In this chapter, we discussed how our dissertation work can be extended in the future. First, we discussed about using eBPF for managing memory, allowing use of DMA registration for RDMA along with per-application policies for tracking hot data. Next, we discussed about designing a tiered approach that uses multiple memory technologies for big data applications

along with hot data tracking according to the tiered memory characteristics. Finally, we discussed about providing stronger reliability guarantees for big data applications while providing DAX guarantees to applications.

Chapter 10

Lessons Learned and Conclusions

Today, data centric applications such as key-value stores, databases, graph processing systems rely on low-latency and high throughput access to data, along with strong guarantees for reliability of their data. This has led to newer storage devices such as PM, that offer fine-grained byte-addressable storage at nanosecond-scale latency. This shifts the bottlenecks from the hardware storage devices, to the software, that manages the storage devices. In this dissertation, we presented solutions that offer high performance along with strong consistency as well as durability for a wide range of applications running on PM.

We started by understanding the different ways in which applications access PM. We then analyzed how the existing OS stack and PM file systems incur software overheads that limit the performance of the applications running on PM. Finally, we observed how the current systems are incapable of running PM applications with large data sets that don't fit in a single server, while providing the same consistency and visibility guarantees.

In this dissertation, we presented our solutions to the above problems. We introduced SplitFS, a new PM file system that reduces the software overheads for legacy POSIX applications. Then we presented WineFS, a PM file system that is aimed at reducing software overheads for newer

PM applications. Finally, we introduced ScaleMem that enables PM applications to scale in terms of capacity to the PM of multiple servers, without compromising on guarantees.

In this chapter, we first summarize our solutions (§10.1). We then describe the lessons we have learned in the course of this dissertation work (§10.2). Finally, we conclude.

10.1 Summary

In this dissertation, we look at three different ways in which applications interact with PM, analyze the software overheads incurred by current systems, and present our solutions. We summarize each class of applications, along with our solutions.

10.1.1 Improving performance of legacy I/O intensive applications

The first part of this dissertation consists of accelerating legacy applications, that are designed for HDDs and SSDs, and are run on PM. We studied the way in which legacy I/O intensive applications interact with PM. We then analyzed the overheads of these applications, and our analysis led to the conclusion that PM file systems suffer from severe software overheads that slow down these applications. Specifically, we saw that the overheads came from common data operations such as `read()` and `write()` performing expensive operations such as allocations and context switches in the critical path.

We presented SplitFS, a new file system for PM that reduces software overheads significantly compared to state-of-the-art PM file systems.

SplitFS presents a novel split of responsibilities between a user-space library file system and an existing kernel PM file system. The user-space library file system handles data operations by intercepting POSIX calls, memory-mapping the underlying file, and serving reads and overwrites using processor loads and stores. Metadata operations are handled by the kernel file system ext4 DAX. By accelerating the data operations, SplitFS is able to accelerate legacy I/O intensive applications by up-to 2× compared to other PM file systems, while providing the same or stronger guarantees.

10.1.2 Improving performance of modern PM applications

The second part of this dissertation consists of improving the performance of modern applications that are designed for byte addressable media such as PM. We observed that new PM applications access PM in a different manner, compared to legacy I/O intensive applications. Specifically, we saw that the applications typically create one or a few large files, memory map the files, and access data using loads and stores from user-space, without entering the kernel for data accesses. We observed that the performance of the applications depends on the number of page faults incurred by the applications during run time. Huge pages help reduce page faults and improve performance. However, getting hugepages on PM requires that DAX file systems allocate file extents on aligned and contiguous hugepage boundaries. We saw that existing PM file systems were unable to reliably allocate file extents on hugepage boundaries, especially when aged, due to free-space fragmentation.

We built WineFS, a novel hugepage-aware PM file system that is aimed

at accelerating moderm PM applications that memory-map files. WineFS combines a new alignment-aware allocator that takes into account alignment of free-space, along with its contiguity, for allocating hugepage-aligned extents for the memory-mapped files. WineFS uses a suitable on-PM layout, and employs fragmentation-avoiding approaches to consistency and concurrency to preserve the ability to use hugepages with age. WineFS is able to reliably get hugepages, even when 90% of the partition is full and aged, and is able to outperform the other PM file systems by up-to 2× on memory-mapped applications.

