UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Revolutionizing Storage Stack for Persistent Memories with NOVA File System

Permalink
https://escholarship.org/uc/item/6ff3h9cq

Author
Xu, Jian

Publication Date
2018

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Revolutionizing Storage Stack for Persistent Memories with NOVA File System

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Computer Science (Computer Engineering)

by

Jian Xu

Committee in charge:

Professor Steven Swanson, Chair
Professor George Porter
Professor Tajana Simunic Rosing
Professor Paul Siegel
Professor Geoffrey Voelker

2018

Copyright
Jian Xu, 2018
All rights reserved.

The dissertation of Jian Xu is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California San Diego

2018

DEDICATION

To my parents, my wife and my children who have always supported me.

EPIGRAPH

And from my pillow,

looking forth by light of moon or favouring stars,

I could behold the antechapel where the statue stood

of Newton with his prism and silent face,

The marble index of a mind for ever

voyaging through strange seas of Thought, alone.

– William Wordsworth

TABLE OF CONTENTS

Signature Page iii

Dedication iv

Epigraph v

Table of Contents vi

List of Figures viii

List of Tables x

Acknowledgements xi

Vita xiii

Abstract of the Dissertation xv

Chapter 1 Introduction 1

Chapter 2 Motivation and background 8
- 2.1 Non-volatile Memories 9
- 2.2 Challenges for NVMM software stack 12
  - 2.2.1 Exploiting NVMM Performance 12
  - 2.2.2 Enforcing Write Ordering 13
  - 2.2.3 File System Consistency and Reliability 14
  - 2.2.4 Providing Atomicity Guarantee 15
- 2.3 Building complex atomic operations 16
- 2.4 File systems for NVMM 17

Chapter 3 NOVA: A Log-Structured Persistent Memory File System 20
- 3.1 NOVA Design Overview 22
- 3.2 NOVA Implementation 25
  - 3.2.1 NVMM data structures and space management 25
  - 3.2.2 Atomicity and enforcing write ordering 30
  - 3.2.3 Directory operations 32
  - 3.2.4 Atomic file operations 35
  - 3.2.5 DAX mmap 38
  - 3.2.6 Garbage collection 38
  - 3.2.7 Shutdown and Recovery 41
- 3.3 Evaluation 42
  - 3.3.1 Experimental setup 43

3.3.2 Microbenchmarks 44
3.3.3 Macrobenchmarks 46
3.3.4 Scalability 48
3.3.5 Garbage collection efficiency 50
3.3.6 Recovery overhead 51
3.4 Summary 52

Chapter 4 NOVA-Fortis: Making Persistent Memory File System Fault-tolerant . . . 54
4.1 Snapshots 57
4.1.1 Taking and Restoring Snapshots 58
4.1.2 Snapshot Management 58
4.1.3 Snapshots for DAX `mmap()`'d Files 62
4.1.4 Design Decisions and Evaluation 65
4.2 Handling Data Corruption in NOVA-Fortis 67
4.2.1 Detecting and Correcting Media Errors 68
4.2.2 Tick-Tock Metadata Protection 69
4.2.3 Protecting File Data 69
4.2.4 Minimizing Vulnerability to Scribbles 71
4.2.5 Protecting DRAM Data Structures and Preventing Scribbles 73
4.3 Performance Trade-offs 74
4.3.1 Microbenchmarks 74
4.3.2 Macrobenchmarks 76
4.3.3 NVMM Storage Utilization 79
4.4 Summary 79

Chapter 5 Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks 82
5.1 Adapting applications to NVMM 84
5.1.1 SQLite 85
5.1.2 Tokyo Cabinet and LMDB 87
5.1.3 RocksDB and Redis 89
5.1.4 Best Practices 92
5.1.5 Reducing NVMM file system journaling overhead 93
5.1.6 FIO: scalability and NUMA limitation 96
5.2 Improving File System Scalability 97
5.2.1 Concurrent file read/write 98
5.2.2 Directory Accesses 100
5.2.3 NUMA Scalability 102
5.3 Summary 103

Chapter 6 Conclusion 104

Bibliography 106

LIST OF FIGURES

Figure 3.1: NOVA data structure layout 26
Figure 3.2: NOVA superblock struct 27
Figure 3.3: NOVA inode 28
Figure 3.4: NOVA journal structure 29
Figure 3.5: NOVA allocator node 29
Figure 3.6: NOVA log appending 31
Figure 3.7: NOVA directory entry structure 32
Figure 3.8: NOVA inode update entry 33
Figure 3.9: NOVA directory operation 34
Figure 3.10: NOVA file write entry structure 36
Figure 3.11: NOVA file structure and write operation 37
Figure 3.12: NOVA log cleaning 40
Figure 3.13: NOVA file operation latency 44
Figure 3.14: PostMark throughput 45
Figure 3.15: Filebench throughput 47
Figure 3.16: NOVA scalability on file operations 49
Figure 3.17: NOVA garbage collection efficiency 50

Figure 4.1: NOVA-Fortis snapshot manifest structure 59
Figure 4.2: NOVA-Fortis snapshot manifest entry 60
Figure 4.3: Snapshots in NOVA-Fortis 61
Figure 4.4: Snapshots of `mmap()`'d data 64
Figure 4.5: Snapshot performance impact 66
Figure 4.6: Snapshot operation latency 66
Figure 4.7: NOVA-Fortis space layout 70
Figure 4.8: Scribble size and metadata bytes at risk 72
Figure 4.9: NOVA-Fortis file operation latency 75
Figure 4.10: NOVA-Fortis latencies for NVDIMM-N 75
Figure 4.11: NOVA-Fortis random read/write bandwidth 76
Figure 4.12: NOVA-Fortis application performance 77
Figure 4.13: NOVA-Fortis storage overheads 79

Figure 5.1: SQLite SET throughput with different journaling modes 86
Figure 5.2: Tokyo Cabinet (TC) and LMDB SET throughput 88
Figure 5.3: Redis MSET throughput 90
Figure 5.4: RocksDB SET throughput 91
Figure 5.5: JDD performance 95
Figure 5.6: Latency break for 4 KB append and RocksDB SET 95
Figure 5.7: FIO read/write bandwidth on many-core machine 96
Figure 5.8: NOVA lock-free read/write 99
Figure 5.9: Concurrent 4 KB read/write throughput 100

Figure 5.10: Concurrent `create`/`unlink` throughput 101
Figure 5.11: NUMA-awareness in the file system 102

LIST OF TABLES

Table 2.1: Properties of non-volatile memory technologies 10

Table 3.1: NVMM emulation characteristics on PMEP 43
Table 3.2: Filebench workload characteristics 46
Table 3.3: NOVA GC pages statistics 51
Table 3.4: Recovery workload characteristics 52
Table 3.5: NOVA recovery time 52

Table 4.1: Snapshot manifest size 67
Table 4.2: Application benchmarks 77

ACKNOWLEDGEMENTS

I'm grateful for the support from many wonderful people during the long adventure that has led me to this milestone in my life.

I would like to thank my adviser Professor Steven Swanson for his kind support as chair of my committee. He has provided many precious advises and suggestions during my PhD study, and his guidance has helped me overcome the difficulties and achieve my goals throughout my graduate school time. His mentorship in both research and career planning is invaluable to me.

Besides, I want to express my gratitude to Professor Tajana Simunic Rosing, Professor Geoffrey Voelker, Professor George Porter and Professor Paul Siegel for their valuable comments and suggestions as my committee members.

I also want to thank the other members of the Non-volatile Systems Laboratory. I would like to thank Meenakshi Sundaram Bhaskaran for his valuable suggestions during my first two years. I am aslo grateful to Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Juno Kim and Andy Rudoff for their help in the papers that we co-authored.

I want to thank my parents for their love and support during my whole life. I want to thank my wife Shuangquan Chen. She has quitted her job in China to accompany me. Her love and support are invaluable to me. I want to thank my children, Colin and Grace. You are my angels and bring light and happiness to my life.

Chapter 1, Chapter 2 and Chapter 3 contains material from "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories", by Jian Xu and Steven Swanson, which is appeared in the 14th USENIX Conference on File and Storage Technologies. The dissertation author is the primary investigator and first author of this paper. The materials are copyright ©2016 by USENIX Association.

Chapter 1, Chapter 2 and Chapter 4 contains material from "NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System", by Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito da Silva, Andy Rudoff and Steven Swanson, which is appeared in the Proceedings of the 26th ACM Symposium on Operating Systems Principles. The dissertation author is the primary investigator and first author of this paper. The materials are copyright ©2017 by Association for Computing Machinery.

Chapter 1, Chapter 2 and Chapter 5 contains material from “Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks”, by Jian Xu, Juno Kim and Steven Swanson, which is submitted to OSDI 2018. The dissertation author is the primary investigator and first author of this paper.

Permission to make digital or hard copies of part of all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage.

VITA

2005	Bachelor of Electronic Engineering, Shanghai Jiao Tong University
2008	Master of Telecommunications and Information System, Shanghai Jiao Tong University
2008-2012	Senior Software Develop Engineer, AMD Shanghai R&D Center
2012-2018	Graduate Student Researcher, University of California San Diego
2013	Internship, HGST, a Western Digital Company
2015	Internship, VMware Inc.
2016	Candidate of Philosophy, University of California San Diego
2017	Internship, Microsoft Research
2018	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Andy Rudoff and Steven Swanson. "NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System." In the 26th ACM Symposium on Operating Systems Principles (SOSP), 2017.

Jian Xu and Steven Swanson. "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories." In the 14th USENIX Conference on File and Storage Technologies (FAST), 2016.

Dejan Vucinic, Qingbo Wang, Cyril Guyot, Robert Mateescu, Filip Blagojevic, Luiz Franca-Neto, Damien Le Moal, Trevor Bunker, Jian Xu, Steven Swanson and Zvonimir Bandic. "DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express." In the 12th USENIX Conference on File and Storage Technologies (FAST), 2014.

Meenakshi Sundaram Bhaskaran, Jian Xu and Steven Swanson. “BankShot: Caching slow storage in fast non-volatile memory.” In the 1st Workshop on Interactions of NVM/Flash with Operating Systems and Workloads (INFLOW), 2013.

Bharathan Balaji, Jian Xu, Rajesh Gupta and Yuvraj Agarwal. “Sentinel: An Occupancy Based HVAC Actuation System using existing WiFi Infrastructure in Commercial Buildings.” In the 11th ACM Conference on Embedded Networked Sensor Systems (SenSys), 2013.

ABSTRACT OF THE DISSERTATION

Revolutionizing Storage Stack for Persistent Memories with NOVA File System

by

Jian Xu

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2018

Professor Steven Swanson, Chair

Fast non-volatile memories (NVMs) are appearing on the processor memory bus alongside DRAM, becoming non-volatile main memories (NVMMs). The resulting hybrid memory systems will provide software with low-latency, high-bandwidth access to persistent data. However, managing, accessing, maintaining consistency and providing protection for data stored in NVMM raises a host of challenges. Existing file systems built for spinning or solid-state disks introduce software overheads that would obscure the performance that NVMs should provide, but proposed NVMM file systems either incur similar overheads or fail to provide the strong consistency and integrity guarantees that applications require.

This thesis first presents NOVA, a log-structured file system designed to maximize performance on hybrid memory systems while providing strong consistency guarantees. NOVA adapts conventional log-structured file system techniques to exploit the fast random access that NVMs provide. In particular, it maintains separate logs for each inode to improve concurrency, appends fine-grained metadata to the log to provide low-overhead atomicity, and stores file data outside the log to minimize log size and reduce garbage collection costs. NOVA's logs provide metadata and data atomicity and focus on simplicity and reliability, keeping complex metadata structures in DRAM to accelerate lookup operations. For operations that span multiple logs, NOVA uses lightweight journaling to provide fast atomic transaction semantics. In case of system failure, the per-inode logging design provides vast parallelism and fast recovery.

NOVA excels in metadata-intensive and write-intensive workloads. Experimental results show that NOVA provides 22% to 216$\times$ throughput improvement compared to state-of-the-art file systems, and 3.1$\times$ to 13.5$\times$ improvement compared to file systems that provide equally strong data consistency guarantees.

NVMM has different failure models from disks and SSDs. Disk I/O errors are not vital for the operating system, but memory errors can hang the entire OS. Persistent memory makes the issue worse: the error is durable and system reboot cannot remove it. How to handle persistent memory errors is still an open question. This thesis presents NOVA-Fortis, a fault-tolerant file system that based on NOVA and is both fast and resilient in the face of corruption due to media errors and software bugs. We identify and propose solutions for the unique challenges in adding fault tolerance and reliability techniques to a NVMM filesystem, and quantify the performance and storage overheads of these techniques. We find that NOVA-Fortis' reliability features consume 14.8% of the storage for redundancy and reduce application-level performance by between 2% and 38% compared to the same file system with the features removed. NOVA-Fortis outperforms DAX-aware file systems without reliability features by 1.5$\times$ on average. It outperforms reliable, block-based file systems running on NVMM by 3$\times$ on average.

Finally, the thesis evaluates existing applications and analyzes their access patterns to the file system. It finds out that existing NVMM file systems such as ext4-DAX and xfs-DAX do not perform well under applications' typical access patterns, like write-ahead logging (WAL). The thesis resolves the issue for both applications and file systems. On the application side, this thesis optimizes the access patterns and avoids the operations that result in high overhead. From the file system perspective, the thesis proposes a new fine-grained, scalable journaling module design for ext4 and improves the WAL performance of databases and key-value stores. The thesis also analyzes the file system scalabilty and NUMA impact on a multi-socket, multi-core machine, proposes and implements several optimizations on NOVA file system to fix the scalability and NUMA impact issues.

Chapter 1

Introduction

With the era of big data emerging, the ability to efficiently store and process huge data is crucial. People post tweets, photos, videos, etc. on social network websites and update their personal pages frequently, expecting their posts to be seen and stored immediately. Companies hosting web services store user data to persistent storage devices, and also process and analyze the data to perdict user activity and make business decisions. High-frequency traders require data storage and analysis latency to be reduced to microsecond level. To keep up with the rapid growth of data, the world is in need of storage systems with fast speed and large capacity. To meet this requirement, storage device vendors have produced NAND flash-based solid state drive (SSDs), and now they are moving to emerging non-volatile memory (NVM) technologies.

NVM technologies such as battery-backed NVDIMMs, spin-torque transfer, phase change, resistive memories [106, 49, 6, 60, 102] and Intel and Micron's 3D XPoint [5] technology try to provide persistent memories that have latency and bandwidth close to DRAM, promise to revolutionize I/O performance. There are several approaches on integrating NVMs into computer systems [27, 29, 36, 67, 74, 87, 108, 130], and the most exciting proposals place NVMs on the processor's memory bus alongside conventional DRAM, leading to hybrid volatile/non-volatile main memory systems [10, 101, 135, 144], as the memory bus is the fastest physical interface to access these new memories. Combining faster, volatile DRAM with slightly slower, denser non-volatile main memories (NVMMs) offers the possibility of storage systems that combine the best characteristics of both technologies.

Hybrid DRAM/NVMM storage systems present a host of opportunities and challenges for system designers. First, these systems need to minimize software overhead if they are to fully exploit NVMM's high performance and efficiently support more flexible access patterns. Second, they must provide the strong consistency guarantees that applications require and respect the limitations of emerging memories (e.g., limited program cycles). Third, they should allow the applications to access the NVMM via the memory interface, i.e. load and store instructions to reduce memory copy overhead. Fourth, they need to provide data integrity guarantees so that data is safe against media errors, memory errors and software bugs.

Conventional file systems are not suitable for hybrid memory systems because they are built for the performance characteristics of disks (spinning or solid state) and rely on disks' consistency guarantees (e.g., that sector updates are atomic) for correctness [95]. Hybrid memory systems are different from conventional storage systems on both counts: NVMMs provide vastly improved performance over disks so that the software stack efficienty becomes critical, and memory interface provides different consistency guarantees (e.g., 64-bit atomic stores) from disks.

Providing strong consistency guarantees is particularly challenging for memory-based file systems because maintaining data consistency in NVMM can be costly. Modern CPU and memory systems may reorder stores to memory to improve performance, breaking consistency in case of system failure. As a result, file systems running on NVMM need to guarantee that persistence ordering is complied to program ordering and explicitly flush data from the CPU's caches to enforce orderings. This is not available in existing disk and SSD file systems, and naively performing cache line flushing adds significant overhead and squanders the improved performance that NVMM can provide [17, 142].

Besides ordering requirements, many applications rely on atomic file system operations to ensure their own correctness. Existing mainstream file systems use journaling, shadow paging, or log-structuring techniques to provide atomicity. Unfortunately, none of these techniques fit NVMM systems perfectly. Journaling doubles the number of writes to the storage device and hence wastes bandwidth , and shadow paging file systems require a cascade of updates from the affected leaf nodes to the root. Both techniques imposes strict ordering requirements that reduce performance.

Log-structured file systems (LFSs) [104] group small random write requests into a larger sequential write that hard disks and NAND flash-based solid state drives (SSDs) can process efficiently. However, conventional LFSs rely on the availability of contiguous free regions, and maintaining those regions requires expensive garbage collection operations. As a result, recent research [113] shows that LFSs perform worse than journaling file systems on NVMM.

To adapt existing storage stack to NVMMs, researchers and open-source communities have proposed and built *native NVMM file systems* [37, 41, 136, 129, 39], and both Linux and Windows have created *adapted NVMM file systems* by adding support for NVMM to existing file systems, such as NTFS, ext4-DAX and xfs-DAX. Commercial applications also began to leverage NVMMs to improve performance and provide fast recovery [7].

Adding system support for NVMM brings a lot of potential benefits. The most obvious of these is faster file access via conventional POSIX file system interfaces, so that existing applications can achieve significant performance gains without changing, demonstrating the benefits of specialized NVMM file systems. A more efficient technique to exploit NVMM performance is *direct access (DAX)*. DAX file access allow applications to access a file's contents directly using load and store instructions, bypassing the system page cache and providing bare-metal performance; DAX-mmap allows an application to map the pages of an NVMM-backed file into its address space and then access it via load and store instructions. DAX-mmap removes all of the system software overhead for common-case accesses enabling the fastest-possible access to persistent data. Using DAX-mmap requires applications to adopt an mmap-based interface to storage, but recent research shows that performance gains can be significant [36, 130, 98, 79].

Despite all these progresses, there are several issues with the existing NVMM storage stack: First, adapted NVMM file systems only optimize the data access path for persistent memories, but the metadata update still have to go through the block-based software stack. As a result, they perform poorly on metadata-intensive workloads. Second, the storage stack is developed for hard disks and SSDs, it does not take into account the fast random concurrent accesses that NVMM can provide, and internal scalability bottlenecks prevent it from scaling on widely-deployed many-core, multi-socket platforms. Third, despite these NVMM-centric performance improvements, none of these file systems provide the data protection features necessary to detect and correct media errors, protect against data corruption due to misbehaving code, or perform consistent backups of the NVMM's contents. File system stacks in wide use (e.g., ext4 running atop LVM, Btrfs, and ZFS) provide some or all of these capabilities for block-based storage. If users are to trust NVMM file systems with critical data, they will need these features as well. Fourth, legacy applications can get performance gain on NVMM file systems, but they are not optimized for NVMM systems. Their I/O access patterns need thorough investigation and modifications if they want to maximize performance on NVMM file systems.

This thesis tries to resolve all these issues by changing the current Linux software storage stack for NVMM. The author designed and implemented a new log-structured NVMM file system called NOVA. NOVA proposes a new file system technique called per-inode logging, allowing for concurrent inode accesses and parallel recovery. NOVA combines journaling, logging and copy-on-write to provide atomicity guarantees, using different technique for different file system operations to improve performance. NOVA adopts a new garbage collection technique based on its novel linked-list log design and resolves the GC performance issue of conventional log-structured file systems.

To make NOVA robust against memory errors and software bugs, the author designed and implemented NOVA-Fortis by adding fault-tolerant features to NOVA. NOVA-Fortis leverages NOVA's log structure to perform file system snapshot with extremely low overhead. NOVA-Fortis also supports taking consistent snapshot when application is performing DAX-mmap, which is not available with current NVMM file systems. NOVA-Fortis provides protection againse media errors and software bugs by doing replication on metadata and RAID-4 parity on data.

Finally, the author investigates several legacy applications, analyze their access I/O patterns and propose optimiziations to boost application performance by making relative simple changes to the I/O path. The author also designed and implemented DAX-aware journaling module to improve ext4-DAX metadata operation performance, and figured out and resolved several scalability bottlenecks in the current Linux software storage stack.

This thesis is organized as follows. In Chapter 2 we survey the technological opportunities and challenges that motivate the research efforts in this thesis, including NVM technologies, file systems, consistency and fault-tolerant mechanisms.

In Chapter 3 we present the design, implementation and evaluation of NOVA, a high-performance, log-structured file system for hybrid volatile/non-volatile main memories with strong metadata and data atomicity guarantees. NOVA redesigns traditional log-structured file system techniques and adapts them to exploit the fast, concurrent random accesses provided by hybrid memory systems. NOVA adopts a novel per-inode logging design to maximize concurrency during file system operation and failure recovery. NOVA performs log appending in an atomic way by updating the log tail pointer atomically, achieves lower latency than existing journaling and shadow paging systems. For operations that span multiple inodes, NOVA uses lightweight journaling that journals inode log tails instead of metadata and data. NOVA stores inode logs as linked lists, avoids the contiguous requirement of conventional log-structured file systems. It stores data outside the log, so that the recovery process only needs to scan a small portion of the NVMM space. NOVA performs data overwriting in a copy-on-write way, reclaims state data pages immediately, significantly reducing garbage collection overhead and allowing NOVA to sustain high performance under write-intensive workloads. We evalute NOVA on a Intel hardware emulation platform, and experimental results show that NOVA gains significant performance improvement (22% to 216×) over state-of-the-art file systems in write-intensive and metadata-intensive workloads. It out-performs file systems that provide equally strong data consistency guarantees by 3.1× to 13.5×.

In Chapter 4 we strength the NOVA filesystem with fault-tolerance features, including snapshot support and metadata and data protection against memory errors and software scribbles. NOVA-Fortis identifies unique challenges when applying state-of-the-art fault-tolerance features to NVMM file systems. First, NVMM system reports media errors as fatal memory errors rather than disk failures. Second, NVMM file system must support DAX-style memory mapping that maps non-volation memory pages directly into the applications' address space, so that application can modify the NVMM data directly. This is not possible with block-based file systems, and new technologies to support snapshot and data protection when applications are performing DAX-mmap are required. Third, these new fault-tolerance features should not sacrifice performance, and the performance impact needs to be re-evaluated. NOVA-Fortis is the first work on NVMM systems fault-tolerance. It resolves these unique challenges by adopting new designs and technologies, and it is reliable against memory errors and software scribbles. We quantify the performance and storage overhead of NOVA-Fortis' reliability features and evaluate their effectiveness at preventing corruption of both file system metadata and data. NOVA-Fortis consumes 14.8% file system space for data protection. It outperforms other state-of-the-art DAX file systems without reliability features by 1.5× on average, and outperforms reliable, block-based file systems running on NVMM by 3× on average.