10.1.3 Improving performance of big-data applications

The final part of this dissertation targets big-data applications, which are PM applications with large data sets that may not fit in the PM of a single server. In this age of data explosion, applications such as graph processing systems, ML training frameworks and key-value stores typically work with terabytes of data. Furthermore, these applications depend on DAX for global visibility and cacheline-level flushing of data. We observed that existing systems such as distributed file systems, far memory systems and distributed shared memory systems fail to honor the properties of DAX while supporting large data sets, which leads to data loss or requires application modifications.

We introduced a new abstraction named distributed DAX memory mappings (`ddmap()`), for PM which allows unmodified PM applications to create DAX memory mappings, regardless of whether the mappings are in the local or remote server. Furthermore, We built ScaleMem, which is a system that implements the `ddmap()` abstraction. ScaleMem codesigns the

file system and memory management layers to provide DAX guarantees to applications with large data sets. ScaleMem provides applications with the illusion of running on a server with a large amount of PM attached to it, and manages application data underneath the hood across multiple servers, without requiring any application modifications. By migrating hot data to local server and reducing software overheads in the file system and memory management layers, ScaleMem is able to outperform NFS by up to 7× on read-heavy workloads while providing stronger guarantees of data durability.

10.2 Lessons Learned

In this section, we present a list of general lessons we learned while working on this dissertation.

10.2.1 Transitioning from block addressability to byte addressability

While working on this dissertation, we realized that transitioning from block addressable storage to byte addressable storage requires revisiting fundamental designs in storage systems that are built for block devices. We discuss policies that are crucial in building high-performance systems for byte addressable storage.

Avoiding work in the critical path. In this dissertation, we extensively studied the software overheads of existing systems on a wide variety of applications. We observed that the root of the software overheads came from performing expensive operations in the critical path of common operations. For example, in SplitFS, we observed that current file systems

performed expensive allocations and metadata management on every data access. In WineFS, we observed that due to suboptimal placement of files, data access incurred page faults which required expensive context switches in the critical path. The contributions in this work include techniques that avoid work in the critical path. We pre-allocate wherever possible, and use a background thread to perform pre-allocation in the background. Similarly, we pre-fault memory mappings, and use a cache to re-use memory mappings as much as possible. We believe that this design principle will be useful for other systems designed for fast storage media.

Tracking fine-grained data accesses. Modern applications such as key-value stores, graph processing systems, or ML training frameworks typically access small keys and values, in the range of tens to hundreds of bytes. However, existing software systems including the Linux kernel tracks data access at the granularity of pages, which are 4KB in size. This results in multiple problems, such as overestimation of hot data causing inefficient caching, or increased write amplification in the case of crash consistency mechanisms such as journaling or copy-on-write. We think that there should be a fundamental change in tracking data, allowing fine-grained tracking of data in an efficient manner. This will greatly benefit applications that leverage modern byte-addressable memory and storage technologies in the future.

10.2.2 Designing application-specific policies

There are a diverse range of applications, each of which have different ways of accessing storage. Designing global policies for managing applications leads to sub-optimal performance for all the applications.

In SplitFS, we provide different crash consistency guarantees, ac-

cording to the durability and consistency requirements of the applications. The SplitFS user space library binds to applications, and provides the application-specific guarantees with the help of operation logging. In ScaleMem, we track the hot data of applications using different policies according to the way the applications access data, by binding to the applications. We believe that such application-tailored policies will become increasingly important with newer applications and hardware technologies, along with the demands for high performance.

10.2.3 Using DRAM effectively

In block addressable storage media, due to high latency of storage, DRAM is used as a write-back cache. However, with the introduction of storage class memory such as PM with access latency similar to DRAM, data directly reach PM without involving the page cache. However, we believe that DRAM still plays a crucial role in the presence of PM.

DRAM should be used to store temporary data, or data that can be reconstructed on a crash. For example, in the case of WineFS, we designed the entire free-space allocator in memory, using efficient in-memory data structures such as red-black tree. This avoids unnecessary wear-out of PM, as the allocator is accessed frequently, causing small random writes on the media. The allocator can be re-constructed on a crash, by scanning the PM partition, which can be done efficiently in parallel.

10.3 Closing Words

We are at a critical juncture of systems research, with newer hardware technologies coming up due to the end of Moore's law, along with high

demands for performance in applications that generate and consume increasing amount of data. This high demand for performance along with innovations in hardware have caused the bottlenecks to shift in the software stack. In this dissertation we present solutions that take into account the characteristics of newer hardware technologies. With newer memory and storage technologies coming up, this dissertation shows that by carefully designing software, it is possible to build systems that achieve both high performance as well as strong guarantees for applications, without placing the burden on application developers.