In Chapter 5 we examined the performance of NVMM-aware storage software stacks to better understand the trade-offs between adapted NVMM file systems (ext4-DAX [134], xfs-DAX [34]) and native NVMM file systems. We investigated the I/O access patterns of existing applications and understood how they can best benefit from migrating from hard disks and SSDs to NVMMs. We find that by making relatively small modifications these applications can achieve substantial performance gains. We also addressed several sources of inefficiency in current NVMM file systems and the Linux storage stack. We found out existing NVMM journaling file systems such as ext4-DAX and xfs-DAX have high journaling overhead in metadata operations, and demonstrated how NVMM-aware journaling can boost performance for applications. Finally, we located several scalability bottlenecks of Linux VFS and file system, and fixed them on the NOVA file system.

Acknowledgments

This chapter contains material from "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories", by Jian Xu and Steven Swanson, which appears in the 12th USENIX Conference on File and Storage Technologies (FAST 2016). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from "NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System", by Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito da Silva, Andy Rudoff and Steven Swanson, which appears in the 26th ACM Symposium on Operating Systems Principles (SOSP 2017). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from "Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks", by Jian Xu, Juno Kim and Steven Swanson, which is submitted to the 13th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 2018). The dissertation author is the primary investigator and first author of this paper.

Chapter 2

Motivation and background

Non-volatile memory technologies have enabled byte-addressable persistent memories with decreased latency and increased density, and systems equipped with non-volatile main memories allow applications to access data directly. With these attractive characteristics, NVMM has become a very promising storage medium for next-generation storage devices. We expect NVMM to appear in large scale data centers as well as mobile devices, e.g. smart phones, tablets and digital cameras.

NVMM requires additional care to utilize the underlying persistent memory due to its unique characteristics. NVMM has limited program cycles and only supports 8-byte atomic updates, and stores to NVMM are not guaranteed persistency unless explicitly flush the cache lines. Modern CPUs may reorder stores and hence compromise the ordering requirements that applications want. Since NVMM is drastically different from magnetic medium and NAND flash memory, it calls for a scrutiny of previous software storage stack design and offer additional opportunities for optimization.

Existing file systems have long been optimized for spinning hard disk and SSDs. Naively running them on top of NVMM cannot fully exploit the high performance that NVMM can provide, and may compromise consistency in case of system failure since NVMM has different atomicity characteristics. NVMM file systems allow applications to directly access file data, but they are still not efficient on metadata operations, and sacrifice strong consistency guarantee and fault-tolerance features to get better performance.

Log-structured file systems [105, 4] have been proposed to meet the characteristics of spinning hard disk. However, directly running a conventional log-structured file system on NVMM can lead to suboptimal performance and efficiency. The background I/O activity, i.e. log cleaning significantly impact the file system throughput, and existing log-structured file systems suffer from the "wandering tree" problem [20].

The rest of this chapter presents a comprehensive introduction to the background of this thesis. Section 2.1 describes the physical characteristics of non-volatile memory technologies. Section 2.2 describes the unique challenges for NVMM software storage stack design. Section 2.3 presents the existing mechanisms of transactional support inside file systems, and Section 2.4 lists the progress on developing NVMM file systems.

2.1 Non-volatile Memories

Emerging non-volatile memory technologies, such as spin-torque transfer RAM (STT-RAM) [60, 89], phase change memory (PCM) [26, 35, 65, 102], resistive RAM (ReRAM) [44, 122], and 3D XPoint memory technology [5], promise to provide fast byte-addressable persistent memories. Suzuki *et al.* [123] provides a survey of these technologies and their evolution over time. Table 2.1 provides an estimation of the properties of these non-volatile memory technologies. The performance of 3D XPoint memory is still not revealed.

These memories have different strengths and weaknesses that make them useful in different parts of the memory hierarchy. STT-RAM has ultra low latency and it may eventually appear in on-chip, last-level caches [143], but its large cell size limits capacity and its feasibility as a DRAM replacement. PCM and ReRAM are denser than DRAM, and may enable very large, non-volatile main memories. However, their relatively high latencies make it unlikely that they will fully replace DRAM as main memory. The 3D XPoint memory technology recently announced by Intel and Micron is claimed to offer performance up to 1,000 times faster than NAND flash [5]. It will appear in both SSDs and on the processor memory bus. As a result, we expect to see hybrid volatile/non-volatile memory hierarchies become common in large systems. In this section, we briefly introduce these memory technologies.

Table 2.1: **Properties of non-volatile memory technologies.**

Item	DRAM	PCM	STT-RAM	ReRAM
Byte Addressable	Yes	Yes	Yes	Yes
Persistent	No	Yes	Yes	Yes
Read Time (ns)	10	40	3	35
Write Time (ns)	10	140	3	36
Bit density (Gbit/cm^2)	9.1	13.5	0.12	9.5

Phash change memory (PCM) [102, 26] is a new persistent memory technology that is more scalable than DRAM. PCM is made of phase change material, which is typically a chalcogenide, and each memory cell in PCM is separated by a resistor and the phase change material. Phase changes are performed by injecting current into the resistor-chalcogenide junction. The current heats the chalcogenide to 650°C, causing it to change state. The amplitude and width of the injected current pulse determines the programmed state. Phase change memory typically operates in two states. The SET and RESET states are defined as the crystalline (low-resistance) and amorphous (high-resistance) phases of the chalcogenide, respectively. When the memory cell is injected by a high and short current pulse, the storage element is switched to RESET state. The short pulse quickly quenching the heat generation and freezing the chalcogenide into the amorphous state. In contrast, a moderate and long current pulse will set the memory cell by gradually cooling down the chalcogenide and inducing crystal growth. Since SET requires longer duration of the current, it has higher latency and causes a inbalance between the PCM read and write performance.

Spin-transfer torque magnetic random access memory (STT-MRAM) [60, 8] is a novel, magnetic memory technology that combines the capacity and cost benefits of DRAM, high performance of SRAM and virtually unlimited endurance. STT-MRAM levelages the quantum-mechanical spin property of electron: The electron charge determines how the electrons behave in an electric field, the spin of electron determines the behavior in a magnetic field. Both the charge and spin of electrons can be used to store information. STT-MRAM is believed to be mature and ready for a concerted push to enter into production at 65 nm and below, for any of various applications such as embedded, SRAM, and DRAM products.

ReRAM [122] is a type of non-volatile memory that works by changing the resistance across a dielectric solid-state material, often referred to as a memristor. ReRAM stores information by generating defects in a thin oxide layer, known as oxygen vacancies (oxide bond locations where the oxygen has been removed), which can subsequently charge and drift under an electric field. The motion of oxygen ions and vacancies in the oxide would be analogous to the motion of electrons and holes in a semiconductor.

Compared to PCM, ReRAM operates at a faster timescale, the switch time can be less than 10 ns. It has a simpler, smaller cell structure than MRAM, with unit size as small as $4F^2$. ReRAM consumes less power than NAND flash, and hence can be used in low-power circumstances. ReRAM is scalable below 30 nm, and is considered a promising candidate for CPU caches [82].

3D XPoint is a non-volatile memory technology developed by Intel and Micron, announced in July 2015 [5]. Currently it is available on the open market in SSD format, and the prototype of 3D XPoint DIMM is provided to research facilities.

The internal technology of 3D XPoint is still unknown. It was claimed using chalcogenide materials for selector and storage parts that differs from other PCM technologies, but today it is thought as a subset of ReRAM. The 3D XPoint memory cell does not need a transistor, so its density will be four times of DRAM. The initial prices of 3D XPoint are less than DRAM, but more than flash memories.

NVDIMM [106, 51] is able to retain its contents when the system power is removed. NVDIMM couples DRAM with persistent storage devices such as SSDs. The NVDIMM system loads and stores data to DRAM during normal operations, and dumps the data into non-volatile memory if the power fails, using an on-board backup power source. NVDIMM achieves the same performance as DRAM, and it does not have concerns about wear and device lifespan. However, including a second storage device to achieve non-volatility (and the on-board backup power source) increases the product cost.

2.2 Challenges for NVMM software stack

NVMM technologies present several challenges to file system designers. The most critical of these focus on exploiting the NVMMs' performance against software overheads, enforcing ordering among updates to ensure consistency, providing atomic semantics and fault-tolerance guarantees.

2.2.1 Exploiting NVMM Performance

The low latencies of NVMMs alters the trade-offs between hardware and software latency. In conventional storage systems, the latency of slow storage devices (e.g., disks) dominates access latency, so software efficiency is not critical. Previous work has shown that with fast NVMM, software costs can dominate memory latency, squandering the performance that NVMMs could provide [18, 28, 131, 139].

Since NVMMs reside on the processor's memory bus, software should be able to access them directly via loads and stores. To meet this requirement, system designers have proposed a new NVMM access technique called *Direct Access (DAX)*. DAX works in two ways: For POSIX file accesses such as `read` and `write`, DAX bypasses the system page cache and eliminates one memory copy. A more powerful usage of DAX is DAX-mmap. DAX-mmap maps the NVMM physical pages that hold file data directly into an application's address space, so that the application can access and modify file data directly. DAX-mmap bypasses the whole OS and gives applications the fastest possible access to stored data and allows them to build complex, persistent, pointer-based data structures. The typical usage model would have the application create a large file in an NVMM file system, use `mmap()` to map it into its own address space, and then rely on a userspace library [36, 130, 97] to manage it. Currently all mainstream Windows and Linux file systems have preliminary DAX support [134, 34, 48].

Although DAX-mmap grants direct access to applications, conventional `mmap()`-based applications use `msync()` to persist the updates. `msync()` is page-based, and it is expensive and non-atomic. Applications and NVMM libraries have to use user-space cache flushing and persit barriers to build complex data structures, but they are difficult to use and error-prone.

2.2.2 Enforcing Write Ordering

Modern processors may reorder store operations to improve performance, and the persistence order may not be same as programming order. The CPU's memory consistency protocol makes guarantees about the ordering of memory updates, but existing models (with the exception of research proposals [37, 94]) do not provide guarantees on when updates will reach NVMMs. As a result, a power failure may leave the data in an inconsistent state.

NVMM-aware software can avoid this by explicitly flushing caches and issuing memory barriers to enforce write ordering. The x86 architecture provides the `clflush` instruction to flush a CPU cacheline, but `clflush` is strictly ordered and needlessly invalidates the cacheline, incurring a significant performance penalty [17, 142]. Memory barriers such as `mfence` instruction enforce order on memory operations before and after the barrier, but `mfence` does not enforce persistence orderings, it only guarantees all CPUs have the same view of the memory.

Intel has proposed new instructions that fix these problems, including `clflushopt` (a more efficient version of `clflush`) and `clwb` (to explicitly write back a cache line without invalidating it) [53, 145]. We believe the NVMM software stack should be built with these instructions in mind.

2.2.3 File System Consistency and Reliability

Apart from their core function of storing and retrieving data, file systems also provide facilities to protect the data they hold from corruption due to system failures, media errors, and software bugs (both in the file system and elsewhere). Highly-reliable file systems like ZFS [21] and Btrfs [1] provide two key features to protect data and metadata: The ability to take snapshots of the file system (to facilitate backups) and set of mechanisms to detect and recover from data corruption due to media errors and other causes. Existing DAX file systems provide neither of these features, limiting their usefulness in mission-critical applications. Below, we discuss the importance of each feature and existing approaches.

Snapshots Snapshots provide a consistent image of the file system at a moment in time. Their most important application is facilitating consistent backups without unmounting the file system, affording protection against catastrophic system failures and the accidental deletion or modification of files. Many modern file systems have built-in support for snapshots [1, 21, 61]. In other systems the underlying storage stack (e.g., LVM in Linux) can take snapshots of the underlying block device. Neither existing DAX-enabled NVMM file systems nor current low-level NVMM drivers support snapshots, making consistent online backups impossible.

Data Corruption File systems are subject to a wide array of data corruption issues including media errors that cause storage media to return incorrect values and software errors that store incorrect data to the media. Data corruption and software errors in the storage stack have been thoroughly studied for hard disks [110, 12, 109], SSDs [80, 88, 111] and DRAM-based memories [112, 120, 121]. The results of DRAM-based studies may apply to DRAM-based

NVDIMMs, but there have been no (publicly-available) studies of error behaviors in emerging NVMM technologies.

Storage devices use error-correcting codes (ECC) to protect against media errors. Errors that ECC detects but cannot correct result in uncorrectable media errors. For block-based storage, these errors appear as read or write failures from the storage driver. Intel NVMM-based systems report these media errors via an unmaskable machine-check exception (MCE). Software errors can also cause data corruption. If the file system is buggy, it may write data in the wrong place or fail to write at all. Other code in the kernel can corrupt file system data by "scribbling" [63] on file system data structures or data buffers. Scribbles are an especially critical problem for NVMM file systems, since the NVMM is mapped into the kernel's address space. As a result, all of file system's data and metadata are always vulnerable to scribbles.

2.2.4 Providing Atomicity Guarantee

Many applications require some kind of transactional support [46], and POSIX-style file system semantics require many operations to be atomic (i.e., to execute in an "all or nothing" fashion). A transaction is a group of operations on the data and/or metadata. If any part of the transaction fails, then the entire transaction fails. The failed transaction should not have any effect on the logical or physical view of the system, depending on the context.

Some transactional updates apply both metadata and data in a file system or application. For instance, appending to a file atomically updates the file data and changes the file's length and modification time. Many applications rely on atomic file system operations for their own correctness. For example, mail server applications use the atomic POSIX `rename` operation to perform atomic writes, and database management systems such as MySQL [3] can use the file system atomic write ability to improve I/O performance.

All the application and file system level atomicity implementation rely on some kind of hardware atomic update guarantees. Storage devices typically provide only rudimentary guarantees about atomicity. Disks provide atomic sector writes and processors guarantee only that 8-byte (or smaller), aligned stores are atomic. To build the more complex atomic updates that file systems require, programmers must use more complex techniques. In the following section, we will further discuss file system transaction techniques.

2.3 Building complex atomic operations

Existing file systems use a variety of techniques like journaling, shadow paging, or log-structuring to provide atomicity guarantees. These work in different ways and incur different types of overheads.

Journaling Journaling (or write-ahead logging) is widely used in journaling file systems [47, 54, 68, 134] and databases [83, 92] to ensure atomicity. A journaling system writes all updates to a journal on persistent storage before applying them and, in case of power failure, replays the journal to restore the system to a consistent state. Journaling requires writing data twice: once to the log and once to the target location, and to improve performance journaling file systems usually only journal metadata. Journaling consumes more space due to logs, and log flushing can be a performance bottleneck since file systems need to persist the log before applying changes.

Shadow paging Shadow paging, or copy-on-write technique is adopted by several file systems [37, 21, 50, 1]. Shadow paging file systems never overwrite existing data. Instead, they write a new version of the data to another location and reclaim the old data. Shadow paging technique relies heavily on the tree structure to provide atomicity, inserts the new version of data pages into the tree by updating the nodes between the pages and root. The resulting cascade of updates is called "wandering tree" problem [20] and can be potentially expensive.

Log-structuring Log-structured systems were originally designed to exploit hard disk drives' high performance on sequential accesses. Log-structured systems buffer random writes in memory and convert them into larger, sequential writes to the disk, making the best of hard disks' strengths.

Log-structured file systems (LFSs) are proposed for both hard disks [104, 114, 4] and SSDs [66], as SSDs also perform best under sequential workloads [22, 31]. Log-structured merge tree [91] (LSM) is the key data structure of some key-value stores such as LevelDB and RocksDB.

Although LFS is an elegant idea, implementing it efficiently is complex, because LFSs rely on writing sequentially to contiguous free regions of the disk. To ensure a consistent supply of such regions, LFSs constantly clean and compact the log to reclaim space occupied by stale data. Log cleaning adds overhead and degrades the performance of LFSs [9, 115]. To reduce cleaning overhead, some LFS designs separate hot and cold data and apply different cleaning policies to each [132, 133]. As LFS performs writes out-of-place, it also has the wandering tree problem as shadow paging file systems do.

2.4 File systems for NVMM

Researchers, companies and open-source communities have designed and implemented NVMM-based file systems. We divide NVMM file systems into two groups. *Native* NVMM filesystems are designed especially for NVMMs. They exploit the byte-addressability of NVMM storage and can dispense with many of the optimizations (and associated complexity) that block-based file systems implement to hide the poor performance of disks.

BPFS [37] is the first native NVMM 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 [41, 99] is a lightweight DAX file system that bypasses the block layer and file system page cache to improve performance. PMFS uses journaling for metadata updates. SCMFS [136] 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 [129] implements the file system interface and functionality in user space to provide low-latency access to data in NVMM. Aerie journals metadata but does not support data atomicity or mmap operation. Strata [64] is a “cross media” file system that runs partly in userspace. It provides strong atomicity and high performance, but does not support DAX.

Adapted NVMM file systems (or just “adapted file systems”) are block-based file systems extended to implement NVMM features, like DAX and DAX-mmap. Xfs-DAX [34], ext4-DAX [134] and NTFS [48] all have modes in which they become adapted file systems. Xfs-DAX and ext4-DAX are the state-of-the-art adapted NVMM 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 [30, 32]. So far, adapted file systems have been built subject to constraints that limit how much they can change to support NVMM. 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 NVMM file systems usually store efficient data structures such as B$^+$tree and radix tree in NVMM to manage inodes and files. However, persistent data structures are difficult to implement and usually incur large performance overheads [33, 85, 128, 140] due to cache flush and memory ordering operations.

Acknowledgments

This chapter contains material from “NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories”, by Jian Xu and Steven Swanson, which appears in the 12th USENIX Conference on File and Storage Technologies (FAST 2016). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from “NOVA-Fortis: A Fault-Tolerant Non-Volatile Main

Memory File System", by Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito da Silva, Andy Rudoff and Steven Swanson, which appears in the 26th ACM Symposium on Operating Systems Principles (SOSP 2017). The dissertation author is the primary investigator and first author of this paper.

This chapter contains material from "Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks", by Jian Xu, Juno Kim and Steven Swanson, which is submitted to the 13th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 2018). The dissertation author is the primary investigator and first author of this paper.

Chapter 3

NOVA: A Log-Structured Persistent Memory File System

Emerging NVM technologies will such as spin-torque transfer, phase change, resistive memories [6, 60, 102] and Intel and Micron's 3D XPoint [5] technology promise to improve I/O performance by a wide margin, and we expect hybrid DRAM/NVMM storage systems to be available on open market soon. These systems should improve conventional file access performance and allow applications to abandon slow, read/write file interfaces in favor of faster memory-mapped, load/store access interfaces. They will also allow for increased concurrency and efficiently support more flexible access patterns. File systems must realize these advantages while still providing the strong consistency guarantees that applications require and respecting the limitations of emerging memories (e.g., limited program cycles).

Conventional file systems are not suitable for hybrid memory systems because they are designed for hard disks and SSDs, but NVMM has entire different characteristics. NVMM has latency and bandwidth that are close to DRAM, and existing software storage stack is too heavy to exploit the performance. NVMM also has different atomic guarantees, and CPU may reorder writes to the memory bus so explicit cache line flushing is needed. As a result, naively running conventional file systems cannot fully exploit the performance that NVMM provides, and may break consistency in case of a system failure.

To overcome all these limitations, we present the *NOn-Volatile memory Accelerated (NOVA)* file system [137]. NOVA adapts conventional log-structured file system techniques to exploit the fast random access provided by hybrid memory systems. NOVA supports massive concurrency and resolves the garbage collection performance issue of conventional log-structured file systems.

NOVA is designed purely for NVMM characteristics, and several aspects of NOVA set it apart from previous log-structured file systems. NOVA assigns each inode a separate log to maximize concurrency during normal operation and recovery. NOVA stores the logs as linked lists, so they do not need to be contiguous in memory, and it uses atomic updates to a log's tail pointer to provide atomic log append. For operations that span multiple inodes, NOVA uses lightweight journaling.

NOVA puts file data outside the log and performs copy-on-write or inplace-update for data, so the recovery process only needs to scan a small fraction of the NVMM. NOVA can immediately reclaim data pages when they become stale, significantly reducing garbage collection overhead and be able to sustain good performance even when the file system is nearly full.

By designing and implementing NOVA, we make the following contributions:

- It describes NOVA, a log-structured NVMM file system built for hybrid file systems that provides metadata and data atomicity.
- It extends existing log-structured file system techniques to exploit the characteristics of hybrid memory systems.
- It demonstrates that NOVA outperforms existing journaling, shadow paging, and log-structured file systems running on hybrid memory systems.

- It shows that NOVA provides these benefits across a range of proposed NVMM technologies.

We evaluate NOVA using a collection of micro- and macro-benchmarks on a hardware-based NVMM emulator. We find that NOVA is significantly faster than existing file systems in a wide range of applications and outperforms file systems that provide the same data consistency guarantees by between 3.1× and 13.5× in write-intensive workloads. We also measure garbage collection and recovery overheads, and we find that NOVA provides stable performance under high NVMM utilization levels and fast recovery in case of system failure.

The remainder of this chapter is organized as follows. Section 3.1 gives a overview of NOVA architecture and Section 3.2 describes the implementation in detail. Section 3.3 evaluates NOVA, and Section 3.4 concludes.

3.1 NOVA Design Overview

NOVA is a log-structured POSIX file system that builds on the strengths of LFS and optimized for NVMM characteristics. Because NOVA targets a different storage technology, NOVA looks very different from conventional log-structured file systems that are built to maximize disk bandwidth.

NOVA is designed based on four observations. First, conventional LFS only has a single log. This makes sense for hard disks where there is a single disk head and improving spatial locality is paramount, but limits concurrency. Since NVMMs support fast, highly concurrent random accesses, using multiple logs does not negatively impact performance. Second, the complexity of cleaning logs stems primarily from the need to supply contiguous free regions of storage, but this is not necessary in NVMM, because random access is relatively cheap. Third, logs are simple and easy to implement correctly in NVMM, but not efficient for search operations such as directory lookup file random access. In contrase, data structures that support fast search such as B-trees are more difficult to implement correctly and efficiently in NVMM [33, 85, 128, 140]. Fourth, conventional file systems work on top of the block layer and have to issue I/O requests of page size. NVMM is byte-addressable and NVMM file systems can get rid of block layer and perform fine-grained updates on NVMM. Based on these observations, we made the following design decisions in NOVA.

Give each inode its own log. Each inode in NOVA has its own log, allowing concurrent updates across files without synchronization. This structure allows for high concurrency both in file access and during recovery, since NOVA can replay multiple logs simultaneously. NOVA also guarantees that the number of valid log entries is small (on the order of the number of extents in the file), which ensures that scanning the log is fast. Per-inode logging is the key difference between NOVA and conventional LFS, since NOVA is designed purely for NVMM and NVMM provides fast concurrent random accesses.

Keep logs in NVMM and indexes in DRAM. NOVA keeps log and file data in NVMM and builds radix trees [71] in DRAM to quickly perform search operations, making the in-NVMM data structures simple and efficient. We use a radix tree because there is a mature, well-tested, widely-used implementation in the Linux kernel. The leaves of the radix tree point to entries in the log which in turn point to file data. The indexes are recoverable by traversing the logs.

Use logging and lightweight journaling for complex atomic updates. NOVA is log-structured because this provides cheaper atomic updates than journaling and shadow paging. To atomically write data to a log, NOVA first appends data to the log and then atomically updates the log tail to commit the updates, thus avoiding both the duplicate writes overhead of journaling file systems and the cascading update costs of shadow paging systems.

Some directory operations, such as a move between directories, span multiple inodes and NOVA uses journaling to atomically update multiple logs. NOVA first writes data at the end of each inode's log, and then journals the log tail updates to update them atomically. NOVA journaling is lightweight since it only involves log tails (as opposed to file data or metadata) and no POSIX file operation operates on more than four inodes.

Implement the log as a singly linked list. The locality benefits of sequential logs are less important in NVMM-based storage, so NOVA uses a linked list of 4 KB NVMM pages to hold the log and stores the next page pointer in the end of each log page.

Allowing for non-sequential log storage provides three advantages. First, allocating log space is easy since NOVA does not need to allocate large, contiguous regions for the log. Second, NOVA can perform log cleaning at fine-grained, page-size granularity. Third, reclaiming log pages that contain only stale entries requires just a few pointer assignments.

Perform fine-grained metadata updates. Since NVMM is byte-addressable, NOVA performs metadata operations such as inode updates and log appending in a fine-grained way. Unlike other DAX file systems that still use block layer to do coarse-grained metadata updates, NOVA bypasses the block layer and writes sub-page modifications to NVMM directly. As a result, NOVA performs much better than DAX file systems in metadata-intensive operations.

Put allocoator in DRAM. Unlike traditional file systems that use on-disk bitmap to store allocator information, NOVA puts the allocator in DRAM. This is because the allocator information is changed frequently during running, and to make it consistent file systems have to perform journaling which impacts performance significantly. By putting the allocator in DRAM, NOVA need not use journaling for single file append operations. NOVA stores the allocator in NVMM upon clean unmount to accelerate mount process. In case of system failure, NOVA can recover the allocator by walking the inode table and logs.

Put file data outside the log. The inode logs in NOVA do not contain file data. Instead, NOVA uses copy-on-write for modified pages and appends metadata about the write to the log. The metadata describe the update and point to the data pages. Section 3.2.4 describes file write operation in more detail.

Using copy-on-write for file data is useful for several reasons. First, it makes the log shorter, accelerating the recovery process. Second, it makes garbage collection simpler and more efficient, since NOVA never has to copy file data out of the log to reclaim a log page. Third, reclaiming stale pages and allocating new data pages are both easy, since they just require adding and removing pages from in-DRAM free lists. Fourth, since it can reclaim stale data pages immediately, NOVA can sustain performance even under heavy write loads and high NVMM utilization levels.

The next section describes the implementation of NOVA in more detail.

3.2 NOVA Implementation

NOVA was originally implemented in the Linux kernel version 4.0 and now it works on kernel 4.13. NOVA uses the existing NVMM hooks in the kernel and has passed the xfstests [72] and Linux POSIX file system test suite [100]. The source code is available on GitHub: https://github.com/NVSL/Linux-nova. In this section we first describe the overall NOVA file system layout, the data structures and its atomicity and write ordering mechanisms. Then, we describe how NOVA performs atomic directory, file, and DAX-mmap operations. Finally we discuss garbage collection and recovery in NOVA.

3.2.1 NVMM data structures and space management

Figure 3.1 shows the high-level layout of NOVA data structures in a region of NVMM it manages. NOVA divides the NVMM space into three logical categories: global data structures, logs and data pages. Global data structures in NOVA include superblock, inode tables and journals. The inode tables contain inodes, the journals provide atomicity to directory operations, and the remaining area contains NVMM log and data pages. We designed NOVA with scalability in mind: NOVA maintains an inode table, journal, and NVMM free page list at each CPU to avoid global

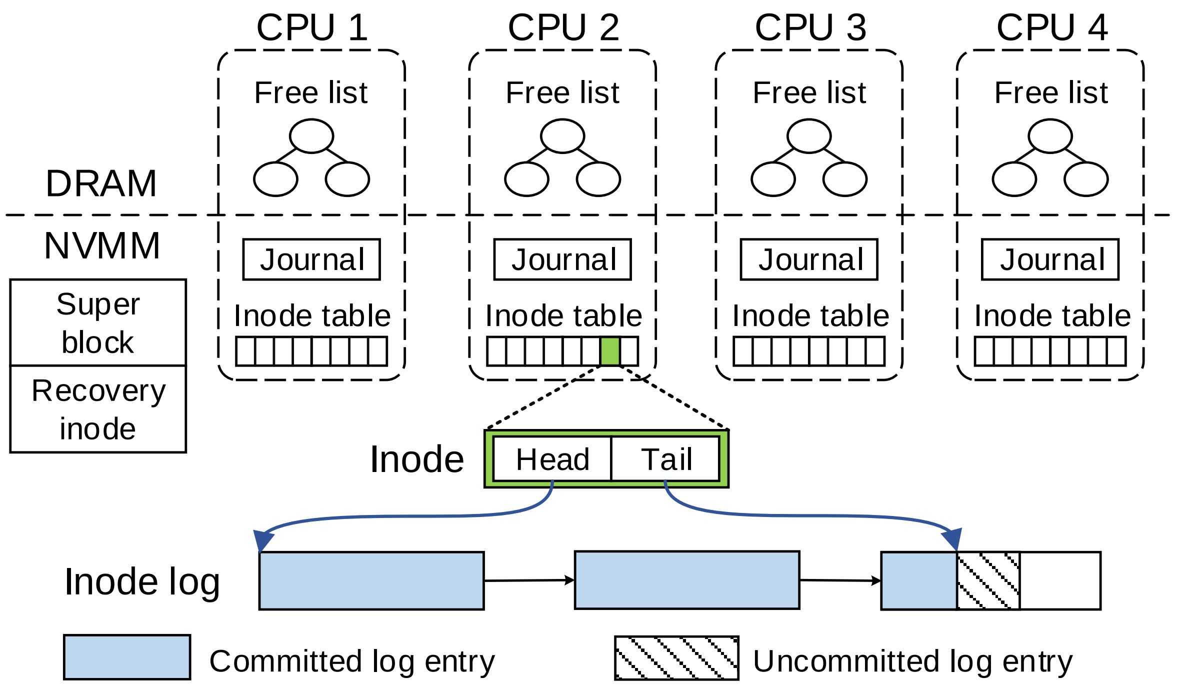

Figure 3.1: **NOVA data structure layout** NOVA has per-CPU free lists, journals and inode tables to ensure good scalability. Each inode has a separate log consisting of a singly linked list of 4 KB log pages; the tail pointer in the inode points to the latest committed entry in the log.

locking and scalability bottlenecks.

Superblock Fig 3.2 shows the NOVA superblock struct. NOVA superblock contains global file system information and is kept simple and small to reduce update overheads. The information of superblock includes NOVA magic number, the blocksize NOVA uses and mount time and write time. The most important field is the `s_size`, which describes the total file system size. Other global file system information can be rebuilt during recovery. NOVA uses a 32bit checksum to ensure the integrity of the superblock.

Inode and inode table Fig 3.3 shows the layout of NOVA inode. NOVA inode holds the metadata of file or directory. A NOVA inode contains pointers to the head and tail of its log. The log is a linked list of 4 KB pages, and the tail always points to the latest committed log entry. NOVA scans the log from head to tail to rebuild the DRAM data structures when the system

```
/* nova_def.h */
struct nova_super_block {
    __le32 s_sum;                /* checksum of this sb */
    __le32 s_magic;              /* magic signature */
    __le32 s_padding32;          /* padding */
    __le32 s_blocksize;          /* blocksize in bytes */
    __le64 s_size;               /* FS total size in bytes */
    char s_volume_name[16];      /* volume name */
    __le32 s_mtime;              /* mount time */
    __le32 s_wtime;              /* write time */
} __attribute((__packed__));
```

Figure 3.2: **NOVA superblock struct.**

accesses the inode for the first time.

Unlike other file systems, inode attributes such as file size and modification time are only flushed to NVMM when the file system evicts the inode or on unmount. NOVA does not update most of the inode attributes in-place; Instead, it appends updates to the log. The only exceptions are log tail pointer and `i_atime`. In case of a power failure, the inode attributes are recovered by walking through the log when rebuilding the DRAM data structures.

NOVA initializes each inode table as a 2 MB block array of inodes. Each NOVA inode is 128 byte long and aligned on 128-byte boundary, and each inode table initially holds 16,384 inodes. Given the inode number NOVA can easily locate the target inode. NOVA assigns new inodes to each inode table in a round-robin order, so that inodes are evenly distributed among inode tables. If the inode table is full, NOVA extends it by building a linked list of 2 MB sub-tables. To reduce the inode table size, each NOVA inode contains a valid bit and NOVA reuses invalid inodes for new files and directories. Per-CPU inode tables avoid the inode allocation contention and allow for parallel scanning in failure recovery.

Journal Fig 3.4 shows the lightweight journal struct in NOVA. A NOVA journal is a 4 KB circular buffer and NOVA manages each journal with a <enqueue, dequeue> pointer pair. To coordinate updates that across multiple inodes, NOVA first appends log entries to each log, and

```
/* nova_def.h */
struct nova_inode {
    u8 i_rsvd;                     /* reserved */
    u8 valid;                      /* Is this inode valid? */
    u8 deleted;                    /* Is this inode deleted? */
    u8 i_blk_type;                 /* data block size */
    __le32 i_flags;                /* Inode flags */
    __le64 i_size;                 /* Size of data in bytes */
    __le32 i_ctime;                /* Inode modification time */
    __le32 i_mtime;                /* Inode data modification time */
    __le32 i_atime;                /* Access time */
    __le16 i_mode;                 /* File mode */
    __le16 i_links_count;          /* Links count */
    __le64 i_xattr;                /* Extended attribute block */
    __le32 i_uid;                  /* Owner Uid */
    __le32 i_gid;                  /* Group Id */
    __le32 i_generation;           /* File version (for NFS) */
    __le32 i_create_time;          /* Create time */
    __le64 nova_ino;               /* nova inode number */

    __le64 log_head;               /* Log head pointer */
    __le64 log_tail;               /* Log tail pointer */

    struct {
        __le32 rdev;               /* major/minor # */
    } dev;                         /* device inode */

    __le32      csum;
    /* Leave 8 bytes for inode table tail pointer */
} __attribute((__packed__));
```

Figure 3.3: **NOVA inode structure.** The log head and tail pointers point to the per-inode log. NOVA uses atomic 8-byte store to update tail pointer and commit log entries.

then starts a transaction by appending all the affected log tails to the current CPU's journal enqueue, and updates the enqueue pointer. After propagating the updates to the target log tails, NOVA updates the dequeue equal to enqueue to commit the transaction. For a `create` operation, NOVA journals the directory's log tail pointer and new inode's valid bit. During power failure recovery, NOVA checks each journal and rolls back any updates between the journal's dequeue

```
/* nova.h */
struct journal_ptr_pair {
    __le64 journal_dequeue;
    __le64 journal_enqueue;
} __attribute((__packed__));

struct nova_lite_journal_entry {
    __le64 type;        /* Journal entry type */
    __le64 addr;        /* Address of data */
    __le64 value;       /* Value of data */
    __le32 padding;
    __le32 csum;        /* Checksum of the entry */
} __attribute((__packed__));
```

Figure 3.4: **NOVA journal structures.** Each NOVA journal is a 4 KB circular buffer.

and enqueue. NOVA maintains a journal at each CPU core to support concurrent transactions with high scalability. NOVA only allows one open transaction at a time on each core. For each directory operation, the kernel's virtual file system (VFS) layer locks all the affected inodes, so concurrent transactions never modify the same inode.

NVMM space management The allocator efficiency plays an important part in file system performance. To make NVMM allocation and deallocation fast, NOVA divides NVMM into pools, one per CPU, and keeps lists of free NVMM pages in DRAM. If no pages are available in the current CPU's pool, NOVA allocates pages from the largest pool, and uses per-pool locks to provide protection. This allocation scheme is similar to scalable memory allocators like Hoard [16].

Fig 3.5 shows the allocator structure in NOVA. To reduce the allocator size, NOVA uses a red-black tree to keep the free list sorted by address, allowing for efficient merging and providing $O(\log n)$ deallocation. Each tree node represents a free NVMM extent. The `range_low` is the start block number of the extent, and `range_high` represents the end block number. The red-block tree makes the free extents sorted, so adjacent free extents can be merged to reduce the allocator fragmentation. To improve performance, NOVA does not store the allocator state in NVMM

```
/* nova.h */
struct nova_range_node {
    struct rb_node node;        /* Red-block tree node */
    unsigned long range_low;
    unsigned long range_high;
};
```

Figure 3.5: **NOVA allocator node.** NOVA divides the NVMM space among the CPUs, and manages each NVMM pool with a red-black tree.

during operation. On a normal shutdown, it records the allocator state to the recovery inode's log and restores the allocator state by scanning the all the inodes' logs in case of a power failure.

NOVA allocates log space aggressively to avoid the need to frequently resize the log. Initially, an inode's log contains one page. When the log exhausts the available space, NOVA allocates sufficient new pages to double the log space and appends them to the log. If the log length is above a given threshold (256 pages), NOVA appends a fixed number of pages each time.

3.2.2 Atomicity and enforcing write ordering

NOVA provides fast atomicity for metadata and data updates using a technique that combines log structuring and lightweight journaling. This technique uses three mechanisms.

64-bit atomic updates Modern processors support 64-bit atomic writes for volatile memory and NOVA assumes that 64-bit writes to NVMM will be atomic as well. NOVA uses 64-bit in-place writes to directly modify metadata for some operations (e.g., the file's *atime* for reads) and uses them to commit updates to the log by updating the inode's log tail pointer.

Logging NOVA uses the inode's log to record operations that modify a single inode. These include operations such as `write`, `msync` and `chmod`. The logs are independent of one another. For inode operations, the log entries contain metadata; For file write operation, the log entry contains the metadata that describe the `write`, and pointers to data pages. Log appending commits

```
new_tail = append_to_log(inode->tail, entry);
/* writes back the log entry cachelines */
clwb(inode->tail, entry->length);
sfence();        /* orders subsequent store */
inode->tail = new_tail;
clwb(&inode->tail, sizeof(inode->tail)); /* Commit and flush */
```

Figure 3.6: **Pseudocode for enforcing write ordering.** NOVA commits the log entry to NVMM strictly before updating the log tail pointer.

both metadata and data in a transactional way.

Lightweight journaling For directory operations that require changes to multiple inodes (e.g., `create`, `unlink` and `rename`), NOVA uses lightweight journaling to provide atomicity. At any time, the data in any NOVA journal are small—no more than 64 bytes: The most complex POSIX `rename` operation involves up to four inodes, and NOVA only needs 16 bytes to journal each inode: 8 bytes for the address of the log tail pointer and 8 bytes for the value.

Enforcing write ordering NOVA relies on three write ordering rules to ensure consistency. First, it commits data and log entries to NVMM before updating the log tail. Second, it commits journal data to NVMM before propagating the updates. Third, it commits new versions of data pages to NVMM before recycling the stale versions. If NOVA is running on a system that supports `clflushopt` and `clwb`, it uses the code in Figure 3.6 to enforce the write ordering.

First, the code appends the entry to the log. Then it flushes the affected cache lines with `clwb`. Next, it issues a `sfence` to prevent the tail update from occurring before the log appending. Finally, it writes back the log tail pointer update to commit the operation.

If the platform does not support the new instructions, NOVA uses `movntq`, a non-temporal move instruction that bypasses the CPU cache hierarchy to perform direct writes to NVMM and uses a combination of `clflush` and `sfence` to enforce the write ordering.

3.2.3 Directory operations

NOVA pays close attention to directory operations because they have a large impact on application performance [76, 70, 125]. NOVA supprots all the major directory operations, including `link`, `symlink` and `rename` with high efficiency.

NOVA directories comprise two parts: the log of the directory's inode in NVMM and a radix tree in DRAM. Figure 3.9 shows the relationship between these components. The directory's log holds two kinds of entries: directory entries (dentry) and inode update entries.

```
/* nova.h */
struct nova_dentry {
    u8 entry_type;        /* Log entry type */
    u8 name_len;          /* length of the dentry name */
    u8 reassigned;        /* Currently deleted */
    u8 invalid;           /* Log invalid now? */
    __le16 de_len;        /* length of this dentry */
    __le16 links_count;   /* Link count */
    __le32 mtime;         /* For both mtime and ctime */
    __le32 csum;          /* entry checksum */
    __le64 ino;           /* inode no pointed to by this entry */
    __le64 size;          /* Directory size */
    char name[256];       /* File name */
} __attribute((__packed__));
```

Figure 3.7: **NOVA directory entry.** NOVA dentry provides mapping from file name to inode number.

Fig 3.7 shows the NOVA dentry struct. When application opens a file, the file system looks up the filename in the parent directory, and uses the inode number to locate the target file. The filename is stored in the `name` array, and `ino` represents the corresponding inode number. `mtime` and `size` represents the parent directory's modification time and size, so that when a dentry is appended or deleted, the attributes of the parent directory are updated atomically. NOVA appends a dentry to the log when it creates, deletes, or renames a file or subdirectory under that directory. A dentry for a `delete` operation has its inode number set to zero to distinguish it from a create dentry.

```
/* nova.h */
struct nova_setattr_logentry {
    u8 entry_type;       /* Log entry type */
    u8 attr;             /* Update attributes */
    __le16 mode;         /* Inode mode */
    __le32 uid;          /* Inode uid */
    __le32 gid;          /* Inode gid */
    __le32 atime;        /* Inode access time */
    __le32 mtime;        /* Inode write time */
    __le32 ctime;        /* Inode modification time */
    __le64 size;         /* Inode file size */
    u8 invalid;          /* Log invalid now? */
    u8 paddings[3];
    __le32 csum;
} __attribute((__packed__));
```

Figure 3.8: **NOVA inode update entry.** NOVA appends inode update entries to the inode log when the state of the inode is changed, e.g. `chmod`.

Fig 3.8 shows the structure of inode update entry. NOVA adds inode update entries to the directory's log to record updates to the directory's inode (e.g., for `chmod` and `chown`). These operations modify multiple fields of the inode, and the inode update entry provides atomicity withour explicit journaling. The fields in inode update entries have the same meanings as the corresponding fields in the inode structure.

To speed up dentry lookups, NOVA keeps a radix tree in DRAM for each directory inode. The key is the hash value of the dentry name, and each leaf node points to the corresponding dentry in the log. The radix tree makes search efficient even for large directories. The latest version of NOVA has replaced the radix tree with red-black tree to reduce memory consumption. Below, we show some examples of NOVA directory operations.

Creating a file Figure 3.9 illustrates the creation of file *zoo* in a directory that already contains file *bar*. The directory has recently undergone a `chmod` operation and used to contain another file, *foo*. The log entries for those operations are visible in the figure. NOVA first selects and

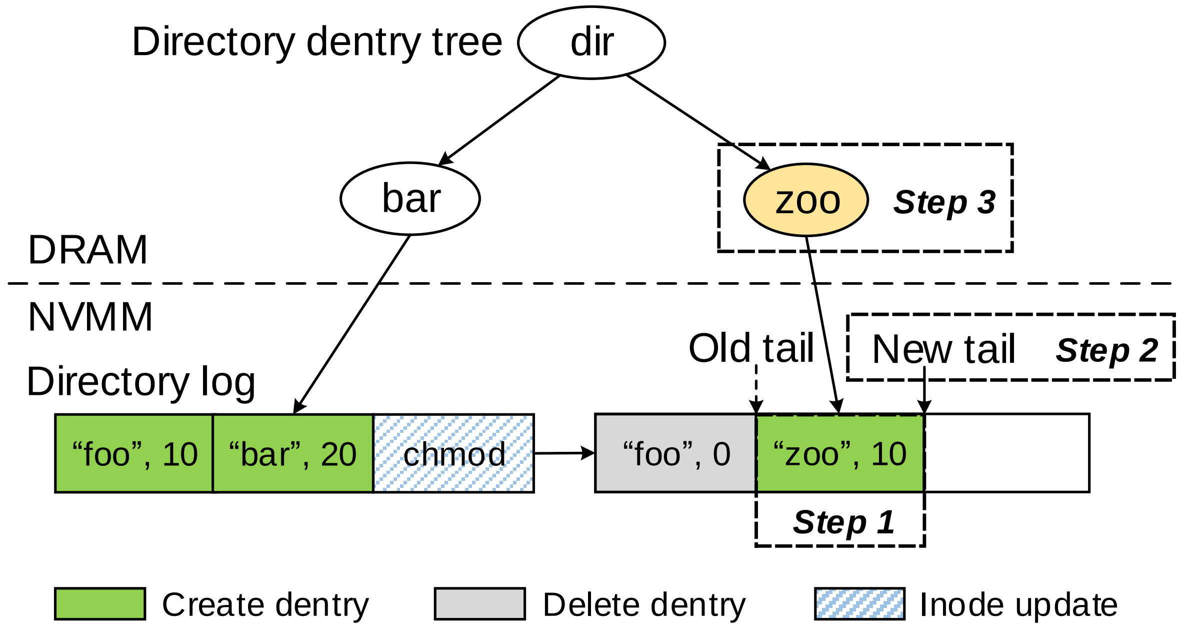

Figure 3.9: **NOVA directory operation.** Dentry is shown in <name, inode_number> format. To create a file, NOVA first appends the dentry to the directory's log (step 1), updates the log tail as part of a transaction (step 2), and updates the radix tree (step 3).

initializes an unused inode in the inode table for *zoo*, and appends a create dentry of *zoo* to the directory's log. Then, NOVA uses the current CPU's journal to atomically update the directory's log tail and set the valid bit of the new inode. Finally NOVA adds the file to the directory's radix tree in DRAM.

Deleting a file In Linux, deleting a file requires two updates: The first decrements the link count of the file's inode, and the second removes the file from the enclosing directory. NOVA first appends a delete dentry log entry to the directory inode's log and an inode update entry to the file inode's log and then uses the journaling mechanism to atomically update both log tails. Finally it propagates the changes to the directory's radix tree in DRAM.