REFERENCES

[1] "Cloudscene: Data generation," https://explodingtopics.com/blog/data-generated-per-day.

[2] Amazon, "Amazon EC2 Reserved Instances Pricing," https://aws.amazon.com/ec2/pricing/reserved-instances/pricing/.

[3] ——, "Amazon AWS," https://aws.amazon.com/.

[4] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Bedekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan, and L. Rigas, "Windows azure storage: A highly available cloud storage service with strong consistency," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11, 2011.

[5] Google, "Google Cloud Platform," https://cloud.google.com/.

[6] ——, "Google Firebase," https://firebase.google.com/.

[7] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?" in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 591–600.

[8] Y. Zhang, C. Dragga, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Viewbox: integrating local file systems with cloud storage services." in *FAST*, vol. 14, 2014, pp. 119–132.

[9] A. Developers, "What is android," *Dosegljivo: http://www.academia. edu/download/30551848/andoid–tech. pdf*, 2011.

[10] A. Charland and B. Leroux, "Mobile application development: web vs. native," *Communications of the ACM*, vol. 54, no. 5, pp. 49–53, 2011.

[11] A. Holzer and J. Ondrus, "Trends in mobile application development," in *Mobile Wireless Middleware, Operating Systems, and Applications-Workshops: Mobilware 2009 Workshops, Berlin, Germany, April 2009, Revised Selected Papers 2*. Springer, 2009, pp. 55–64.

[12] S. Holla and M. M. Katti, "Android based mobile application development and its security," *International Journal of Computer Trends and Technology*, vol. 3, no. 3, pp. 486–490, 2012.

[13] Facebook, "RocksDB — A persistent key-value store," http://rocksdb.org, 2017.

[14] Google, "Leveldb," https://github.com/google/leveldb, 2019.

[15] A. Fedorova, C. Mustard, I. Beschastnikh, J. Rubin, A. Wong, S. Miucin, and L. Ye, "Performance comprehension at wiredtiger," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena*

Vista, FL, USA, November 04-09, 2018, 2018, pp. 83–94. [Online]. Available: https://doi.org/10.1145/3236024.3236081

[16] "Hyperleveldb performance benchmarks." http://hyperdex.org/performance/leveldb/.

[17] Intel, "PmemKV — Key/Value Datastore for Persistent Memory," https://github.com/pmem/pmemkv, 2018.

[18] P. Raju, R. Kadekodi, V. Chidambaram, and I. Abraham, "PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP '17)*, Shanghai, China, October 2017.

[19] B. Lepers, O. Balmau, K. Gupta, and W. Zwaenepoel, "Kvell: the design and implementation of a fast persistent key-value store," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 447–461.

[20] M. Seeger and S. Ultra-Large-Sites, "Key-value stores: a practical overview," *Computer Science and Media, Stuttgart*, 2009.

[21] F. Xia, D. Jiang, J. Xiong, and N. Sun, "Hikv: A hybrid index key-value store for dram-nvm memory systems." in *USENIX Annual Technical Conference*, 2017, pp. 349–362.

[22] B. Chandramouli, G. Prasaad, D. Kossmann, J. Levandoski, J. Hunter, and M. Barnett, "Faster: A concurrent key-value store with in-place updates," in *Proceedings of the 2018 International Conference on Management of Data*, 2018, pp. 275–290.

[23] "Redis: In-memory data structure store," https://redis.io, 2019.

[24] PostgreSQL, "PostgreSQL: The World's Most Advanced Open Source Relational Database," https://www.postgresql.org/, 1996.

[25] M. PostgreSQL, "Announcing Memhive PostgreSQL," https://www.postgresql.org/about/news/announcing-memhive-postgresql-2088/, 2020.

[26] MongoDB, "MongoDB," https://www.mongodb.com, 2017.

[27] "MySQL," https://www.mysql.com.

[28] R. Greenwald, R. Stackowiak, and J. Stern, *Oracle essentials: Oracle database 12c*. " O'Reilly Media, Inc.", 2013.

[29] A. Pavlo, G. Angulo, J. Arulraj, H. Lin, J. Lin, L. Ma, P. Menon, T. C. Mowry, M. Perron, I. Quah *et al.*, "Self-driving database management systems." in *CIDR*, vol. 4, 2017, p. 1.

[30] SQLite, "SQLite transactional SQL database engine," http://www.sqlite.org/.

[31] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A durable and energy efficient main memory using phase change memory technology," in *ACM SIGARCH computer architecture news*, vol. 37, no. 3. ACM, 2009, pp. 14–23.

[32] E. Kültürsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu, "Evaluating STT-RAM as an energy-efficient main memory alternative," in *Proceedings of International Symposium on Performance Analysis of Systems and Software*, 2013.

[33] Intel Corporation, "Revolutionary memory technology," http://www.intel.com/content/www/us/en/architecture-and-technology/non-volatile-memory.html, 2019.

[34] J. Elliott and J.-H. Choi, "Flash memory summit keynote 6: Memory innovations navigating the big data era," in *Flash Memory Summit*, Santa Clara, CA, Aug. 2022, accessed: 2022-12-13. [Online]. Available: https://www.flashmemorysummit.com/English/Conference/Keynotes_2022.html

[35] J. Izraelevitz, J. Yang, L. Zhang, J. Kim, X. Liu, A. Memaripour, Y. J. Soh, Z. Wang, Y. Xu, S. R. Dulloor, J. Zhao, and S. Swanson, "Basic performance measurements of the intel optane DC persistent memory module," *CoRR*, vol. abs/1903.05714, 2019. [Online]. Available: http://arxiv.org/abs/1903.05714

[36] Linux, "Direct access for files," https://www.kernel.org/doc/Documentation/filesystems/dax.txt.

[37] J. Xu and S. Swanson, "NOVA: A log-structured file system for hybrid volatile/non-volatile main memories," in *14th USENIX Conference on File and Storage Technologies, FAST 2016, Santa Clara, CA, USA, February 22-25, 2016.*, 2016, pp. 323–338. [Online]. Available: https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[38] Linux, "Direct access for files," https://www.kernel.org/doc/Documentation/filesystems/dax.txt, 2019.

[39] "XFS: DAX support," https://lwn.net/Articles/635514/.

[40] D. S. Rao, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System software for persistent memory," in *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014*, 2014, pp. 15:1–

15:15. [Online]. Available: http://doi.acm.org/10.1145/2592798.2592814

[41] PostgreSQL, "PostgreSQL: The World's Most Advanced Open Source Relational Database," https://www.postgresql.org/, 1996.

[42] A. L. Drapeau, K. W. Shirriff, J. H. Hartman, E. L. Miller, S. Seshan, R. H. Katz, K. Lutz, D. A. Patterson, E. K. Lee, P. M. Chen *et al.*, "Raid-ii: A high-bandwidth network file server," in *Proceedings of 21 International Symposium on Computer Architecture.* IEEE, 1994, pp. 234–244.

[43] "Exim mail server." [Online]. Available: https://www.exim.org/

[44] Facebook, "Universal compaction," https://github.com/pmem/pmem-rocksdb, 2017.

[45] "Redis: In-memory data structure store," https://pmem.io/2020/09/25/memkeydb.html, 2019.

[46] S. K. Lee, J. Mohan, S. Kashyap, T. Kim, and V. Chidambaram, "Recipe: converting concurrent DRAM indexes to persistent-memory indexes," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, 2019, pp. 462–477. [Online]. Available: https://doi.org/10.1145/3341301.3359635

[47] D. Hwang, W.-H. Kim, Y. Won, and B. Nam, "Endurable transient inconsistency in {Byte-Addressable} persistent {B+-Tree}," in *16th USENIX Conference on File and Storage Technologies (FAST 18)*, 2018, pp. 187–200.

[48] J. Yang, Y. Yue, and K. Rashmi, “Segcache: a memory-efficient and scalable in-memory key-value cache for small objects,” in *18th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 21)*, 2021.

[49] Facebook, “Universal compaction,” https://memcached.org/blog/persistent-memory/, 2017.

[50] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, “{PowerGraph}: Distributed {Graph-Parallel} computation on natural graphs,” in *10th USENIX symposium on operating systems design and implementation (OSDI 12)*, 2012, pp. 17–30.

[51] D. Nguyen, A. Lenharth, and K. Pingali, “A lightweight infrastructure for graph analytics,” in *Proceedings of the twenty-fourth ACM symposium on operating systems principles*, 2013, pp. 456–471.

[52] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, “Pytorch: An imperative style, high-performance deep learning library,” *Advances in neural information processing systems*, vol. 32, 2019.

[53] Intel, “Persistent memory development kit,” http://pmem.io.