Move/rename a file Linux uses `move` command to perform file/directory move and rename operations. `Move` is a complex operation because based on the parameters, it may perform move, rename or overwrite operations. A `move` operation such as

```
mv dir1/foo dir2/bar
```

may involve updates of up to four inodes: *foo*, *bar*(if exists), *dir1* and *dir2*(if different from *dir1*). To make the operation atomic, NOVA must update all these logs atomically. Assuming *bar* exists and *dir2* different from *dir1*, NOVA performs the following updates to the logs:

1) Append an inode update entry to the log of *foo* to update ctime;

2) Append an inode update entry to the log of *bar* to decrease the link count and update ctime; if the link count becomes zero after the operation, the valid field of *bar* is also journaled;

3) Append a delete *foo* dentry to the log of *dir1*;

4) Append a create *bar* dentry with the inode number of *foo* to the log of *dir2*.

5) Copy the inode of *foo* to the log of *dir2*. This step is necessary because all the inodes should reside on the parent inode's log.

After all the log updates complete, NOVA uses journaling to make all the log tail updates atomic, and updates the valid field of *bar* if needed. Finally NOVA updates the dentry radix trees of *dir1* and *dir2* to reflect the changes.

3.2.4 Atomic file operations

`Read` and `write` are the most common file operations and play an important part in file system performance. Both of them modify inode metadata as well. The NOVA file structure uses logging to provide metadata and data atomicity with low overhead, and it uses copy-on-write for file data to reduce the log size and make garbage collection simple and efficient. NOVA is the first NVMM file system that achieves atomic file writes.

NOVA performs writes by appending file write entries to the inode log. Fig 3.10 shows the file write entry in NOVA. A file write entry describes a file write operation and points to a contiguous region of data blocks. `pgoff` field represents the start page offset of this entry, `num_pages` represents the initial length of the extent, and `block` stores the address of the data blocks. Inode attributes such as `mtime` and `size` are also embeded in the entry. NOVA uses

```
/* nova.h */
struct nova_file_write_entry {
    u8 entry_type;              /* Log entry type */
    u8 reassigned;              /* Data is not latest */
    u8 updating;                /* Updating entry */
    u8 padding;
    __le32 num_pages;           /* Num of data pages */
    __le64 block;               /* Start block address of data pages */
    __le64 pgoff;               /* Page offset */
    __le32 invalid_pages;       /* For GC */
    __le32 mtime;               /* For both ctime and mtime */
    __le64 size;                /* File size */
    __le32 csumpadding;
    __le32 csum;
} __attribute((__packed__));
```

Figure 3.10: **NOVA file write entry structure.** A NOVA file write entry describes a write operation. It includes the metadata of the write, and points to data pages outside the log.

copy-on-write for file data and reclaim stale data pages immediately. `invalid_pages` represents how many pages have been overwritten and reassigned. When `invalid_pages` become equal to `num_pages`, the file write entry is dead and can be garbage collected. When NOVA commits a file write entry by updating the log tail pointer, both metadata and data are committed atomically.

If the `write` is large, NOVA may not be able to describe it with a single write entry. If NOVA cannot find a large enough set of contiguous pages, it breaks the `write` into multiple write entries and appends them all to the log to satisfy the request. To maintain atomicity, NOVA commits all the entries with a single update to the log tail pointer.

Figure 3.11 shows the structure of a NOVA file. The file inode's log records metadata changes, data pages are resided outside the log, and each file has a radix tree in DRAM to locate data in the file by the file offset. The notation <*file pgoff*, *num pages*> denotes the page offset and number of pages a `write` affects. The first two entries in the log describe two `write`s, <0, 1> and <1, 2>, of 4 KB and 8 KB (i.e., 1 and 2 pages), respectively. A third, 8 KB `write`, <2, 2>, is in flight.

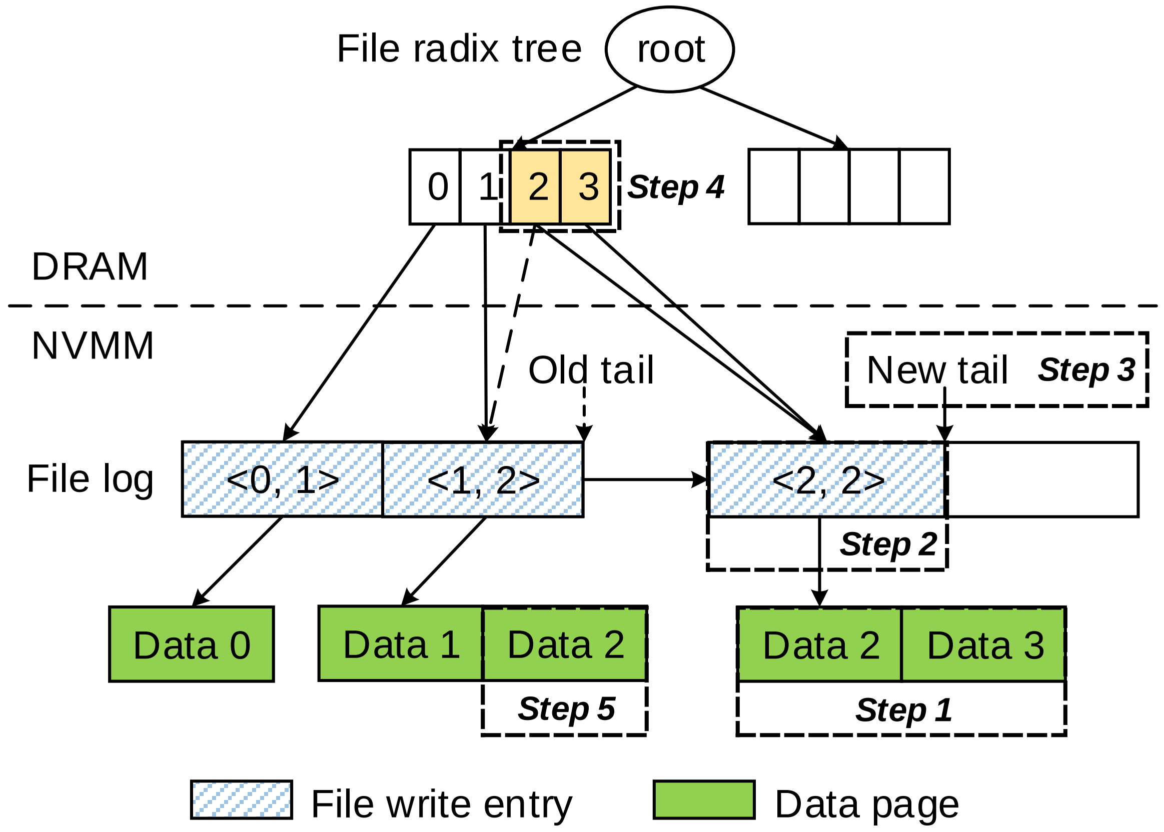

Figure 3.11: **NOVA file structure and write operation.** An 8 KB (i.e., 2-page) write to page two (<2, 2>) of a file requires five steps. NOVA first writes a copy of the data to new pages (step 1) and appends the file write entry (step 2). Then it updates the log tail (step 3) and the radix tree (step 4). Finally, NOVA returns the old version of the data to the allocator (step 5).

To perform the <2, 2> `write`, NOVA fills data pages and then appends the <2, 2> entry to the file's inode log. Then NOVA atomically updates the log tail to commit the write, and updates the radix tree in DRAM, so that offset "2" points to the new entry. The NVMM page that holds the old contents of page 2 returns to the free list immediately. During the operation, a per-inode lock protects the log and the radix tree from concurrent updates. When the `write` system call returns, all the updates are persistent in NVMM.

NOVA performs copy-on-write on page size granularity, so a single byte write may result in 4 KB write to NVMM. To resolve the issue, NOVA supports in-place writes to trade data atomicity for better performance. If the application performs a overwrite in in-place write mode, NOVA locates the file write entry, copies the data to the existing data pages directly, and updates the file write entry with lightweight journaling.

For a `read` operation, NOVA updates the file inode's access time with a 64-bit atomic write, locates the required page using the file's radix tree, and copies the data from NVMM to the user buffer.

3.2.5 DAX mmap

DAX file systems allow applications to access NVMM directly via load and store instructions by mapping the physical NVMM file data pages into the application's address space. This *DAX-mmap* exposes the NVMM's raw performance to the applications and is likely to be a critical interface in the future.

NOVA supports DAX-mmap. When an application issues a `mmap` request and accesses the page, a page fault is triggered and NOVA handles the page fault by looking up the target page and adding it to the application's page table, so that applications can load/store the NVMM page directly.

When a page is `mmap`ed, copy-on-write is disabled, since a new `write` cannot reclaim the old page that is mapped. Instead, NOVA converts the `write` to an in-place `write` and writes to the page directly. This compromises data atomicity, but the key of DAX-mmap is to sacrifice data atomicity for performance.

3.2.6 Garbage collection

Garbage collection efficiency is critical to LFS's performance. NOVA's logs are linked lists and contain only metadata, making garbage collection simple and efficient. By putting data outside the log, NOVA is free from the need to constantly move data to maintain a supply of contiguous free regions.

NOVA handles garbage collection for stale data pages and stale log entries separately. NOVA collects stale data pages immediately during `write` operations (see Section 3.2.4).

Cleaning inode logs is more complex. A log entry is dead in NOVA if it is not the last entry in the log (because the last entry records the inode's latest *ctime*) and any of the following conditions is met:

- A file write entry is dead, if it does not refer to valid data pages. i.e. following writes have completely overwrite its data pages.

- An inode update that modifies metadata (e.g., *mode* or *mtime*) is dead, if a later inode update modifies the same piece of metadata.

- A dentry update is dead, if a delete dentry is committed and it is marked invalid.

NOVA mark dentries invalid in certain cases. For instance, file creation adds a create dentry to the log. Deleting the file adds a delete dentry, and it also marks the create dentry as invalid. The garbage collector will then reclaim both create dentry and delete dentry. If the NOVA garbage collector reclaimed the delete dentry but left the create dentry, the file would seem to reappear.

These rules determine which log entries are alive and dead, and NOVA uses two different garbage collection (GC) techniques to reclaim dead entries. NOVA performs log cleaning when the log is full. As the log is a linked list consists of 4 KB pages, log cleaning is performed at fine-grained granularity.

Fast GC Fast GC emphasizes speed over thoroughness and it does not require any copying. NOVA uses it to quickly reclaim space when it extends an inode's log. If all the entries in a log page are dead, fast GC reclaims it by deleting the page from the log's linked list. Figure 3.12(a) shows an example of fast log garbage collection. Originally the log has four pages and page 2 contains only dead log entries. NOVA atomically updates the next page pointer of page 1 to point

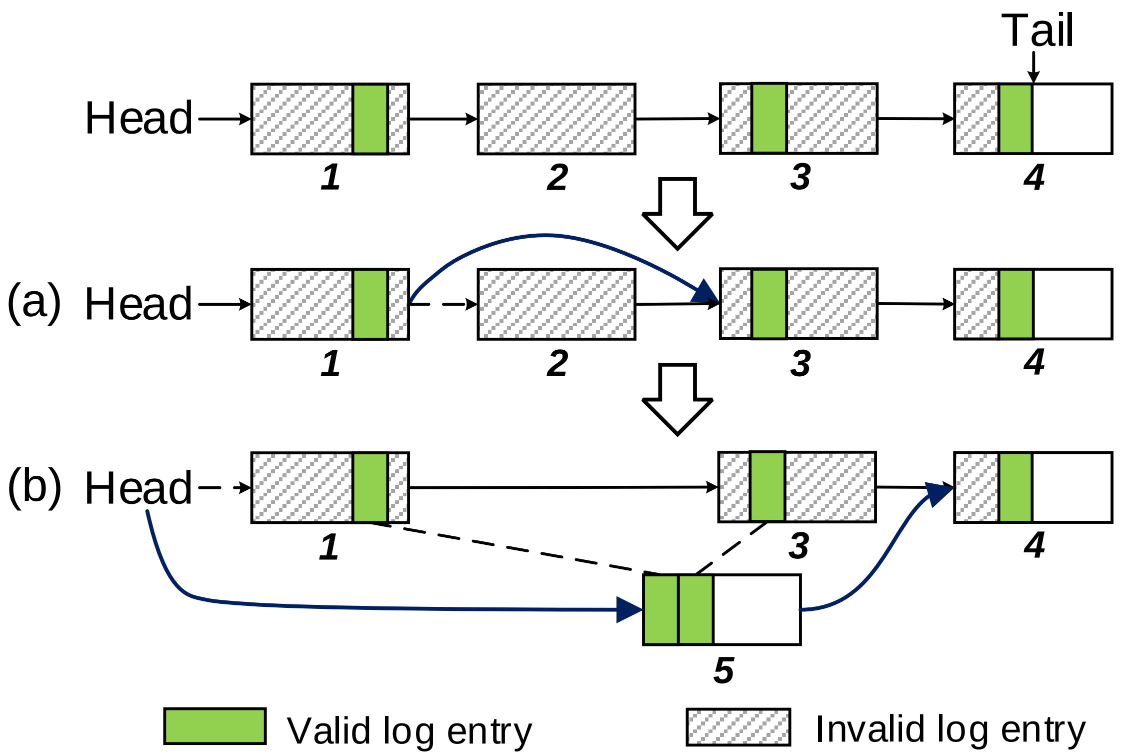

Figure 3.12: **NOVA log cleaning.** The linked list structure of log provides simple and efficient garbage collection. Fast GC reclaims invalid log pages by deleting them from the linked list (a), while thorough GC copies live log entries to a new version of the log (b).

to page 3 and frees page 2. Fast GC does not involve any metadata or data copy and is highly efficient.

Thorough GC During the fast GC log scan, NOVA tallies the space that live log entries occupy. If the live entries account for less than 50% of the log space, NOVA applies thorough GC after fast GC finishes, copies live entries into a new, compacted version of the log, updates the DRAM data structure to point to the new log, then atomically replaces the old log with the new one, and finally reclaims the old log.

Figure 3.12(b) illustrates thorough GC after fast GC is complete. NOVA allocates a new log page 5, and copies valid log entries in page 1 and 3 into it. Then, NOVA links page 5 to page 4 to create a new log and replace the old one. NOVA does not copy the live entries in page 4 to avoid updating the log tail, so that NOVA can atomically replace the old log by updating the log head pointer.

All the garbage collections are performed atomically, so that file system consistency is always guaranteed, no matter which kind of garbage collection is running when the system fails.

3.2.7 Shutdown and Recovery

NOVA puts a host of auxiliary data structures such as allocator and per-inode radix trees in DRAM, because these data structures are complex and have high consistency overhead if put in NVMM. Even though NOVA has to rebuild these data structures on remount, since NVMM has low latency and high bandwidth and supports fast concurrent random accesses, scanning NVMM in parallel on mount is acceptable.

When NOVA mounts the file system, it reconstructs the in-DRAM data structures it needs. Since applications may access only a portion of the inodes while the file system is running, NOVA adopts a policy called *lazy rebuild* to reduce the recovery time: It postpones rebuilding the radix tree and the inode until the system accesses the inode for the first time. This policy accelerates the recovery process and reduces DRAM consumption. As a result, during remount NOVA only needs to reconstruct the NVMM allocator. NOVA uses different algorithms for clean shutdowns and system failures.

Recovery after a normal shutdown On a clean unmount, NOVA stores the NVMM page allocator state in a reserved recovery inode's log and restores the allocator during the subsequent remount. Since NOVA does not scan any inode logs in this case, the recovery process is very fast: Our measurement shows that NOVA can remount a 50 GB file system in 1.2 milliseconds.

Recovery after a failure In case of a unclean dismount (e.g., system crash), NOVA must rebuild the NVMM allocator information by scanning the inode logs. NOVA log scanning is fast because of two design decisions. First, per-CPU inode tables and per-inode logs allow for vast parallelism in log recovery. Second, NOVA logs only contain metadata and they tend to be short. The number of live log entries in an inode log is roughly the number of extents in the file. As a result, NOVA only needs to scan a small fraction of the NVMM during recovery. The NOVA failure recovery consists of two steps:

First, NOVA checks each journal and rolls back any uncommitted transactions to restore the file system to a consistent state.

Second, NOVA starts a recovery thread on each CPU and scans the inode tables in parallel, performing log scanning for every valid inode in the inode table. NOVA use different recovery mechanisms for directory inodes and file inodes: For a directory inode, NOVA scans the log's linked list to enumerate the pages it occupies, but it does not inspect the log's contents. For a file inode, NOVA reads the write entries in the log to enumerate the data pages. NOVA does not read data page contents during recovery.

During the recovery scan NOVA builds a bitmap of occupied pages, and rebuilds the allocator based on the result. After this process completes, the file system is ready to accept new requests.

3.3 Evaluation

In this section we evaluate the performance of NOVA and answer the following questions:

- How does NOVA perform against state-of-the-art file systems built for disks, SSDs, and NVMM?
- What kind of operations benefit most from NOVA?
- How do underlying NVMM characteristics affect NOVA performance?
- How does NOVA scale on many-core, multi-socket platform?
- How efficient is NOVA garbage collection compared to other approaches?

- How expensive is NOVA recovery?

We evaluate NOVA and compare to other state-of-the-art NVMM file systems with a host of micro- and macro-benchmarks on a NVMM hardware emulation platform. Below we first describe the experimental setup, and then show the results of our testing.

3.3.1 Experimental setup

Different NVMM technologies have different characteristics. To emulate different types of NVMM and study their effects on NVMM file systems, we use the Intel Persistent Memory Emulation Platform (PMEP) [41]. PMEP is a dual-socket Intel Xeon processor-based platform with special CPU microcode and firmware. The processors on PMEP run at 2.6 GHz with 8 cores and 4 DDR3 channels. The BIOS marks the DRAM memory on channels 2 and 3 as emulated NVMM. PMEP supports configurable latencies and bandwidth for the emulated NVMM, allowing us to explore NOVA's performance on a variety of future memory technologies. PMEP emulates `clflushopt` and `clwb` instructions with processor microcode.

Table 3.1: **NVMM emulation characteristics.** STT-RAM emulates fast NVMs that have access latency and bandwidth close to DRAM, and PCM emulates NVMs that are slower than DRAM.

NVMM	Read latency	Write bandwidth	`clwb` latency
STT-RAM	100 ns	Full DRAM	40 ns
PCM	300 ns	1/8 DRAM	40 ns

In our tests we configure the PMEP with 32 GB of DRAM and 64 GB of NVMM. To emulate different NVMM technologies, we choose two configurations for PMEP's memory emulation system (Table 3.1): For STT-RAM we use the same read latency and bandwidth as DRAM; For PCM we use 300 ns for the read latency and reduce the write bandwidth to 1/8th of DRAM. `clwb` takes 500 ns for both settings.

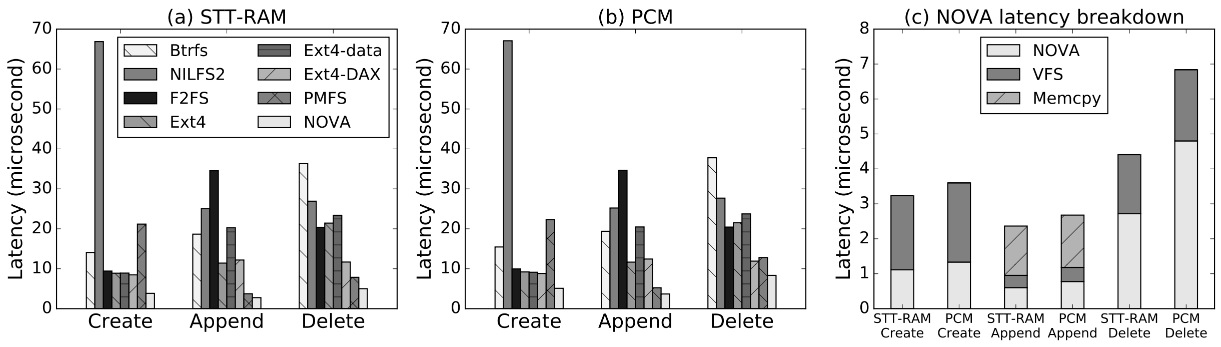

Figure 3.13: **NOVA file operation latency.** The single-thread benchmark performs create, append and delete operations on a large number of files.

We evaluate NOVA on Linux kernel 4.0 against seven file systems: PMFS and ext4-DAX are two open source NVMM file systems. PMFS is a native file system designed for NVMM, and ext4-DAX enhances ext4 with DAX support. Both of them journal metadata and perform in-place updates for file data. NILFS2 and F2FS are log-structured file systems designed for HDD and flash-based storage, respectively. We also compare to ext4 in default mode (ext4) and in data journal mode (ext4-data) which provides data atomicity. Finally, we compare to Btrfs [1], a state-of-the-art copy-on-write Linux file system. Except for ext4-DAX and ext4-data, all the file systems are mounted with default options. Btrfs and ext4-data are the only two file systems in the group that provide the same, strong consistency guarantees as NOVA.

PMFS manages NVMM directly and do not require a block device interface. For the other file systems, we use the Intel persistent memory driver [96] to emulate NVMM-based ramdisk-like device. We add `clwb` instructions to flush data where necessary in each file system.

3.3.2 Microbenchmarks

We use a single-thread micro-benchmark to evaluate the latency of basic file system operations. The benchmark creates 10,000 files, makes sixteen 4 KB appends to each file, calls `fsync` to persist the files, and finally deletes them.

Figures 3.13(a) and 3.13(b) show the results on STT-RAM and PCM, respectively. The

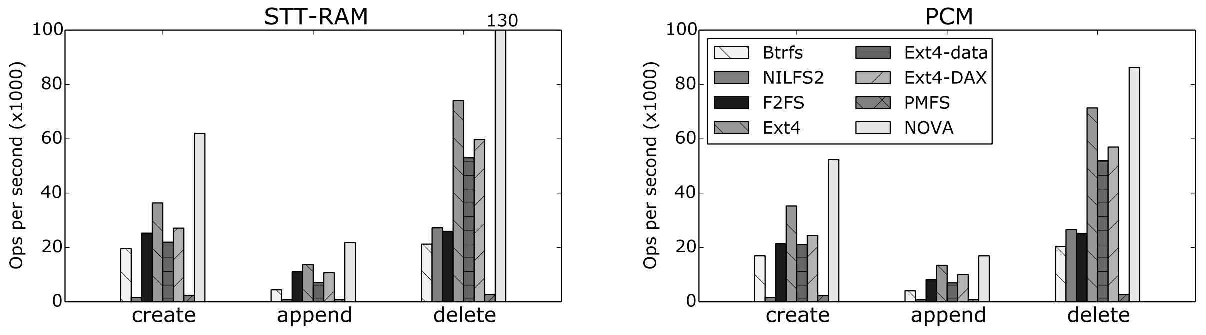

Figure 3.14: **PostMark throughput.**

latency of `fsync` is amortized across the `append` operations. NOVA provides the lowest latency for each operation, outperforms other file systems by between 35% and 17×, and improves the `append` performance by 7.3× and 6.7× compared to Ext4-data and Btrfs respectively. PMFS is closest to NOVA in terms of `append` and `delete` performance.

NOVA is more sensitive to NVMM performance than the other file systems because NOVA's software overheads are lower, and so overall performance more directly reflects the underlying memory performance. Figure 3.13(c) shows the latency breakdown of NOVA file operations on STT-RAM and PCM. For `create` and `append` operations, NOVA only accounts for 21%–28% of the total latency. On PCM the NOVA `delete` latency increases by 76% because NOVA reads the inode log to free data and log blocks and PCM has higher read latency. For the `create` operation, the VFS layer accounts for 49% of the latency on average. The memory copy from the user buffer to NVMM consumes 51% of the `append` execution time on STT-RAM, suggesting that the POSIX interface may be the performance bottleneck on high speed memory devices.

We also use PostMark [59] to evaluate basic file system operation performance. The benchmark creates 100,000 files with random size between 2 KB and 9 KB, performs random append operations to them, and then deletes them.

Figure 3.14 shows the results on STT-RAM and PCM. NOVA provides the highest throughput, outperforms other file systems by between 26 and 75%, and improves performance

Table 3.2: **Filebench workload characteristics.** The selected four workloads have different read/write ratios and access patterns.

Workload	Average file size	I/O size (r/w)	Threads	R/W ratio	# of files Small/Large
Fileserver	128 KB	16 KB/16 KB	50	1:2	100K/400K
Webproxy	32 KB	1 MB/16 KB	50	5:1	100K/1M
Webserver	64 KB	1 MB/8 KB	50	10:1	100K/500K
Varmail	32 KB	1 MB/16 KB	50	1:1	100K/1M

by up to 3× comparing to Ext4-data.

Plain Ext4 is closest to NOVA in terms of performance and narrows the gap further on PCM, because slower NVMM enhances the benefits of the DRAM page cache. PMFS performs poorly in this test for two reasons: first, PMFS allocates NVMM pages from a single free list, limiting performance under frequent allocation requests. Second, PMFS performs directory lookup by linearly searching the directory entries, so it does not scale well with large directories. NILFS2 also performs poorly in create and append operations, suggesting that naively using log-structured, disk-oriented file systems on NVMM is unwise.

3.3.3 Macrobenchmarks

We select four Filebench [2] workloads—fileserver, webproxy, webserver and varmail—to evaluate the application-level performance of NOVA. Table 3.2 summarizes the characteristics of the workloads. For each workload we test two dataset sizes by changing the number of files. The *small* dataset will fit entirely in DRAM, allowing file systems that use the DRAM page cache to cache the entire dataset. The *large* dataset is too large to fit in DRAM, so the page cache is less useful. We run each test five times and report the average. Figure 3.15 shows the Filebench throughput with different NVMM technologies and data set sizes.

In the fileserver workload, NOVA outperforms other file systems by between 1.8×

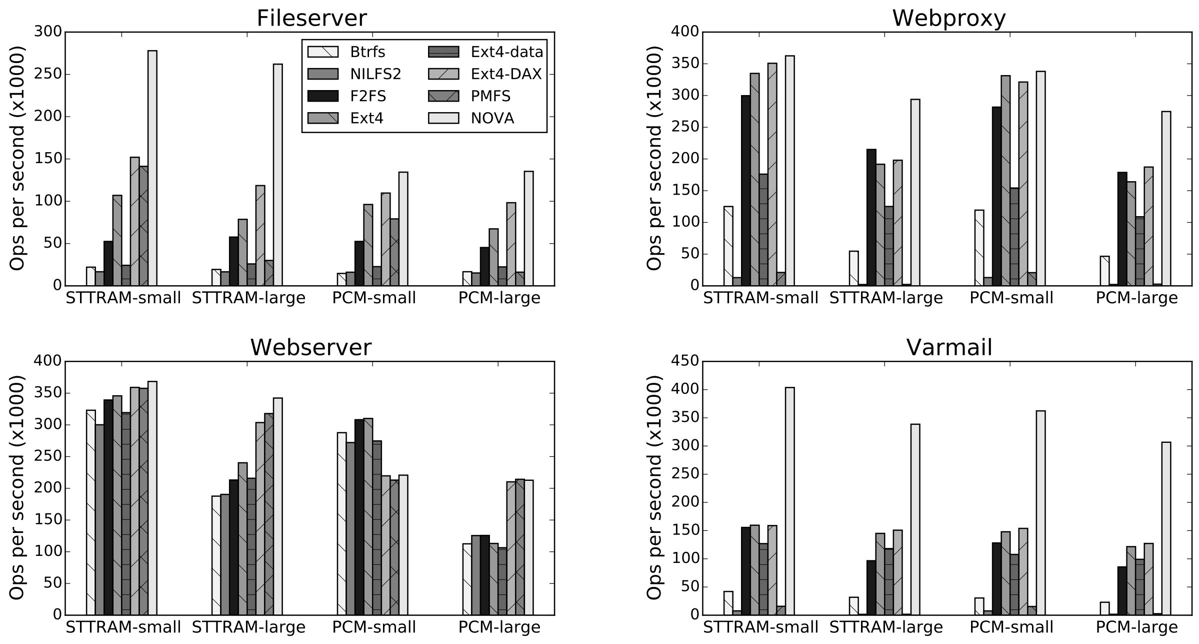

Figure 3.15: **Filebench throughput with different file system patterns and dataset sizes on STT-RAM and PCM.** Each workload has two dataset sizes so that the small one can fit in DRAM entirely while the large one cannot. The standard deviation is less than 5% of the value.

and 16.6× on STT-RAM, and between 22% and 9.1× on PCM for the large dataset. NOVA outperforms Ext4-data by 11.4× and Btrfs by 13.5× on STT-RAM, while providing the same consistency guarantees. NOVA on STT-RAM delivers twice the throughput compared to PCM, because of PCM's lower write bandwidth. PMFS performance drops by 80% between the small and large datasets, indicating its poor scalability. NILFS2 could not complete the fileserver workload more than once because its garbage collection is inefficient, so we report the value for the single completed run.

Webproxy is a read-intensive workload. For the small dataset, NOVA performs similarly to Ext4 and Ext4-DAX, and 2.1× faster than Ext4-data. For the large workload, NOVA performs between 36% and 53% better than F2FS and Ext4-DAX. PMFS performs directory lookup by linearly searching the directory entries, and NILFS2's directory lock design is not scalable [107], so their performance suffers since webproxy puts all the test files in one large directory. Significant work and rearchitecture would be required to overcome the limitations of both file systems.

Webserver is a read-dominated workload and does not involve any directory operations. As a result, non-DAX file systems benefit significantly from the DRAM page cache and the workload size has a large impact on performance. Since STT-RAM has the same latency as DRAM, small workload performance is roughly the same for all the file systems with NOVA enjoying a small advantage. On the large data set, NOVA performs 10% better on average than Ext4-DAX and PMFS, and 63% better on average than non-DAX file systems. On PCM, NOVA's performance is about the same as the other DAX file systems. For the small dataset, non-DAX file systems are 33% faster on average due to DRAM caching. However, for the large dataset, NOVA's performance remains stable while non-DAX performance drops by 60%.

Varmail emulates an email server with a large number of small files and involves both `read` and `write` operations. NOVA outperforms Btrfs by 11.1× and Ext4-data by 3.1× on average, and outperforms the other file systems by between 2.2× and 216×, demonstrating its capabilities in write-intensive workloads and its good scalability with large directories. NILFS2 and PMFS still suffer from poor directory operation performance.

Overall, NOVA achieves the best performance in almost all cases and provides data consistency guarantees that are as strong or stronger than the other file systems. The performance advantages of NOVA are largest on write-intensive workloads with large number of files. NOVA outperforms other file systems because it performs fine-grained logging for metadata updates and designed with scalability on large directories.

3.3.4 Scalability

NOVA is designed with scalability in mind. It does not have any global locks, partitions the allocator, inode table and journal among all the CPUs. NOVA is targeted to resolve the scalability bottleneck of conventional file systems.

To evaluate the scalability of NOVA, We perform the FxMark filesystem scalability test suite [81] on a dual-socket, 80-core machine. In the tests, each thread performs `create`, `unlink`,

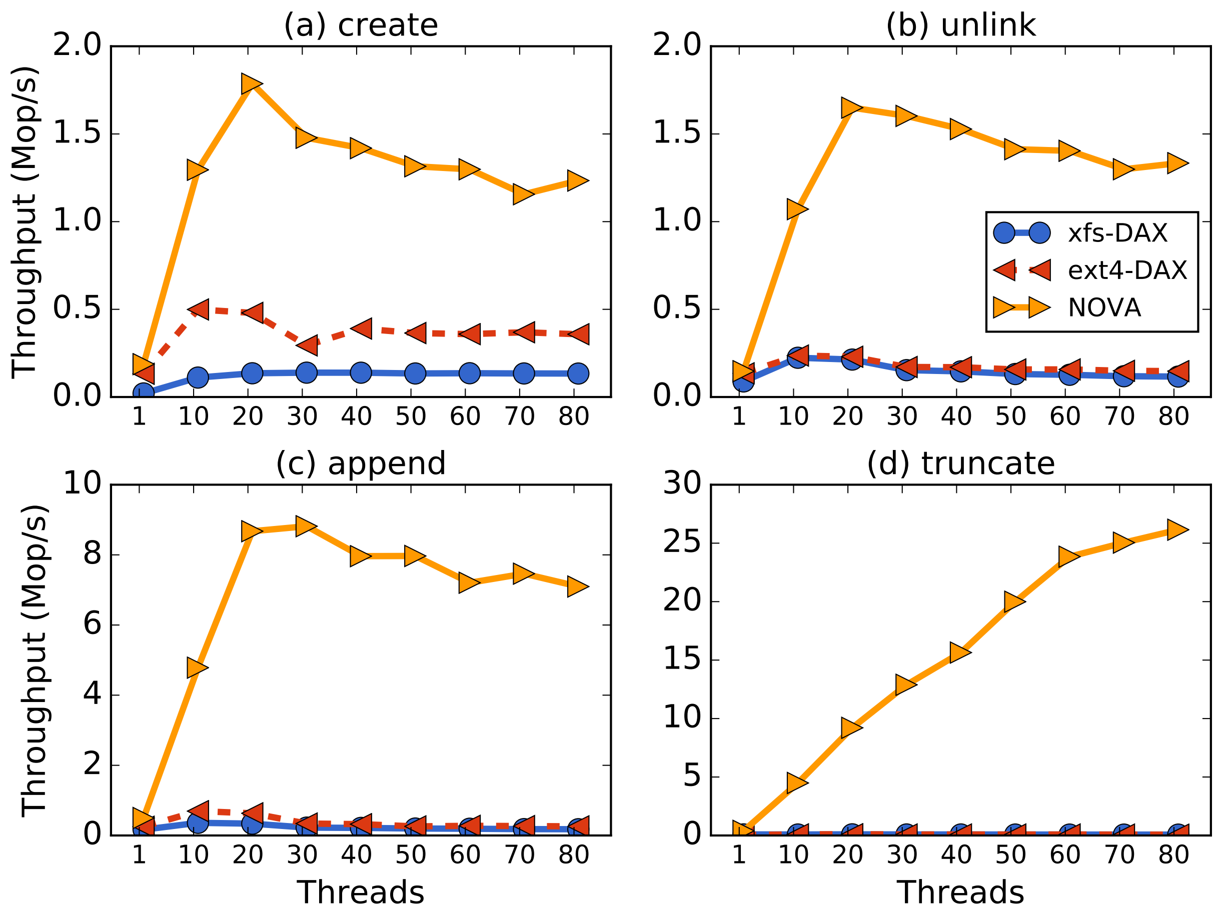

Figure 3.16: **NOVA scalability on file operations.** NOVA has good scalability comparing to other DAX file systems. For `create` and `unlink` NOVA is limited by VFS bottleneck, and for `append` NOVA saturates the persistent memory device bandwidth.

`append` and `truncate` on private directory or file, and we score the aggregated throughput. We run the test with fast STT-RAM emulation and compare to ext4-DAX and xfs-DAX.

Figure 3.16 shows the result. NOVA achieves much higher bandwidth than ext4-DAX and xfs-DAX: 3.4× on `create`, 9× on `unlink`, 27× on `append` and 358× on `truncate`, respectively. Ext4-DAX and xfs-DAX cannot scale because they use a single journal and all the transcations contend for the same journal. NOVA eliminates global locks and uses per-CPU journal to support concurrent transactions, scaling much better on many-core platforms.

For `create` and `unlink`, NOVA scales to twenty cores, due to the scalability bottleneck in VFS layer. The persistent memory device is attached to NUMA node 0, and for `append`

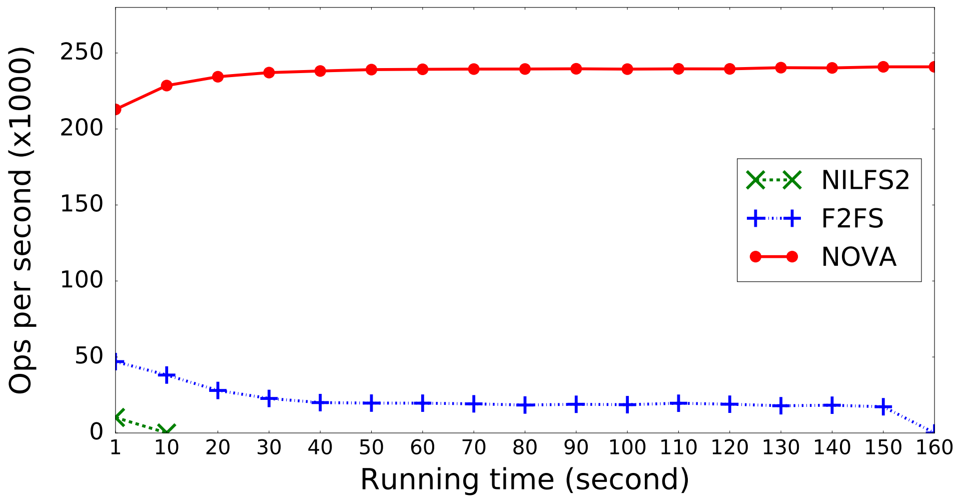

Figure 3.17: **NOVA garbage collection efficiency.** The test runs a 30 GB fileserver workload under 95% NVMM utilization with different durations.

NOVA saturates the memory bandwidth after twenty threads. We will further investigate these inefficiencies and NUMA impact in Section 5.2.

3.3.5 Garbage collection efficiency

NOVA resolves the garbage collection performance issue that many LFSs suffer from, i.e. they have performance problems under heavy write loads, especially when the file system is nearly full. NOVA reduces the log cleaning overhead by reclaiming stale data pages immediately, keeping log sizes small, and making garbage collection of those logs efficient.

To evaluate the efficiency of NOVA garbage collection when NVMM is scarce, we run a 30 GB write-intensive fileserver workload under 95% NVMM utilization for different durations, and compare with the other log-structured file systems, NILFS2 and F2FS. We run the test with PMEP configured to emulate STT-RAM.

Figure 3.17 shows the result. NILFS2 could not finish the 10-second test due to garbage collection inefficiencies. F2FS fails after running for 158 seconds, and the throughput drops by 52% between the initial state and stable state due to log cleaning overhead. In contrast, NOVA

Table 3.3: **NOVA GC pages statistics.** The table shows the number of pages that NOVA garbage collector reclaimed in the test.

Duration	10s	30s	120s	600s	3600s
Fast GC	0	255	17,385	159,406	1,170,611
Thorough GC	102	2,120	9,633	27,292	72,727

outperforms F2FS by 12× and successfully runs for the full hour. NOVA's throughput also remains stable, dropping by less than 8% between the 10s and one-hour tests.

We also count the number of pages that NOVA garbage collector reclaimed and show the result in table 3.3. On the 30s test fast GC reclaims 11% of the stale log pages. With running time rises, fast GC becomes more efficient and is responsible for 94% of reclaimed pages in the one-hour test. The result shows that in long-term running, the simple and low-overhead fast GC is efficient enough to reclaim the majority of stale log pages.

3.3.6 Recovery overhead

NOVA uses DRAM to maintain the NVMM free page lists that it must rebuild when it mounts a file system. NOVA accelerates the recovery by rebuilding inode information lazily, keeping the logs short, and performing log scanning in parallel.

To measure the recovery overhead, we use the three workloads in Table 3.4. Each workload represents a different use case for the file systems: Videoserver contains a few large files accessed with large-size requests, mailserver includes a large number of small files and the request size is small, fileserver is in between. For each workload, we measure the cost of mounting after a normal shutdown and after a power failure.

Table 3.5 summarizes the results. With a normal shutdown, NOVA recovers the file system in 1.2 ms, as NOVA does not need to scan the inode logs. After a power failure, NOVA recovery time increases with the number of inodes (because the number of logs increases) and as the I/O

Table 3.4: **Recovery workload characteristics.** The number of files and typical I/O size both affect NOVA's recovery performance.

Dataset	File size	Number of files	Dataset size	I/O size
Videoserver	128 MB	400	50 GB	1 MB
Fileserver	1 MB	50,000	50 GB	64 KB
Mailserver	128 KB	400,000	50 GB	16 KB

Table 3.5: **NOVA recovery time on different scenarios.** NOVA is able to recover 50 GB data in 116ms in case of power failure.

Dataset	Videoserver	Fileserver	Mailserver
STTRAM-normal	156 μs	313 μs	918 μs
PCM-normal	311 μs	660 μs	1197 μs
STTRAM-failure	37 ms	39 ms	72 ms
PCM-failure	43 ms	50 ms	116 ms

operations that created the files become smaller (because file logs become longer as files become fragmented). Recovery runs faster on STT-RAM than on PCM because NOVA reads the logs to reconstruct the NVMM free page lists, and PCM has higher read latency than STT-RAM. On both PCM and STT-RAM, NOVA is able to recover 50 GB data in 116ms, achieving failure recovery bandwidth higher than 400 GB/s.

3.4 Summary

Emerging NVMM technology will change the storage stack fundamentally, and new file systems are required to fully exploit the performance potential. We have implemented and described NOVA, a log-structured file system designed for hybrid volatile/non-volatile main memories. NOVA extends ideas of LFS to leverage NVMM, yielding a simpler, high-performance file system that supports fast and efficient garbage collection and quick recovery from system failures. Our measurements show that NOVA outperforms existing NVMM file systems by a wide margin on a wide range of applications while providing stronger consistency and atomicity guarantees.

Acknowledgments

This chapter contains material from “NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories”, by Jian Xu and Steven Swanson, which appears in the 12th USENIX Conference on File and Storage Technologies (FAST 2016). The dissertation author is the primary investigator and first author of this paper.

The author also would like to thank John Ousterhout, Niraj Tolia, Isabella Furth and the FAST anonymous reviewers for their insightful comments and suggestions. The author is also thankful to Subramanya R. Dulloor from Intel for his support and hardware access.

Chapter 4

NOVA-Fortis: Making Persistent Memory File System Fault-tolerant

We have designed and implemented NOVA, a NVMM file system that provides high performance and strong consistency guarantees. Despite the NVMM-centric performance improvements, neither NOVA nor existing NVMM file systems provide the data protection features necessary to detect and correct media errors, protect against data corruption due to misbehaving code, or perform consistent backups of the NVMM's contents. Enterprise file systems such as Btrfs and ZFS provide some or all of these capabilities for block-based storage. If users are to trust NVMM file systems with critical data, they will need these features as well.

As NVMM has very different characteristics from block-based hard disks and SSDs, there are four key differences between conventional block-based file systems and NVMM file systems from a reliability perspective.

First, the memory controller reports persistent memory media errors as non-maskable interrupts rather than error codes from a block driver. Further, the granularity of errors is smaller (e.g., a cache line) and varies depending on the memory device.

Second, persistent memory file systems must support DAX-style memory mapping that maps persistent memory pages directly into the application's address space. DAX is the fastest way to access persistent memory since it eliminates all operating and file system code from the access path. However, it means a file's contents can change without the file system's knowledge, which is not possible in a block-based file system.

Third, the entire file system resides in the kernel's address space, vastly increasing vulnerability to "scribbles", i.e. errant stores from misbehaving kernel code.

Fourth, NVMMs are vastly faster than block-based storage devices. This means that the trade-offs block-based file systems make between reliability and performance need a thorough re-evaluation.

Based on NOVA, we explore the impact of these differences on file system reliability mechanisms and built *NOVA-Fortis* [138], an fault-tolerant NVMM file system. We implemented snapshots, replication, checksums, and RAID-4 parity protection in NOVA-Fortis. In applying these techniques to an NVMM file system, we have developed the principle of *caveat DAXor* ("let the DAXer beware"): Applications that use DAX-style `mmap()` must accept responsibility for protecting their data's integrity and consistency.

Protecting and guaranteeing consistency for DAX `mmap()`'d data is complex and challenging. The file system cannot fix that and should not try. Instead, the file system should studiously avoid imposing any performance overhead on DAX-style access, except when absolutely necessary. For data that is not mapped, the file system should retain responsibility for data integrity.

Caveat DAXor has two important consequences for NOVA-Fortis' design. The first applies to most other NVMM file systems: To maximize performance, applications are responsible for enforcing ordering on stores to mapped data to ensure consistency in the face of system failure.

The second consequence arises because NOVA-Fortis uses parity to protect file data from corruption. Keeping error correction information up-to-date for mapped data would require interposing on every store, imposing a significant performance overhead. Instead, NOVA-Fortis requires applications to take responsibility for data protection of data *while it is mapped* and restores parity protection when the memory is unmapped.

We quantify the performance and storage overhead of NOVA-Fortis' fault-tolerance mechanisms and these design decisions and evaluate their effectiveness at preventing corruption of both file system metadata and file data.

By developing and describing NOVA-Fortis, We make the following contributions:

1. We identify the unique challenges that the *caveat DAXor* principle presents to building a fault-tolerant NVMM file systems.

2. We identify a key challenge to taking consistent file system snapshots while using DAX-style `mmap()` and develop algorithm that resolves it.

3. We adapt state-of-the-art techniques for metadata and data protection to work in NOVA-Fortis and to accommodate DAX-style `mmap()`.

4. We quantify NOVA-Fortis' vulnerability to scribbles and develop techniques to reduce this vulnerability.

5. We quantify the performance and storage overheads of NOVA-Fortis' data protection mechanisms.

We find that the extra storage NOVA-Fortis needs to provide fault-tolerance consumes 14.8% of file system space and reduces application-level performance by between 2% and 38% compared to NOVA. NOVA-Fortis outperforms DAX-aware file systems without reliability features by $1.5\times$ on average. It outperforms reliable, block-based file systems running on NVMM by $3\times$ on average.

The remainder of this chapter is organized as follows. We first describe NOVA-Fortis' snapshot and (meta)data protection mechanisms (Sections 4.1 and 4.2). Section 4.3 evaluates these mechanisms, and Section 4.4 presents our conclusions.