[54] H. Volos, A. J. Tack, and M. M. Swift, “Mnemosyne: Lightweight persistent memory,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI, 2011, pp. 91–104.

[55] R. Kadekodi, S. K. Lee, S. Kashyap, T. Kim, and V. Chidambaram, “SplitFS: A File System that Minimizes Software Overhead in File Systems for Persistent Memory,” in *Proceedings of the 27th ACM*

Symposium on Operating Systems Principles (SOSP '19), Ontario, Canada, October 2019.

[56] J. Xu, J. Kim, A. Memaripour, and S. Swanson, "Finding and fixing performance pathologies in persistent memory software stacks," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, 2019, pp. 427–439. [Online]. Available: https://doi.org/10.1145/3297858.3304077

[57] R. Kadekodi, S. Kadekodi, S. Ponnapalli, H. Shirwadkar, G. R. Ganger, A. Kolli, and V. Chidambaram, "Winefs: a hugepage-aware file system for persistent memory that ages gracefully," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 804–818.

[58] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. E. Anderson, "Strata: A cross media file system," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017*. ACM, 2017, pp. 460–477. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132770

[59] "Symas Lightning Memory-Mapped Database," https://symas.com/products/lightning-memory-mapped-database/.

[60] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck, "Network file system (nfs) version 4 protocol," Tech. Rep., 2003.

[61] "Symas Lightning Memory-Mapped Database," https://symas.com/products/lightning-memory-mapped-database/.

[62] Y. Chen, Y. Lu, B. Zhu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and J. Shu, "Scalable persistent memory file system with {Kernel-Userspace} collaboration," in *19th USENIX Conference on File and Storage Technologies (FAST 21)*, 2021, pp. 81–95.

[63] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee, "Better I/O Through Byte-addressable, Persistent Memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ser. SOSP '09, 2009, pp. 133–146.

[64] X. Wu, S. Qiu, and A. L. N. Reddy, "SCMFS: A file system for storage class memory and its extensions," *TOS*, vol. 9, no. 3, pp. 7:1–7:23, 2013. [Online]. Available: http://doi.acm.org/10.1145/2501620.2501621

[65] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible file-system interfaces to storage-class memory," in *Proceedings of the Ninth European Conference on Computer Systems*, ser. EuroSys '14, 2014.

[66] N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch, "A five-year study of file-system metadata," *ACM Transactions on Storage (TOS)*, vol. 3, no. 3, pp. 9–es, 2007.

[67] J. R. Douceur and W. J. Bolosky, "A large-scale study of file-system contents," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 1, pp. 59–70, 1999.

[68] D. T. Meyer and W. J. Bolosky, "A study of practical deduplication," *ACM Transactions on Storage (ToS)*, vol. 7, no. 4, pp. 1–20, 2012.

[69] K. A. Smith and M. I. Seltzer, "File system aging—increasing the relevance of file system benchmarks," in *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. ACM, 1997, pp. 203–213.

[70] S. Kadekodi, V. Nagarajan, and G. R. Ganger, "Geriatrix: Aging what you see and what you don't see. a file system aging approach for modern storage systems," 2018.

[71] A. Conway, A. Bakshi, Y. Jiao, W. Jannen, Y. Zhan, J. Yuan, M. A. Bender, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and M. Farach-Colton, "File systems fated for senescence? nonsense, says science!" in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST 17)*, 2017, pp. 45–58.

[72] A. Conway, E. Knorr, Y. Jiao, M. A. Bender, W. Jannen, R. Johnson, D. Porter, and M. Farach-Colton, "Filesystem aging: It's more usage than fullness," in *11th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 19)*, 2019.

[73] N. Agrawal, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "Generating realistic impressions for file-system benchmarking," *ACM Transactions on Storage (TOS)*, vol. 5, no. 4, pp. 1–30, 2009.

[74] J. Leskovec and R. Sosič, "Snap: A general-purpose network analysis and graph-mining library," *ACM Transactions on Intelligent Systems and Technology (TIST)*, vol. 8, no. 1, p. 1, 2016.

[75] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “Consistency Without Ordering,” in *Proceedings of the 10th USENIX Symposium on File and Storage Technologies (FAST ’12)*, San Jose, California, Feb. 2012, pp. 101–116.