4.1 Snapshots

NOVA-Fortis' snapshot support lets system administrators take consistent snapshots of the file system while applications are running. The system can mount a snapshot as a read-only file system or roll the file system back to a previous snapshot. NOVA-Fortis supports an unlimited number of snapshots, and snapshots can be deleted in any order. NOVA-Fortis is the first NVMM file system that supports taking consistent snapshots when applications modify file data via DAX `mmap()`.

Supporting snapshot in NVMM file systems is no trivial task. Snapshot file systems such as ZFS [21] and Btrfs [1] are designed for hard disks, and their high software overhead compromise the performance that NVMM can provide.

Implementating snapshot in NVMM file systems has three major challenges:

1. Performance. As NVMM file system expects to minimize the software overhead and exploits the fast NVMM, it has to minimize the time and space overhead of snapshots. Specifically, creating and deleting a snapshot should not block file system operations, and the overhead should not grow with file system size or number of files.

2. Space management. When appliaction deletes a snapshot, the space that the snapshot occupied needs to be reclaimed correctly and efficiently, otherwise it may cause memory leak or data corruption.

3. DAX-mmap. The current programming model of NVMM is to mmap the NVMM pages directly into user space, so that applications can load and store NVMM directly. This is the most efficient way to access NVMM, and adopted by NVM libraries [97]. However, DAX-mmap bypasses the operating system and does in-place updates, and NVMM file systems have no information about user space accesses. As a result, DAX-mmap is not compatible with snapshot, and none of the existing DAX file systems support snapshot.

Below, we described how NOVA-Fortis' snapshot mechanisms address all these challenges.

4.1.1 Taking and Restoring Snapshots

Applications expect efficient snapshot creation as well as high performance from NVMM file systems: Taking a snapshot should have low latency, incur minimal writes to NVMM, and not block file system write operations. None of the existing NVMM file systems meet all these requirements.

NOVA-Fortis implements snapshots by maintaining a global snapshot ID for the file system and storing the current snapshot ID in each log entry. Taking a snapshot increments the global snapshot ID and records the old snapshot ID in a list of valid snapshots. Creating a new snapshot in NOVA-Fortis does not block file system write operations, and if no files are `mmap()`'d it takes constant time regardless of the file system volume size.

To restore the file system to a snapshot, NOVA-Fortis mounts the file system read only. Then, to open a file, it traverses the log only while the log entrys' snapshot IDs are smaller than or equal to the target snapshot's ID.

Below we describe how the snapshot mechanism interacts with normal file operations. Then, we describe how NOVA-Fortis takes consistent snapshots while applications are using DAX-`mmap()`. Finally, we evaluate the impact of snapshots on NOVA-Fortis' resource consumption and on application performance.

4.1.2 Snapshot Management

To take a snapshot NOVA-Fortis must preserve file contents from previous snapshots while also being able to recover the space a snapshot occupied after its deletion. In this decription, we use "snapshot" to denote both the file system's state when a snapshot is taken and the set of file operations since the previous snapshot.

A key challenge in snapshot file system is how to manage space efficiently. When a snapshot is taken, file system needs to redirect new writes to its data blocks need to new blocks in a CoW way; when the snapshot is deleted, the file system needs to free the snapshot's unique data blocks to reclaim space.

```
/* snapshot.h */
/* Per-CPU DRAM log of updates to a snapshot. */
struct snapshot_list {
        struct mutex list_mutex;         /* Synchronization */
        unsigned long num_pages;         /* List length */
        unsigned long head;              /* List head */
        unsigned long tail;              /* List tail */
};

/* DRAM info about a snapshot manifest. */
struct snapshot_info {
        u64     epoch_id;                /* Snapshot ID */
        u64     timestamp;               /* Current timestamp */
        struct snapshot_list *lists;     /* Per-CPU snapshot list */
};
```

Figure 4.1: **NOVA-Fortis snapshot manifest structure.** NOVA-Fortis implements snapshot manifest as per-CPU logs to improve scalability.

NOVA-Fortis maintains a *snapshot manifest* for each snapshot. Each snapshot manifest is comprised of per-CPU snapshot list that contains snapshot entries, as shown in figure 4.1.

Entries in the manifest contain a pointer to a snapshot log entry. The snapshot log entry describes an inode or a contiguous region of data blocks, as shown in figure 4.2. The snapshot manifest log entry contains the delete snapshot id (i.e. the current snapshot id when the metadata is appended to the log). The snapshot snapshot id and the delete snapshot id in the snapshot manifest entry indicates the life of the deleted inode/data. The manifest for a snapshot ID contains entries for all the log entries that were *born* (i.e., created) with that snapshot ID and have since died (i.e., become stale because another write replaced them) in a *different* snapshot. When the file system deletes an inode or overwrites data blocks, it checks the snapshot id of the inode/file

```
/* snapshot.h */
/* Snapshot log entry for inode */
struct snapshot_inode_entry {
        u8      type;                   /* Entry type */
        u8      deleted;                /* Reclaimed? */
        u8      padding[6];
        u64     padding64;
        u64     nova_ino;               /* Deleted inode number */
        u64     delete_epoch_id;        /* Deleted snapshot ID */
} __attribute((__packed__));

/* Snapshot log entry for write operation */
struct snapshot_file_write_entry {
        u8      type;                   /* Entry type */
        u8      deleted;                /* Reclaimed? */
        u8      padding[6];
        u64     nvmm;                   /* Data page address */
        u64     num_pages;              /* Data length */
        u64     delete_epoch_id;        /* Deleted snapshot ID */
} __attribute((__packed__));
```

Figure 4.2: **NOVA-Fortis snapshot manifest entry.** NOVA-Fortis snapshot log entry contains information about deleted inode or range of reclaimed data pages.

write entry that refers the data blocks. If the snapshot id is not equal to the current snapshot id, the inode/data blocks belong to a old snapshot. NOVA-Fortis appends the metadata of the inode/data blocks to the snapshot manifest instead of reclaiming the space immediately.

If a snapshot is deleted, the manifest entries for that snapshot become part of the following snapshot. This can result in a snapshot containing multiple manifest entries for the same file data. In this case, it is safe to remove the older entry (i.e., those with earlier birth snapshot IDs).

Figure 4.3 illustrates how NOVA-Fortis manages snapshot manifests. In Figure 4.3 (a), an application issues two writes. In (b), NOVA-Fortis takes a snapshot, increments the snapshot ID (Step 1), and performs at write to location 0. The new write log entry has snapshot ID 1 (Step 2), since it was born in snapshot 1.

The write to location 0 in (b) kills the write to that location in (a). NOVA-Fortis checks

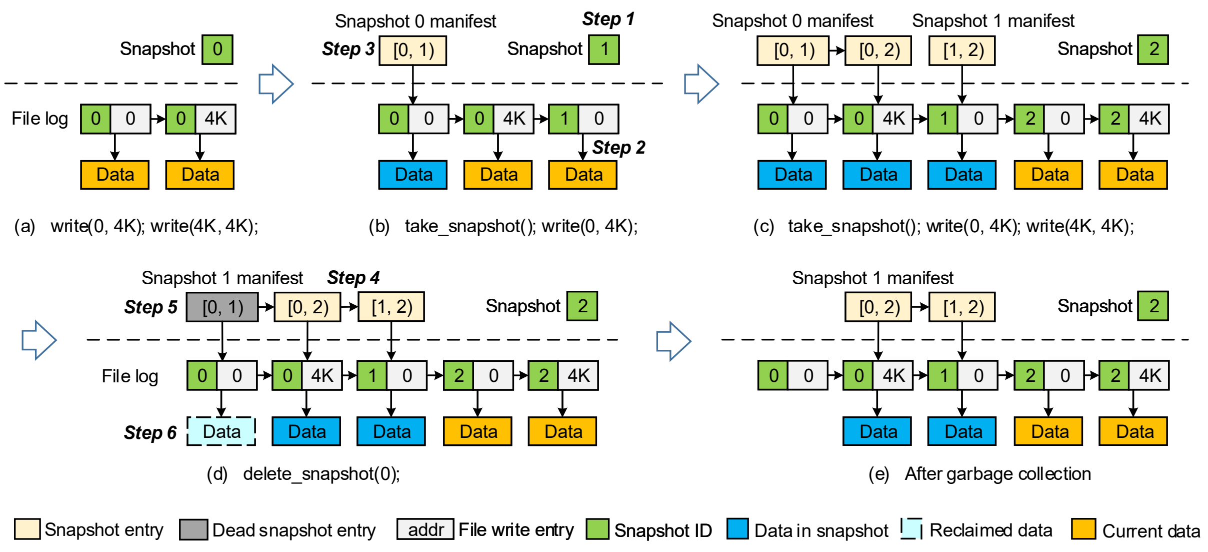

Figure 4.3: **Snapshots in NOVA-Fortis.** NOVA-Fortis stores the current snapshot ID in each log entry, and takes a snapshot by incrementing the snapshot ID. When a data block or log entry becomes dead in the current snapshot, NOVA-Fortis records its death in the snapshot manifest for most recent snapshot it survives in. To delete a snapshot, NOVA-Fortis adds the manifest for the deleted snapshot to the manifest for the next snapshot and removes redundant entries.

whether that log entry was born and died in different snapshots. In the example, it was born in snapshot 0 and died in snapshot 1, so NOVA-Fortis adds a entry to the snapshot 0 manifest (Step 3) and preserves the log entry. In Figure 4.3 (c), the application issues two writes that overwrite entries in snapshots 0 and 1, so NOVA-Fortis adds entries to those manifests.

In Figure 4.3 (d), NOVA-Fortis deletes snapshot 0 by removing it from the list of live snapshots and adding the manifest entries for snapshot 0 to snapshot 1's manifest (Step 4). In effect, this moves the start of snapshot 1 to the beginning of snapshot 0.

Then, NOVA-Fortis starts a background thread that compacts the combined manifest by discarding all the manifest entries that cover one part of the file except the most recent (Step 5). As it discards the manifest entries, it marks the corresponding log entries as dead and reclaims the associated file data (Step 6). Finally, (e) shows the file system state after manifest compaction is complete.

NOVA-Fortis keeps the list of live snapshots in NVMM, but it keeps the contents of the manifests in DRAM. On a clean shutdown, it writes the manifests to NVMM in the recovery inode. After an unclean shutdown, NOVA-Fortis can reconstruct the manifests while it scans the inode logs during recovery.

4.1.3 Snapshots for DAX `mmap()`'d Files

NOVA-Fortis monitors all the `mmap()` operations and records every read-write NVMM regions that mmaps to user space. When applications calls *mmap* with write permission, NOVA-Fortis adds the corresponding *vm_area_struct* (vma) struct which describes the mmaped pages to a linked list. When user takes a snapshot, NOVA-Fortis traverses the vma list, and changes the access permission from read-write to read-only. When applications stores to the page, a page fault is triggered (we call it *snapshot page fault*), and we modify the page fault handler to handover to NOVA-Fortis. NOVA-Fortis will perform a copy-on-write on the faulted page, update the page table entry to point to the CoW page, and changes the page access permission to writeable again, so that when the page fault handler returns, application continue stores to new pages.

The design has several advantages. First, applications do not need any modifications. Second, the virtual address does not change after the copy-on-write, and it is totally transparent to the applications. Third, copy-on-write is performed on-demand: if the application does not access or only load the page after the snapshot is taken, no copy-on-write is performed and application pays no overhead.

There are two challenges with the design. First, the OS page fault handler needs to tell a snapshot page fault from a real access violation error. To resolve the issue, we add a *original_write* flag to the vma struct to resolve the issue. NOVA-Fortis set the flag before changing the permission of a vma. If the page fault handler catches a fault where write access to a vma without the flag set, it treats the fault as a permission violation error and sends a SIGSEGV signal to the application.

Second, NOVA-Fortis needs to guarantee that each snapshot is consistent. Taking consistent snapshots while applications are modifying files using DAX-style `mmap` requires NOVA-Fortis to reckon with the order in which stores to NVMM become persistent (i.e., reach physical NVMM so they will survive a system failure). These applications rely on the processor's "memory persistence model" [94] to make guarantees about when and in what order stores become persistent, allowing them to restore their data to a consistent state when the application restarts after a system failure.

From the application's perspective, reading a snapshot is equivalent to recovering from a system failure. In both cases, the contents of the memory-mapped file reflect its state at a moment when application operations might be in-flight and when the application had no chance to shut down cleanly.

We do not require applications to stop accessing mmaped pages when taking a snapshot, as snapshot creation never blocks file operations. Since applications use DAX-mmap bypass the file system, there is a subtle synchronization issue between the application thread and NOVA-Fortis file system, and it may leave the snapshot in an inconsistent state in case of a system failure: Setting the page to read-only captures its contents for the snapshot, and the kernel requires NOVA-Fortis to set the pages as read-only one at a time. If the order in which NOVA-Fortis marks pages as read-only is incompatible with ordering that the application requires, the snapshot will contain an inconsistent version of the file.

Consider an example with a data value D, and a flag V that is `True` when D contains valid data. To store data in D while enforcing this invariant, a thread could issue a persist barrier between the stores to D and V. Assume D and V reside in different pages, and initially V is `False`.

If NOVA-Fortis marks D's page as read-only *before* the program updates D, and marks V's page as read-only *after* the program updates V, the snapshot has V equal to `True`, but D with its old, incorrect value.

Figure 4.4 illustrates how we resolve the problem: When NOVA-Fortis starts marking pages as read-only in (a), it blocks page faults on read-only `mmap()`'d pages from completing

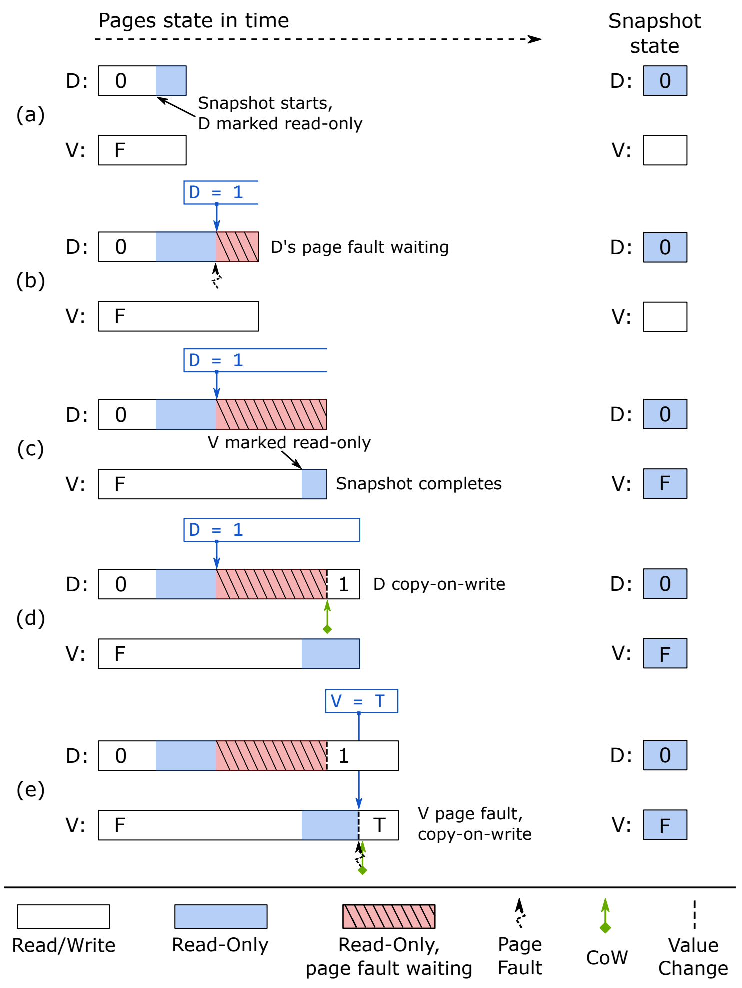

Figure 4.4: **Snapshots of `mmap()`'d data.** NOVA-Fortis marks `mmap()`'d pages as read-only to capture their contents in the snapshot. To preserve the application's constraints on when stores become persistent (here, that the assignment to V follow the assignment to D), it prevents pages faults to read-only pages from completing until it can mark all the pages read-only.

until it has marked all the `mmap()`'d pages read-only. D's page fault in (b) will not complete (and the store to V will not occur) until all the pages (including V's) have been marked read-only and the snapshot is complete in (c). Then NOVA-Fortis allows the copy-on-write for D's page to proceed, allowing the modification of D to complete in (d). Since the assignment to V occurs after the assignment to D in program order, the copy-on-write and modification to V in (e) will occur after creating a consistent snapshot.

This solution is also correct for multi-threaded programs. If thread 1 updates D and, later, thread 2 should update V, the threads must use some synchronization primitive to enforce that ordering. For instance, thread 1 may release a lock after storing to D. If D's page is marked read-only before D changes, the unlock will not occur until the kernel marks all the `mmap()`'d pages read-only, ensuring that the store to V will not appear in the snapshot.

This approach guarantees that, for each thread, the snapshot will reflect a prefix of the program order-based sequence of NVMM store operations the thread has performed. This model is equivalent to strict persistence [94], the most restrictive model for how NVMM memory operations should behave (i.e., in what order updates can become persistent) in a multi-processor system. CPUs may implement more aggressive, relaxed models for memory persistence, but strict persistence is strictly more conservative than all proposed models, so NOVA-Fortis' approach is correct under those models as well [1].

4.1.4 Design Decisions and Evaluation

There are several ways we could have addressed the problems of creating snapshots, managing the data they contain, and correctly capturing `mmap()`'d data. Our approach stores snapshot manifests in DRAM. The size of snapshot manifest is proportional to the changes between snapshots, and each data block and log entry is referred to by at most one snapshot manifest. In the worst case the snapshot manifest size is proportional to the size of the live data in file system, this happens when the user takes a snapshot and then modifies the entire file system.

In practice, the size of the manifests is manageable. We ran the fileserver workloads described in Section 4.3 and the WHISPER [86] workloads while taking snapshots every 10 seconds. WHISPER is a suite of eight applications that use DAX `mmap()` for data access. Table 4.1 shows the results. The size of the manifests ranged from 0.027% to 0.05% of the space in use in the file system.

[1]This is directly analogous to sequential memory consistency being a valid implementation of any more relaxed memory consistency model. Strict persistence is a natural extension of sequential consistency to include a notion of persistence.

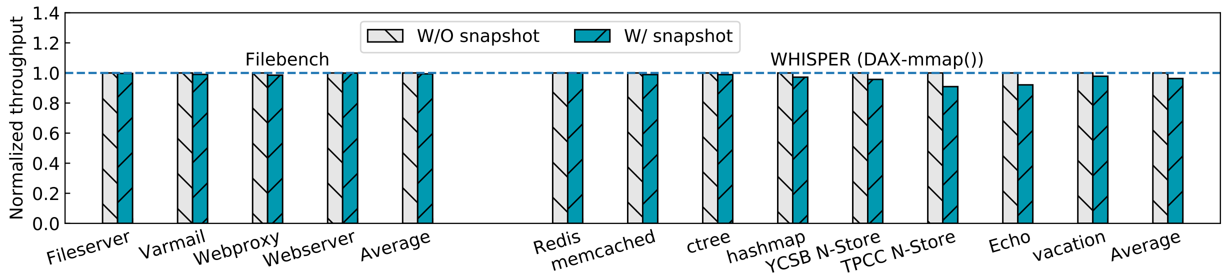

Figure 4.5: **Snapshot performance impact.** file-based application ("Filebench"), taking a snapshot every 10 s reduces performance by just 0.6% on average. WHISPER applications that use DAX-`mmap()` see a larger drop: 3.7% on average.

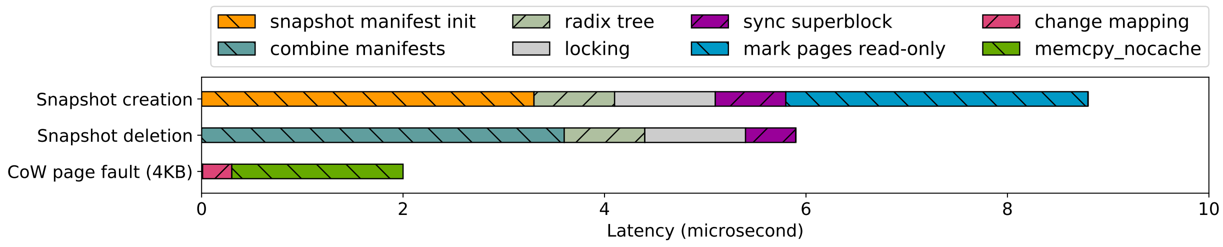

Figure 4.6: **Snapshot operation latency.** The time required to mark pages read-only during snapshot creation is proportional to the number of write-faulted pages (300 in this example) in the DAX `mmap()`'d regions.

Taking snapshots has different effects on the performance of applications that use file-based access and those that use DAX-style `mmap()`. Figure 4.5 measures the impact on both groups. On the right, it shows results for the WHISPER applications. On the left are results for four Filebench [2] workloads.

For all the applications, the figure compares performance without snapshots to performance while running for 1 to 5 minutes and taking a snapshot every 10 seconds. For the WHISPER applications that use DAX `mmap()`, taking periodic snapshots only reduces performance by 3.7% on average. For file-based filebench workloads, the impact is negligible – 0.6% on average.

Figure 4.6 shows the latency of snapshot operations. For file-based applications, snapshot creation takes constant time. For DAX `mmap()` applications, the time to mark `mmap()`'d regions as read-only is proportional to the number of write-faulted pages, and each page takes about 10 ns. Snapshot deletion always takes constant time, since it only needs to combine the deleted

Table 4.1: **Snapshot manifest size.** The size of snapshot manifest is proportional to the size of changes between snapshots.

Application	Workload size	Max manifest size
Fileserver	13 GB	3.6 MB
Varmail	4 GB	2 MB
Redis	4 GB	1.1 MB
TPCC	8.5 GB	3.1 MB
ctree	10 GB	3.3 MB

snapshot's manifest with the next snapshot's manifest and then perform stale data reclamation in the background. The background thread takes about 400 ns to reclaim 4 KB data. A copy-on-write page fault takes 2 μs: 1.8 μs for copying data to new page and 200 ns to update the page table.

4.2 Handling Data Corruption in NOVA-Fortis

NVMM is volnerable to data and metadata corruption from media failures and software bugs, like all storage medias. To prevent, detect, and recover from data corruption, NOVA-Fortis relies on the capabilities of the system hardware and operating system as well as its own error detection and recovery mechanisms.

This section describes the interfaces that NOVA-Fortis expects from the memory system hardware and the OS and how it leverages them to detect and recover from corruption. We also discuss a technique that prevents data corruption in many cases and NOVA-Fortis' ability to trade reliability for performance. Finally, we discuss NOVA-Fortis' protection mechanisms in the context of recent work on file system reliability.

4.2.1 Detecting and Correcting Media Errors

NOVA-Fortis detects NVMM media errors with the same mechanisms that processors provide to detect DRAM errors. The details of these mechanisms determine how NOVA-Fortis and other NVMM file systems can protect themselves from media errors.