[76] D. R. Engler, M. F. Kaashoek, and J. O’Toole, “Exokernel: An operating system architecture for application-level resource management,” in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado, USA, December 3-6, 1995*, 1995, pp. 251–266. [Online]. Available: http://doi.acm.org/10.1145/224056.224076

[77] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, “IRON file systems,” in *Proceedings of the 20th ACM Symposium on Operating Systems Principles 2005, SOSP 2005, Brighton, UK, October 23-26, 2005*, 2005, pp. 206–220. [Online]. Available: https://doi.org/10.1145/1095810.1095830

[78] T. P. P. Council, “Tpc benchmark c, standard specification version 5,” 2001.

[79] SQLite, “SQLite transactional SQL database engine,” http://www.sqlite.org/, 2019.

[80] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, “Benchmarking cloud serving systems with ycsb,” in *Proceedings of the 1st ACM symposium on Cloud computing.* ACM, 2010, pp. 143–154.

[81] Google, “Leveldb,” https://github.com/google/leveldb, 2019.

[82] Facebook, "RocksDB Tuning Guide," https://github.com/facebook/rocksdb/wiki/RocksDB-Tuning-Guide, 2017.

[83] G. Wallace, F. Douglis, H. Qian, P. Shilane, S. Smaldone, M. Chamness, and W. Hsu, "Characteristics of backup workloads in production systems," in *Proceedings of the 10th USENIX conference on File and Storage Technologies, FAST 2012, San Jose, CA, USA, February 14-17, 2012*, 2012, pp. 33–48. [Online]. Available: https://www.usenix.org/conference/fast12/characteristics-backup-workloads-production-systems

[84] V. Tarasov, E. Zadok, and S. Shepler, "Filebench: A flexible framework for file system benchmarking," *login: The USENIX Magazine*, vol. 41, no. 1, pp. 6–12, 2016.

[85] T. E. Anderson, M. Canini, J. Kim, D. Kostić, Y. Kwon, S. Peter, W. Reda, H. N. Schuh, and E. Witchel, "Assise: Performance and availability via client-local {NVM} in a distributed file system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 1011–1027.

[86] Y. Wang, "A statistical study for file system meta data on high performance computing sites," *Master's thesis, Southeast University*, 2012.

[87] J. Xu, L. Zhang, A. Memaripour, A. Gangadharaiah, A. Borase, T. B. D. Silva, S. Swanson, and A. Rudoff, "Nova-fortis: A fault-tolerant non-volatile main memory file system," in *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai,*

China, October 28-31, 2017. ACM, 2017, pp. 478–496. [Online]. Available: http://doi.acm.org/10.1145/3132747.3132761

[88] J. Mohan, A. Martinez, S. Ponnapalli, P. Raju, and V. Chidambaram, "Finding crash-consistency bugs with bounded black-box crash testing," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 33–50.

[89] L. kernel developers, "Linux POSIX file system test suite," https://lwn.net/Articles/276617/, 2008.

[90] LMDB, "Database Microbenchmarks," http://www.lmdb.tech/bench/microbench/, 2012.

[91] A. Wilson, "The new and improved filebench," in *6th USENIX Conference on File and Storage Technologies (FAST 08)*, 2008.

[92] E. Amaro, C. Branner-Augmon, Z. Luo, A. Ousterhout, M. K. Aguilera, A. Panda, S. Ratnasamy, and S. Shenker, "Can far memory improve job throughput?" in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–16.

[93] J. Gu, Y. Lee, Y. Zhang, M. Chowdhury, and K. G. Shin, "Efficient memory disaggregation with infiniswap," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, 2017, pp. 649–667.

[94] Y. Shan, S.-Y. Tsai, and Y. Zhang, "Distributed shared persistent memory," in *Proceedings of the 2017 Symposium on Cloud Computing*, 2017, pp. 323–337.

[95] "Userfaultfd," https://man7.org/linux/man-pages/man2/userfaultfd.2.html, 2020.

[96] A. Raybuck, T. Stamler, W. Zhang, M. Erez, and S. Peter, "Hemem: Scalable tiered memory management for big data applications and real nvm," in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, 2021, pp. 392–407.

[97] R. Kadedodi, S. Kadekodi, S. Ponnapalli, H. Shirwadkar, G. Ganger, A. Kolli, and V. Chidambaram, "WineFS: a hugepage-aware file system for persistent memory that ages gracefully," in *Proceedings of the 28th ACM Symposium on Operating Systems Principles (SOSP '21)*, October 2021.

[98] Intel, "Syscall Intercept — Userspace syscall intercepting library," https://github.com/pmem/syscall_intercept, 2020.

[99] Z. Cao and S. Dong, "Characterizing, modeling, and benchmarking rocksdb key-value workloads at facebook," in *18th USENIX Conference on File and Storage Technologies (FAST'20)*, 2020.

[100] "XFS: There and back ... and there again?" https://lwn.net/Articles/638546/.

[101] "APFS in Detail: Overview," http://dtrace.org/blogs/ahl/2016/06/19/apfs-part1/.

[102] "Btrfs History," https://en.wikipedia.org/wiki/Btrfs#.

[103] "XFS History," https://en.wikipedia.org/wiki/XFS#History.

[104] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, 2009.

[105] D. Gouk, S. Lee, M. Kwon, and M. Jung, "Direct access,{High-Performance} memory disaggregation with {DirectCXL}," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 287–294.

[106] H. A. Maruf, H. Wang, A. Dhanotia, J. Weiner, N. Agarwal, P. Bhattacharya, C. Petersen, M. Chowdhury, S. Kanaujia, and P. Chauhan, "Tpp: Transparent page placement for cxl-enabled tiered-memory," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, 2023, pp. 742–755.

[107] M. Jung, "Hello bytes, bye blocks: Pcie storage meets compute express link for memory expansion (cxl-ssd)," in *Proceedings of the 14th ACM Workshop on Hot Topics in Storage and File Systems*, 2022, pp. 45–51.

[108] Y. Shan, W. Lin, Z. Guo, and Y. Zhang, "Towards a fully disaggregated and programmable data center," in *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2022, pp. 18–28.

[109] L. A. Eisner, T. Mollov, and S. J. Swanson, *Quill: Exploiting fast non-volatile memory by transparently bypassing the file system.* Department of Computer Science and Engineering, University of California, San Diego, 2013.

[110] T. von Eicken, A. Basu, V. Buch, and W. Vogels, "U-net: A user-level network interface for parallel and distributed computing," in *Proceedings of the Fifteenth ACM Symposium on Operating System Principles, SOSP 1995, Copper Mountain Resort, Colorado,*

USA, December 3-6, 1995, 1995, pp. 40–53. [Online]. Available: https://doi.org/10.1145/224056.224061

[111] M. DeBergalis, P. F. Corbett, S. Kleiman, A. Lent, D. Noveck, T. Talpey, and M. Wittle, "The direct access file system," in *Proceedings of the FAST '03 Conference on File and Storage Technologies, March 31 - April 2, 2003, Cathedral Hill Hotel, San Francisco, California, USA*, 2003. [Online]. Available: http://www.usenix.org/events/fast03/tech/debergalis.html

[112] G. A. Gibson, D. Nagle, K. Amiri, F. W. Chang, E. M. Feinberg, H. Gobioff, C. Lee, B. Ozceri, E. Riedel, D. Rochberg, and J. Zelenka, "File server scaling with network-attached secure disks," in *Proceedings of the 1997 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Seattle, Washington, USA, June 15-18, 1997*, 1997, pp. 272–284. [Online]. Available: https://doi.org/10.1145/258612.258696

[113] E. K. Lee and C. A. Thekkath, "Petal: Distributed virtual disks," in *ASPLOS-VII Proceedings - Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts, USA, October 1-5, 1996.*, 1996, pp. 84–92. [Online]. Available: https://doi.org/10.1145/237090.237157

[114] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson, "Providing safe, user space access to fast, solid state disks," in *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2012, London, UK, March*

3-7, 2012, 2012, pp. 387–400. [Online]. Available: https://doi.org/10.1145/2150976.2151017

[115] A. Belay, A. Bittau, A. J. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *10th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2012, Hollywood, CA, USA, October 8-10, 2012*, 2012, pp. 335–348. [Online]. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/belay

[116] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion, "IX: A protected dataplane operating system for high throughput and low latency," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, 2014, pp. 49–65. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/belay

[117] S. Peter, J. Li, I. Zhang, D. R. K. Ports, D. Woos, A. Krishnamurthy, T. E. Anderson, and T. Roscoe, "Arrakis: The operating system is the control plane," in *11th USENIX Symposium on Operating Systems Design and Implementation, OSDI '14, Broomfield, CO, USA, October 6-8, 2014.*, 2014, pp. 1–16. [Online]. Available: https://www.usenix.org/conference/osdi14/technical-sessions/presentation/peter

[118] P. R. Barham, "A fresh approach to file system quality of service," in *Proceedings of 7th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'97).* IEEE, 1997, pp. 113–122.