NOVA-Fortis assumes the memory system provides ECC for NVMM, and the memory controller transparently corrects correctable errors, and silently returns invalid data for undetectable errors.

For detectable but uncorrectable errors, Intel's Machine Check Architecture (MCA) [52] raises a *machine check exception (MCE)* in response to uncorrectable memory errors. After the exception, MCA registers hold information that allows the OS to identify the memory address and instruction responsible for the exception.

The default response to an MCE in the kernel is a kernel panic. However, recent Linux kernels include a version of `memcpy()`, called `memcpy_mcsafe()`, that returns an error to the caller instead of crashing in response to memory-error-induced MCEs, and allows the kernel software to recover from the exception. NOVA-Fortis always uses this function when reading from NVMM and checks its return code to detect uncorrectable media errors. Writes to NVMM do not fail because the memory controller transparently maps around memory cells on writes to faulty cells. In rare cases (e.g., an MCE occurring during a page fault), MCEs are not recoverable, and a kernel panic is inevitable.

When the processor hardware detects an uncorrectable media error, it "poisons" a contiguous region of physical addresses. The size of this region is the *poison radius* (PR) of a media error. We assume PRs are a power of two in size and aligned to that size. Loads to poisoned addresses cause an MCE, and all the data in the PR is lost. The poisoned status of a PR persists across system failures and the PMEM driver collects a list of poisoned PRs at boot. On Intel processors the poison radius is 64 bytes (one cache line), but after boot, Linux reports poisoned regions at 512-byte granularity, so NOVA-Fortis uses 512-byte.

We also assume that NVMM platforms will allow system software to clear a poisoned PR to make the address range usable again. Intel processors provide this capability via the "Clear Uncorrectable Error" command that is part of the Advanced Configuration and Power Interface (ACPI) specification [127].

4.2.2 Tick-Tock Metadata Protection

NOVA-Fortis protects its metadata by keeping two copies of each structure – a *primary* and a *replica* – and adding a CRC32 checksum to both.

To update a metadata structure, NOVA-Fortis first copies the contents of the data structure into the primary (the *tick*), and issues a persist barrier to ensure that data is written to NVMM. Then it does the same for the replica (the *tock*). This scheme ensures that, at any moment, at least one of the two copies is correctly updated and has a consistent checksum.

To reliably access a metadata structure NOVA-Fortis copies the primary and replica into DRAM buffers using `memcpy_mcsafe()` to detect media errors. If it finds none, it verifies the checksums for both copies. If it detects that one copy is corrupt due to a media error or checksum mismatch, it restores it by copying the other. If both copies are error free but not identical, the system failed between the tick and tock phases of a previous update, and NOVA-Fortis copies the primary to the replica, effectively completing the interrupted update. If both copies are corrupt, the metadata is lost, and NOVA-Fortis returns an error.

4.2.3 Protecting File Data

NOVA-Fortis adopts RAID-4 parity protection and checksums to protect file data and it includes features to maximize protection for files that applications access data via DAX-style `mmap()`.

RAID Parity and Checksums NOVA-Fortis treats each 4 KB file page as a stripe, and divides it

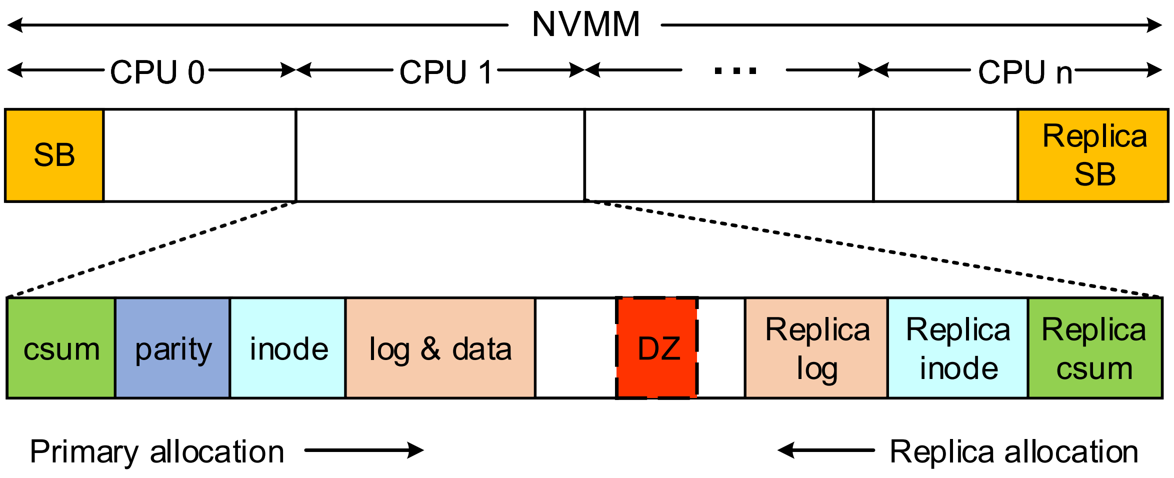

Figure 4.7: **NOVA-Fortis space layout.** NOVA-Fortis' per-core allocators satisfy requests for primary and replica storage from different directions. They also store data pages and their checksum and parity pages separately. The "dead zone" (DZ) for metadata allocation guarantees a gap between primary and replica pages.

into PR-sized (or larger) stripe segments, or *strips*. NOVA-Fortis stores a parity strip for each file page in a reserved region of NVMM. It also stores two copies of a CRC32 checksum for each data strip in separate reserved regions. Figure 4.7 shows the checksum and parity layouts in the NVMM.

For reads, NOVA-Fortis reads both the data and checksum, and uses parity to recover data if checksum mismatches. Writes and atomic parity updates are simple since NOVA-Fortis uses copy-on-write for data: For each file page write, NOVA-Fortis allocates new pages, populates them with the written data, computes the checksums and parity, and finally commits the write with an atomic log appending operation.

Caveat DAXor: Protecting DAX-mmap'd Data By design, DAX-style `mmap()` lets application modify file data without involving the file system, so it is impossible for NOVA-Fortis to keep the checksums and parity for read/write mapped pages up-to-date. Instead, NOVA-Fortis follows the *caveat DAXor* principle and provides the following guarantee: The checksums and parity for data pages are up-to-date at all times, except when those pages are mapped read/write into an application's address space.

We believe this is the strongest guarantee that NOVA-Fortis can provide on current hardware, and it raises several challenges. First, users that use DAX-`mmap()` take on responsibility for data protection and recovery. Second, NOVA-Fortis must be able to tell when a data page's checksums and parity should match the data it contains and when they might not.

To accomplish this, when a portion a file is `mmap()`'d, NOVA-Fortis records this fact in the file's log, signifying that the checksums and parity for the affected pages are no longer valid. NOVA-Fortis only recomputes the checksums and parity for dirty pages on `msync()` and `munmap()`. On `munmap()`, it adds a log entry that restores protection for these pages when the last mapping for the page is removed. If the system crashes while pages are mapped, the recovery process will identify these pages while scanning the logs, recompute checksums and parity, and add a log entry to mark them as valid.

4.2.4 Minimizing Vulnerability to Scribbles

Scribbles such as software bugs pose significant risk to NOVA-Fortis' data and metadata, since a scribble can impact large, continuous regions of memory. To quantify the risk that these errors pose, we define *bytes-at-risk (BAR)* for a scribble as the number of bytes it may render irretrievable. For replicated log pages, a scribble that spans both copies of a byte will corrupt the page.

To protect NOVA-Fortis metadata and data from large scribbles, NOVA-Fortis uses a two-way allocator to ensure enough space between primary and replicas. Figure 4.7 shows the space layout of NOVA-Fortis. NOVA-Fortis puts the super block at the beginning of the NVMM space and the replica at the end. For each core allocator, NOVA-Fortis allocates primary from low address to high address, and replicas from the reverse direction. This simple mechanism guarantees NOVA-Fortis allocates primary and replica copies far from each other, and put data pages away from their checksum and parity pages.

To further provide protection when the file system is full, NOVA-Fortis puts a virtual

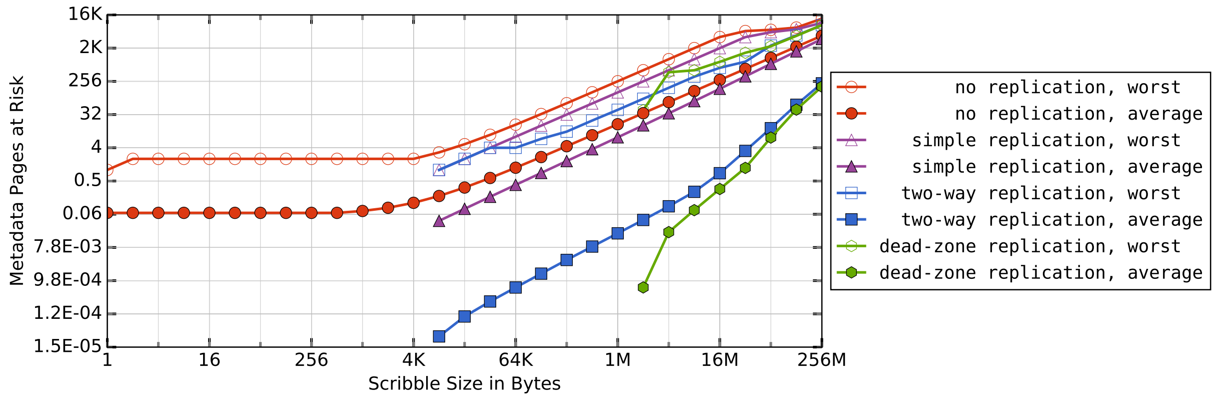

Figure 4.8: **Scribble size and metadata bytes at risk.** Replicating metadata pages and taking care to allocate the replicas separately improves resilience to scribbles. The most effective technique enforces a 1 MB "dead zone" between replicas and eliminates the threat of a single scribble smaller than 1 MB. The graph omits zero values due to the vertical log scale.

1 MB "dead zone" between the primary and replica. The dead zone can store file data but not metadata. The more separation the allocator provides, the less likely a scribble will corrupt a pair of mirrored metadata pages.

To show the efficiency of two-way allocator, we measure impact of scribble size on NOVA-Fortis integrity. Figure 4.8 shows the maximum and average metadata BAR for a 64 GB NOVA-Fortis file system with four protection schemes for metadata: "no replication" does not replicate metadata; "simple replication" allocates the primary and replicas naively and tends to allocate lower addresses before higher address, so the primary and replica are often close; "two-way replication" enables the two-way allocator, and "dead-zone replication" enables the virtual dead zone.

The data show that even for the smallest 1-byte scribble, the unprotected version will lose up to a whole page (4 KB) of metadata and an average of 0.06 pages. With simple replication, scribbles smaller than 4 KB have zero BAR. Under simple replication, an 8 KB scribble can corrupt up to 4 KB, but affects only 0.04 pages on average.

Two-way replication reduces the average bytes at risk with an 8 KB scribble to 2.9×10^{-5} pages, but the worst case remains the same because the allocator's options are limited when space is scarce. Enforcing the dead zone further improves protection: A 1 MB dead zone can eliminate metadata corruption for scribbles smaller than 1 MB. The dead zone size is configurable, so NOVA-Fortis can increase the 1 MB threshold for scribble vulnerability if larger scribbles are a concern.

Scribbles also place data pages at risk. Since NOVA-Fortis stores the strips of data pages contiguously, scribbles that are larger than the strip size may causes data loss. NOVA-Fortis could tolerate larger scribbles to data pages by interleaving strips from different pages, but this would disallow DAX-style `mmap()`. Increasing the strip size can also improve scribble tolerance, but at the cost of increased storage overhead for the parity strip.

4.2.5 Protecting DRAM Data Structures and Preventing Scribbles

Corruption of DRAM data structures can result in file system corruption [141, 38], and NOVA-Fortis protects most of its critical DRAM data structures with checksums. Most DRAM structures that NOVA-Fortis does not protect are short lived (e.g., the DRAM copies we create of metadata structures) or are not written back to NVMM. However, the snapshot logs and allocator state are exceptions and NOVA-Fortis protects them with checksums.

NOVA-Fortis supports a technique that is used in WAFL [63] and PMFS [41] to prevent scribbles. NOVA-Fortis marks all of NVMM as read-only and then clearing Intel's WriteProtect Enable (WP) bit to disable *all* write protection when NOVA-Fortis needs to modify NVMM. Clearing and re-setting the bit takes $\sim$400 ns on our systems. However, the WP approach cannot prevent NOVA-Fortis from corrupting its own data by performing "misdirected writes," a common source of data corruption in file systems [13].

4.3 Performance Trade-offs

NOVA-Fortis' reliability features improve its resilience but also incur overhead in terms of performance and storage space. This section quantifies these overheads and explores the trade-offs they allow. We use the PMEP platform introduced in Section 3.3.1 for evaluation. NVDIMM-N [106] emulates fast NVMMs that have latency and bandwidth same as DRAM, and PCM emulates slow NVMMs.

We evaluate the performance for existing NVMM file systems and several versions of NOVA-Fortis: "Baseline" is the baseline version with no data or metadata protection; Based on the baseline version, "MP" provides metadata protection (Section 4.2.2); "MP+WP" combines metadata protection and the write protection to prevent scribbles (Section 4.2.5); "MP+DP" provides data and metadata protection (Section 4.2.3); "MP+DP+WP" enables all our reliability mechanisms; finally, "Relaxed mode" is the high-performance mode of NOVA that disables protection and allows in-place updates of data.

4.3.1 Microbenchmarks

We first evaluate basic file system operations: `create`, 4 KB `append`, 4 KB `write`, 512 B `write`, and 4 KB `read`. Figure 4.9 measures the latency for these operations with NVDIMM-N configuration. Data for PCM has similar trends.

`Create` is a metadata-only operation. NOVA is 1.9× to 5× faster than the existing file systems, and adding metadata protection increases the latency by 47% compared to the baseline. `Append` affects metadata and data updates. Adding metadata and data protection increase the latency by 36% and 100%, respectively, and write protection increases the latency by an additional 22%. NOVA-Fortis with *full protection* (i.e., "w/ MP+DP+WP") is 59% slower than NOVA.

For overwrite, NOVA-Fortis performs copy-on-write for file data to provide data atomicity guarantees, and the latency is close to that of `append`. For 512 B overwrite, NOVA-Fortis has

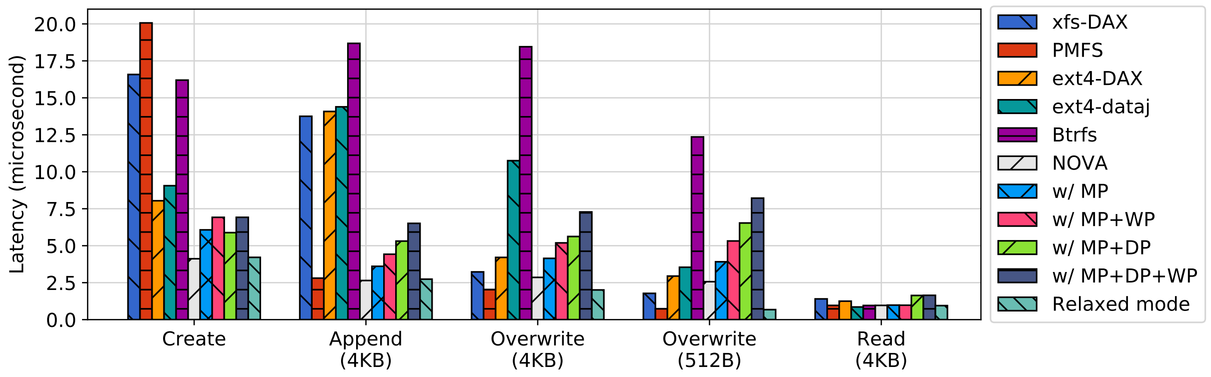

Figure 4.9: **NOVA-Fortis file operation latency.** NOVA-Fortis' basic file operations are faster than competing file systems except in cases where the other file system provides weaker consistency guarantees and/or data protection (e.g., writes and overwrites vs. Ext4-DAX and xfs-DAX). Sacrificing these protections in NOVA-Fortis ("Relaxed mode") improves performance.

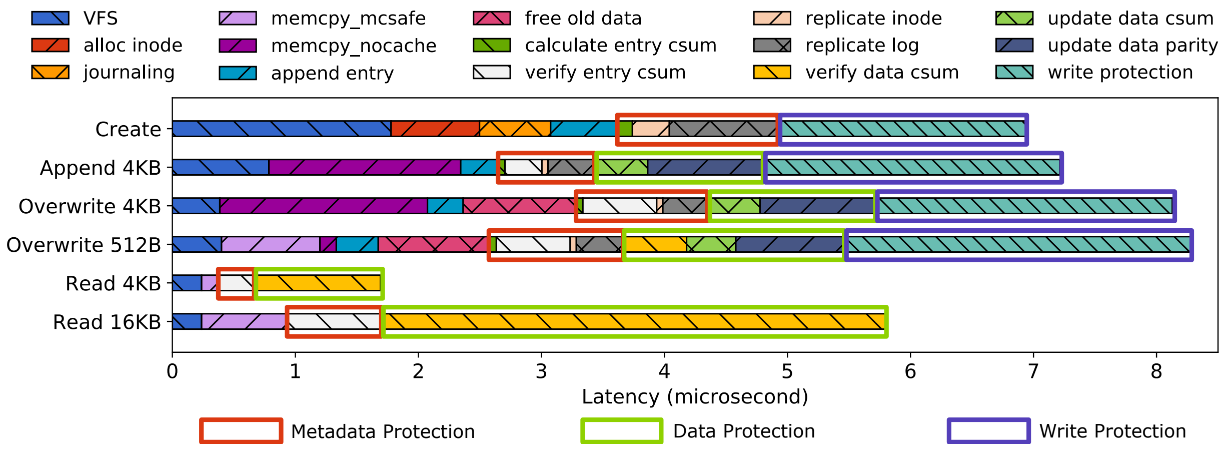

Figure 4.10: **NOVA-Fortis latencies for NVDIMM-N.** file data is usually more expensive than protecting metadata because the cost of computing checksums and parity for data scales with access size.

longer latency than other DAX file systems since its requires reading and writing 4 KB. Full protection increases the latency by 2.2×. Relaxed mode is 3.8× faster than NOVA since it performs in-place updates. For `read` operations, data protection adds 70% overhead because it verifies the data checksum before returning to the user.

Figure 4.10 breaks down the latency for NOVA-Fortis and its reliability mechanisms. For `create`, inode allocation and appending to the log combine to consume 48% of latency,

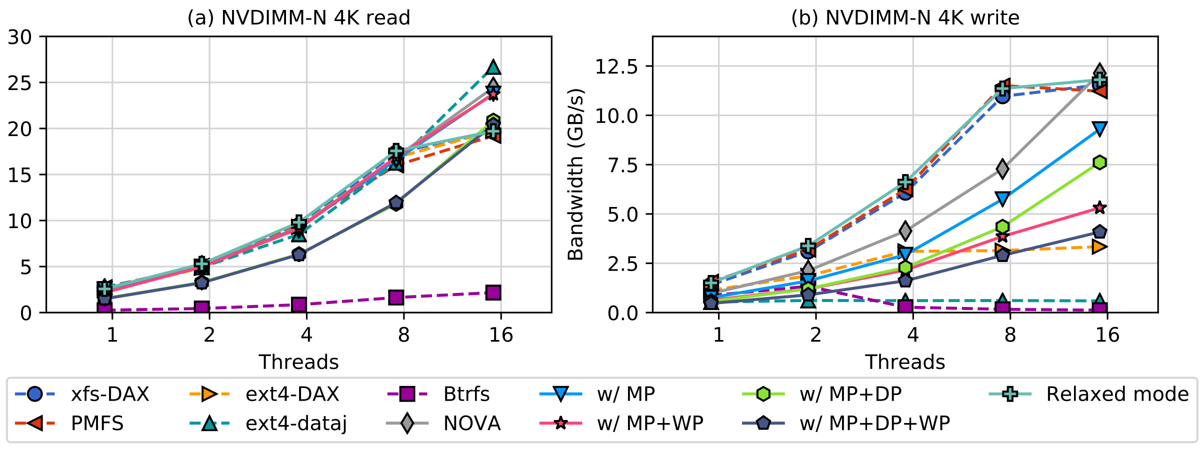

Figure 4.11: **NOVA-Fortis random read/write bandwidth on NVDIMM-N.** Read bandwidth is similar across all the file systems except Btrfs, and NOVA-Fortis' reliability mechanisms reduces its throughput by between 14% and 19%. For writes the cost of reliability is higher – up to 66%, but still outperforms Btrfs by 33× with 16 threads.

due to inode/log replication and checksum calculation. For 4 KB `append` and `overwrite`, data protection has almost the same latency as memory copy (`memcpy_nocache`), and it accounts for 31% of the total latency in 512 B `overwrite`.

Figure 4.11 shows FIO [11] measurements for the multi-threaded read/write bandwidth of the file systems. For writes, NOVA-Fortis' relaxed mode achieves the highest bandwidth. With sixteen threads, metadata protection reduces NOVA-Fortis bandwidth by 24% compared to the baseline, data protection reduces throughtput by 37%, and enabling all of NOVA-Fortis' protection features reduces bandwidth by 66%. For reads, all the file systems scale well except Btrfs, while NOVA-Fortis data protection incurs 14% overhead on 16 threads, due to checksum verification.

4.3.2 Macrobenchmarks

We use nine application-level workloads to evaluate NOVA-Fortis: Four Filebench [2] workloads (fileserver, varmail, webproxy, and webserver), two key-value stores (RocksDB [43] and MongoDB [84]), the Exim email server [42], SQLite [119], and TPC-C running on Shore-

Table 4.2: **Application benchmarks.**

Application	Data size	Notes
Filebench-fileserver	64 GB	R/W ratio: 1:2
Filebench-varmail	32 GB	R/W ratio: 1:1
Filebench-webproxy	32 GB	R/W ratio: 5:1
Filebench-webserver	32 GB	R/W ratio: 10:1
RocksDB	8 GB	db_bench's overwrite test
MongoDB	10 GB	YCSB's 50/50-read/write
Exim	4 GB	Mail server
SQLite	400 MB	Insert operation
TPC-C	26 GB	The 'Payment' query

MT [117]. Fileserver, varmail, webproxy and Exim are metadata-intensive workloads, while other workloads are data-intensive. Table 4.2 summarizes the workloads.