[119] J. Choi, J. Kim, and H. Han, "Efficient memory mapped file I/O for in-memory file systems," in *9th USENIX Workshop on Hot Topics in Storage and File Systems, HotStorage 2017, Santa Clara, CA, USA, July 10-11, 2017*, 2017. [Online]. Available: https://www.usenix.org/conference/hotstorage17/program/presentation/choi

[120] A. Mathur, M. Cao, S. Bhattacharya, A. T. Andreas Dilge and, and L. Vivier, "The New Ext4 filesystem: Current Status and Future Plans," in *Ottawa Linux Symposium (OLS '07)*, Ottawa, Canada, Jul. 2007.

[121] T. Mason, T. D. Doudali, M. Seltzer, and A. Gavrilovska, "Unexpected performance of intel® optane dc persistent memory," *IEEE Computer Architecture Letters*, vol. 19, no. 1, pp. 55–58, 2020.

[122] N. Amit, A. Tai, and M. Wei, "Don't shoot down tlb shootdowns!" in *Proceedings of the Fifteenth European Conference on Computer Systems*, 2020, pp. 1–14.

[123] J. Yang, J. Izraelevitz, and S. Swanson, "Orion: A distributed file system for {Non-Volatile} main memory and {RDMA-Capable} networks," in *17th USENIX Conference on File and Storage Technologies (FAST 19)*, 2019, pp. 221–234.

[124] Y. Lu, J. Shu, Y. Chen, and T. Li, "Octopus: an {RDMA-enabled} distributed persistent memory file system," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 773–785.

[125] OpenSFS and EOFS, "The Lustre File System," https://www.lustre.org/, 2023.

[126] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The google file system," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '03, 2003.

[127] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *Proceedings of the 7th symposium on Operating systems design and implementation*, 2006, pp. 307–320.

[128] Z. Ruan, M. Schwarzkopf, M. K. Aguilera, and A. Belay, "{AIFM}:{High-Performance},{Application-Integrated} far memory," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 315–332.

[129] A. Dragojević, D. Narayanan, M. Castro, and O. Hodson, "{FaRM}: Fast remote memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014, pp. 401–414.

[130] M. K. Aguilera, N. Amit, I. Calciu, X. Deguillard, J. Gandhi, S. Novakovic, A. Ramanathan, P. Subrahmanyam, L. Suresh, K. Tati *et al.*, "Remote regions: a simple abstraction for remote memory," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 775–787.

[131] P. Keleher, A. L. Cox, S. Dwarkadas, and W. Zwaenepoel, "Treadmarks: Distributed shared memory on standard workstations and operating systems," *Distributed Shared Memory: Concepts and Systems*, pp. 211–227, 1994.

[132] J. B. Carter, "Design of the munin distributed shared memory system," *Journal of Parallel and Distributed Computing*, vol. 29, no. 2,

pp. 219–227, 1995.

[133] J. Nelson, B. Holt, B. Myers, P. Briggs, L. Ceze, S. Kahan, and M. Oskin, "{Latency-Tolerant} software distributed shared memory," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 291–305.

[134] Q. Cai, W. Guo, H. Zhang, D. Agrawal, G. Chen, B. C. Ooi, K.-L. Tan, Y. M. Teo, and S. Wang, "Efficient distributed memory management with rdma and caching," *Proceedings of the VLDB Endowment*, vol. 11, no. 11, pp. 1604–1617, 2018.

[135] S. Kaxiras, D. Klaftenegger, M. Norgren, A. Ros, and K. Sagonas, "Turning centralized coherence and distributed critical-section execution on their head: A new approach for scalable distributed shared memory," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*, 2015, pp. 3–14.

[136] A. Greenberg, J. Hamilton, D. A. Maltz, and P. Patel, "The cost of a cloud: research problems in data center networks," pp. 68–73, 2008.

[137] Y. Zhou, H. M. Wassel, S. Liu, J. Gao, J. Mickens, M. Yu, C. Kennelly, P. Turner, D. E. Culler, H. M. Levy *et al.*, "Carbink:{Fault-Tolerant} far memory," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022, pp. 55–71.

[138] *Proceedings of the 26th Symposium on Operating Systems Principles, Shanghai, China, October 28-31, 2017.* ACM, 2017. [Online]. Available: http://dl.acm.org/citation.cfm?id=3132747