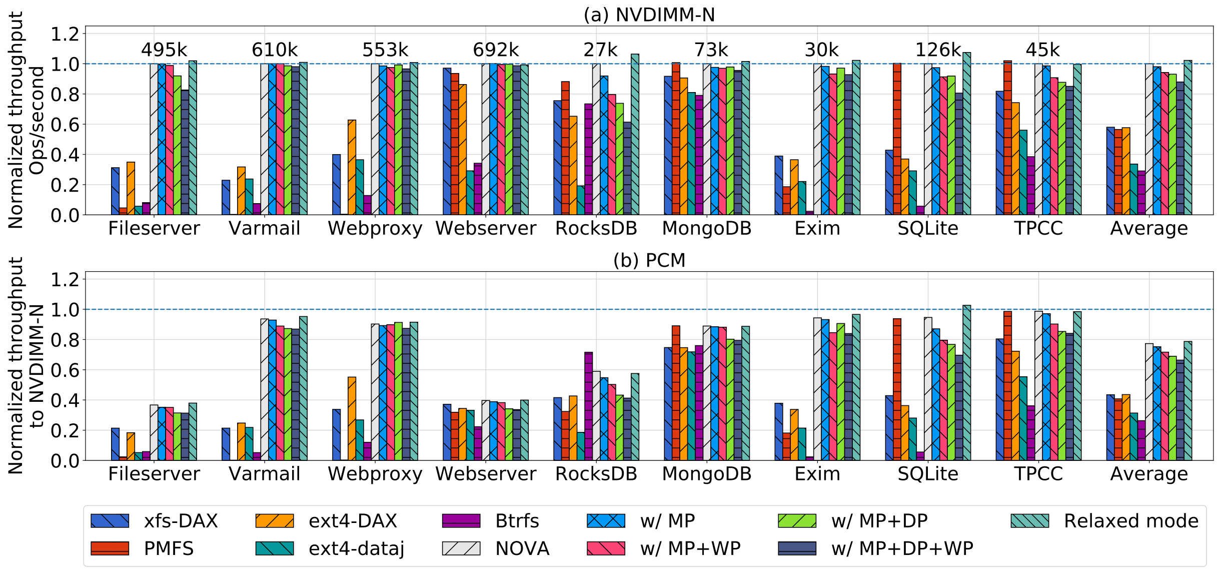

Figure 4.12: **NOVA-Fortis application performance.** Reliability overheads and the benefits of relaxed mode have less impact on applications than microbenchmarks (Figures 4.11 and 4.9). The change is especially small for read-intensive workloads (e.g., web-proxy), while databases and key-value stores see greater differences. The numbers above the bars measure NOVA throughput in operations per second.

Figure 4.12 measures their performance on our five comparison file systems and several NOVA-Fortis configurations, normalized to the NOVA throughput on NVDIMM-N. NOVA-Fortis outperforms xfs-DAX and ext4-DAX by between 3% and 4.4×. PMFS shows similar performance to NOVA on data-intensive workloads, but NOVA-Fortis outperforms it by a wide margin (up to 350×) on metadata-intensive workloads. Btrfs provides reliability features similar to NOVA-Fortis', but it is slower: NOVA-Fortis with all its protection features enabled outperforms it by between 26% and 42×. NOVA-Fortis achieves larger improvement on metadata-intensive workloads, such as varmail and Exim.

Adding metadata protection reduces performance by between 0 and 9% and using the WP bit costs an additional 0.1% to 13.4%. Enabling all protection features reduces performance by between 2% and 38%, with write-intensive workloads seeing the larger drops. The figure also shows that the performance benefits of giving up atomicity in file operations ("Relaxed mode") are modest – no more than 6.4%.

RocksDB sees the biggest performance loss with NOVA-Fortis with all protections enabled because it issues many non-page-aligned writes that result in extra reads, writes, and checksum calculation during copy-on-write. Relaxed mode avoids these overheads, so it improves performance for RocksDB more than for other workloads.

For the PCM configuration, fileserver, webserver and RocksDB show the largest performance drop compared to NVDIMM-N. Fileserver and RocksDB are write-intensive and saturate PCM's write bandwidth. Webserver is read-intensive and PCM's read latency limits performance. Btrfs outperforms other DAX file systems on RocksDB because this workload does not call `fsync` frequently, allowing it to leverage the page cache.

Compared to other file systems, NOVA-Fortis is more sensitive to NVMM performance, because it has lower software overhead and reveals the underlying NVMM performance more directly. Overall, NOVA outperforms other DAX file systems by 1.75× on average, and adding full protection reduces performance by 12% on average compared to NOVA.

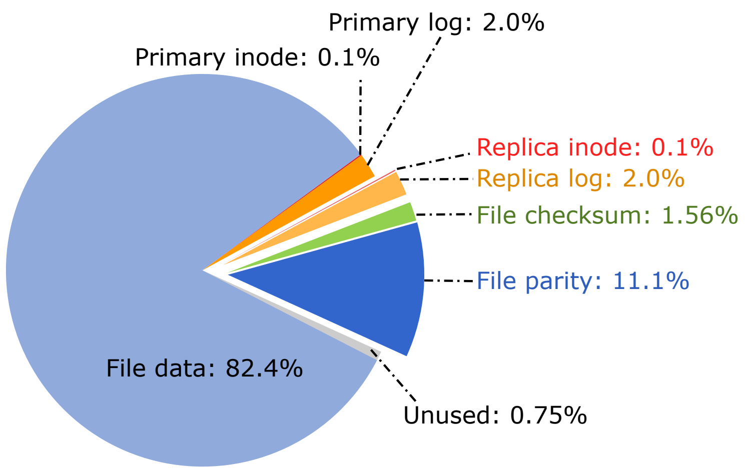

Figure 4.13: **NOVA-Fortis storage overheads.** Extra storage required for reliability is highlighted to the right. Protecting data is more expensive that protecting metadata, consuming 12.7% of storage compared to just 2.1% for metadata.

4.3.3 NVMM Storage Utilization

Protecting data integrity introduces storage overheads, and by design NOVA-Fortis has fixed overhead for each metadata structure (between 1.4% and 12.5%) and each file data page (12.7%). Nevertheless due to variation in the number of log entries, the storage overheads for real workloads also depend on the ratio between their metadata and file data.

Figure 4.13 shows the breakdown of space among (meta)data structures in an aged, 64 GB NOVA-Fortis file system. Overall, NOVA-Fortis devotes 14.8% of storage space to improving reliability. Of this, metadata redundancy accounts for 2.1% and data redundancy occupies 12.7%.

4.4 Summary

We have designed and developed NOVA-Fortis to explore the unique challenges that improving NVMM file system reliability presents. The solutions that NOVA-Fortis implements facilitate backups by taking consistent snapshots of the file system and provide significant protection against media errors and corruption due to software errors.

The extra storage required to implement these changes is modest, but their performance impact is significant for some applications. In particular, the cost of checking and maintaining checksums and parity for file data incurs a steep cost for both reads and writes. Providing atomicity for unaligned writes is also a performance bottleneck.

These costs suggest that NVMM file systems should provide users with a range of protection options that trade off performance against the level of protection and consistency. For instance, NOVA-Fortis can selectively disable checksum based file data protection and the write protection mechanism. Relaxed mode disables copy-on-write.

Making these policy decisions rationally is currently difficult due to a lack of two pieces of information. First, the rate of uncorrectable media errors in emerging NVMM technologies is not publicly known. Second, the frequency and size of scribbles has not been studied in detail. Without a better understanding in these areas, it is hard to determine whether the costs of these techniques are worth the benefits they provide. Interestingly, NOVA-Fortis provides the means to track scribbles, since it occupies and performs consistency checks on a large amount of kernel address space.

Despite these uncertainties, NOVA-Fortis demonstrates that NVMM file system can provide strong reliability guarantees while providing high performance and supporting DAX-style `mmap()`. It also makes a clear case for developing special file systems and reliability mechanisms for NVMM rather than blithely adapting existing schemes: The challenges NVMMs presents are different, different solutions are appropriate, and the systems built with these differences in mind can be very fast and highly reliable.

Acknowledgments

This chapter contains material from “NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System”, by Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito da Silva, Andy Rudoff and Steven Swanson, which appears in the

26th ACM Symposium on Operating Systems Principles (SOSP 2017). The dissertation author is the primary investigator and first author of this paper.

Chapter 5

Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks

So far we have described NOVA, a high-performance NVMM file system that provides strong consistency guarantee, and we have shown how to make NOVA reliable and fault-tolerant. In this section we explore the whole software storage stack for NVMM, and try to find and fix performance anomalies in the persistent memory software stack.

Progress has been made in understanding how file systems and application should adapt to NVMMs. Researchers, companies, and open-source communities have built *native NVMM file systems* built specifically for NVMMs[37, 41, 129, 137, 138, 64], both Linux and Windows have created *adapted NVMM file systems* by adding support for NVMM to existing file systems (e.g., ext4-DAX, xfs-DAX and NTFS),

Adapted file systems extend block-based file systems to support NVMM, and they still subject to the constraints of legacy disk support. They also give up some features when running in DAX mode. For instance, ext4 does not support data journaling in DAX mode, so it cannot provide strong consistency guarantees on file data.

On the other hand, people have built NVMM libraries [36, 130, 98, 128, 79, 140], try to hide the complexity from the applications and provide easy-to-use transaction support. These libraries generally provide persistent memory allocators, an object model, and support for transactions on persistent objects. However, using these libraries require substantial rewriting effort , which is not realistic for existing applications.

Despite all these progresses and known challenges, there is still a missing link in the NVMM storage stack. Although some commercial applications have begun to leverage NVMMs to improve performance and reduce recovery time [7, 77], several important questions remain about how applications can best exploit NVMMs and how file systems can best support those applications. These questions include:

1. How much effort is required to adapt legacy applications to exploit NVMMs? What best practices should developers follow?
2. Are sophisticated NVMM-based data structures necessary to exploit NVMM performance?
3. How effectively can legacy files systems evolve to accommodate NVMMs? What trade-offs are involved?
4. How effectively can current NVMM file systems scale to many-core, multi-socket systems?

This chapter tries to offer insight into all of these questions. We analyze the performance and behavior of benchmark suites and full-scale applications on multiple NVMM-aware file systems on a many-core machine. We identify bottlenecks caused by application design, file system algorithms, generic kernel interfaces, and basic limitations of NVMM performance. In each case, we propose solutions that aim to boost performance while minimizing the burden on the application programmer, thereby easing the transition to NVMM.

Our results offer a broad view of the current landscape of NVMM-optimized system software. Our findings include the following:

- A small number of relatively simple changes can provide substantial performance gains for legacy applications.
- Simple applications of DAX mmap can, in some cases, provide almost as much benefit as more complex persistent data structures.
- Block-based journaling mechanisms are a bottleneck for adapted NVMM file systems. Adding DAX-aware journaling improves performance on many operations.
- Despite several optimizations, a performance gap remains between native and adapted NVMM file systems.
- All NVMM file systems we tested suffer due to poor scalability of kernel locking primitives and/or VFS locking protocols. Generic solutions at the VFS layer are often possible (and beneficial). In some cases, the file systems can implement faster, specialized mechanisms.
- Poor performance in accessing non-local memory (NUMA effects) can significantly impact NVMM file system performance. Adding NUMA-aware interfaces to NVMM file systems can relieve these problems.

The remainder of the chapter is organized as follows. Section 5.1 evaluates applications on NVMM and recounts the lessons we learned in porting them to NVMMs. Section 5.2 describes the scalability bottlenecks of NVMM file systems and how to fix them, and Section 5.3 summaries.

5.1 Adapting applications to NVMM

We believe the first applications to use NVMM in production are likely to be legacy applications originally built for block-based storage. Directly running that code on a new, faster storage system will yield some gains, but fully exploiting NVMM's potential will require some tuning and modification. The amount, kind, and complexity of tuning required will help determine how quickly and how effectively applications can adapt. If application developers understand how to efficiently access NVMM file systems, they can adjust and optimize the I/O access patterns to better exploit NVMM performance.

In this section, we gathered first-hand experience with porting legacy applications to NVMM-based storage systems by modifying five lightweight databases and key-value stores to better utilize NVMMs. These applications have different access patterns to the file system and we believe they are representive. Below, we detail our experience and identify some best practices for NVMM programmers. Then, based on these findings, we propose DAX-aware journaling scheme for ext4-DAX that eliminates block IO overheads.

We use a quad-socket prototype HPE Scalable Persistent Memory server [49] to evaluate these applications. The server combines DRAM with NVMe SSDs and an integrated battery backup unit to create NVMM. The server hosts four Xeon Gold 6148 processors (a total of 80 cores), 300 GB of DRAM, and 300 GB of NVMM. We evaluate all the applications on Linux kernel 4.13.

5.1.1 SQLite

SQLite [119] is a lightweight embedded relational database that is popular in mobile systems. Many mobile applications use SQLite to store data such as user profiles. SQLite stores data in a B+ tree in a single file, and each tree node holds one or more database records.

To ensure consistency, SQLite performs journaling on each write transaction and keeps the journal in a file. SQLite uses one of four different techniques to log operations to the journal file. Three of the techniques, DELETE, TRUNCATE and PERSIST, store undo logs while the last, WAL stores redo logs.

The undo logging modes invalidate the log after every operation. DELETE and TRUNCATE, respectively, delete the log file or truncate it. PERSISTS issues a write to set an "invalid" flag in log file header.

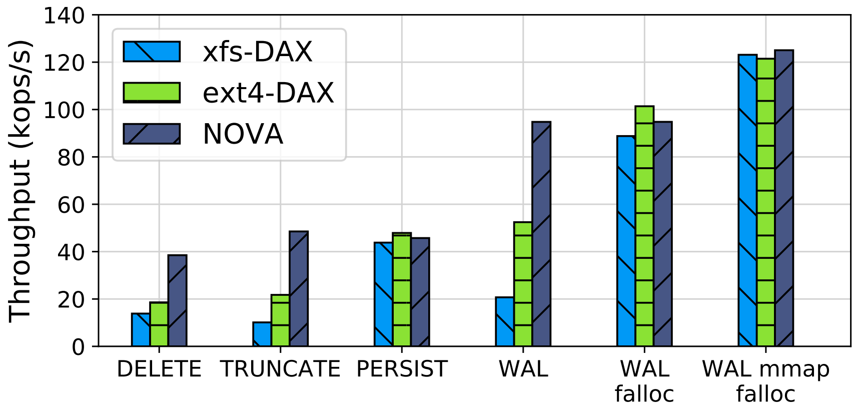

Figure 5.1: **SQLite SET throughput with different journaling modes.** Preallocating space for the log file using `falloc` avoids allocation overheads and makes write ahead logging (WAL) the clearly superior logging mode for SQLite running NVMM file systems. The further step to move the writes into userspace with DAX-mmap offers an additional boost.

WAL appends redo log entries to a log file and takes periodic checkpoints after which it resets the log and starts again.

We use Mobibench [55] to test the SET performance of SQLite in each journaling mode. The workload inserts 100 byte long values into a table. Figure 5.1 shows the result. DELETE and TRUNCATE incur significant file system journaling overhead with ext4-DAX and xfs-DAX. NOVA performs better because it does not need to journal single file operations. PERSIST mode performs equally on all three file systems.

WAL avoids file creation and deletion, but it does require allocating new space for each log entry. Ext4-DAX and xfs-DAX keep their allocator state in NVMM and keep it consistent at all times, so allocation is expensive. Persistent allocator state is necessary in block-based file systems to avoid a time-consuming media scan after a crash.

Scanning NVMM after crash is much less costly, so NOVA keeps allocator state in DRAM and only writes it to NVMM on a clean unmount. As a result, allocation is much faster. This difference in allocation overhead limits WAL's performance advantage compared to PERSIST to 9% for ext4-DAX, reduces performance by 53% for xfs-DAX, but improves NOVA's performance by 107%.

We modified SQLite to avoid allocation overhead by using `fallocate` to pre-allocate the WAL file. This change closes the gap between the three file systems, improves performance by 4.3× for xfs-DAX and 93% for ext4-DAX, respectively. To improve performance further, we can use DAX-mmap to avoid the kernel completely for writes to the WAL file. In this case, SQLite maps the WAL file into its address space and uses non-temporal stores and `clwb` to ensure the log entries are reliably stored in NVMM. Implementing these changes required changing just 266 lines of code but improved performance by between 15% and 38%.

This final DAX-aware version of SQLite outperforms the PERSIST version by between 2.5× and 2.8×.

Other groups have adapted SQLite to solid-states storage as well. Jeong et al. [56] and WALDIO [69] investigate SQLite I/O access patterns and optimized them on SSDs. SQLite/PPL [90], NVWAL [62] use slotted paging [116] to make SQLite to run efficiently on NVMM. A comparison to these systems would be interesting, but unfortunately, none of them is publicly available.

5.1.2 Tokyo Cabinet and LMDB

Even without DAX some applications access files via `mmap`. This makes them a natural match for DAX file systems, but maximizing the benefits of DAX still requires some changes. We select two applications to explore what is required: Tokyo Cabinet and LMDB.

Tokyo Cabinet Tokyo Cabinet [45] is a high performance database library. It stores the database in a single file with database metadata at the head. Tokyo Cabinet memory maps the metadata region, uses load/store instructions to access and update it, and calls `msync` to persist the changes.

We use Mobibench [55] to test the SET performance of SQLite in each Tokyo Cabinet makes frequent `msync` system calls to ensure that updates to memory-mapped data are persistent. DAX-mmap allows userspace to provide these guarantees using a series of `clwb` instructions followed by a memory fence. Flushing in user space is also more precise, since `msync` operates on pages rather cache lines.

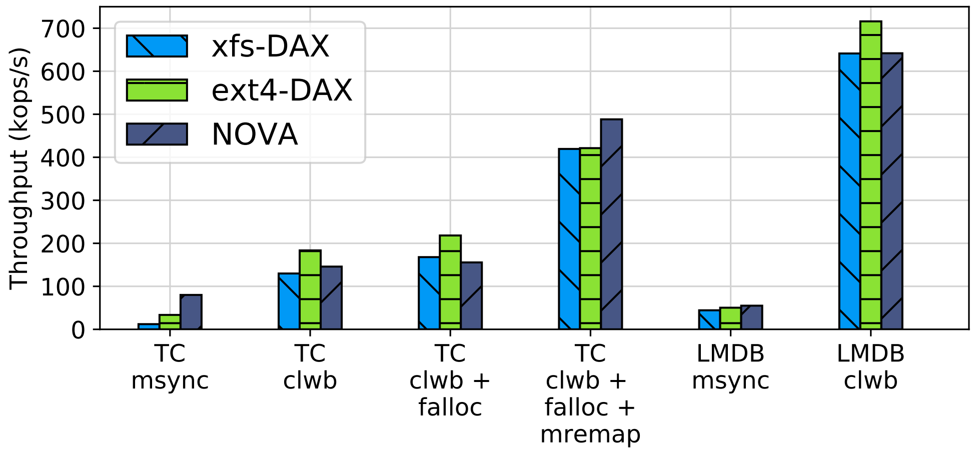

Figure 5.2: **Tokyo Cabinet (TC) and LMDB SET throughput.** Applications that use `mmap` can improve performance by performing `msync` in userspace.

One small caveat is that the call to `mmap` must include the recently-added MAP_SYNC flag that ensures that the file is fully allocated and its metadata has been flushed to media. This is necessary because, without MAP_SYNC, in the disk-optimized implementations of `msync` and `mmap` that ext4 and xfs provide, `msync` can sometimes require metadata updates.

The first two bar groupings in Figure 5.2 show the impact of these changes. The graph plots throughput for SET operation on Tokyo Cabinet. The key size is 8 bytes and value size is 1024 bytes. Avoiding `msync` improves performance by 82% for NOVA, 5.4× for ext4-DAX, and 10.7× for xfs-DAX. The significant performance improvement comes from fine-grained cache flushing: Most modified piece of metadata is much smaller than the page size, so `msync` results in unnecessary clean cache flushing and impacts performance.

By default, Tokyo Cabinet only mmaps the first 64 MB of the file, which includes the header and ∼63 MB of data. Tokyo Cabinet uses `write` to append new records to the file. Pre-allocating file with `fallocate` improves performance like it did for SQLite – 29% for xfs-DAX and 19% for ext4-DAX.

We can also keep the whole file mapped using `mrepmap` and then use load and store instructions to access all the records. This boosts the throughput for all the file systems by between 6× to 34×, compared to the baseline implementation that issued `msync` system calls.

Implementing the fully-optimized version required changing just 241 lines of code.

LMDB Lightning Memory-Mapped Database Manager (LMDB) [124] is a Btree-based lightweight database management library. LMDB memory-maps the entire database, so that all data accesses directly load and store the mapped memory region. Comparing the file appending, LMDB reduces the metadata changes and hence reduces the metadata journaling overhead in the NVMM file systems. LMDB performs copy-on-write on data pages to provide atomicity, a technique that requires frequent `msync` calls.

For LMDB, using `clwb` instead of `msync` improves the throughout by between 11× to 14×. Ext4-DAX out-performs xfs-DAX and NOVA by about 11% because ext4-DAX supports super-page (2 MB) `mmap` which reduces the number of page faults. These changes entailed changes to 101 lines of code.

5.1.3 RocksDB and Redis

Since disk is slow, many disk-based applications keep data structures in DRAM and flush them to disk only when necessary. To provide persistence, they also record updates in a persistent log, since sequential access is most efficient for disks. We consider two such applications, Redis and RocksDB, to understand how this technique can be adapted to NVMM.

Redis Redis [103] is a in-memory key-value store widely used in web site development as a caching layer and message queue applications. Redis provides optional durability guarantees. It can be executed purely in DRAM with no persistence, which suffice in some use cases. Redis also provides an "append only file" (AOF) to log all the write operations to the storage device for durability purpose. At recovery, it replays the log. The frequency at which Redis flushes the AOF persistent storage allows the administrator to trade-off between performance and consistency.

Figure 5.3 measures Redis MSET (multiple set) performance. As we have seen with other

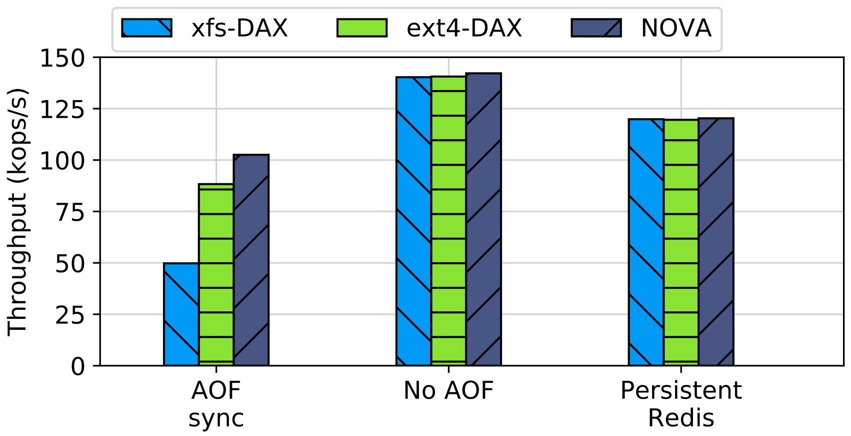

Figure 5.3: **Redis MSET throughput.** Redis appends new data to the AOF file and uses `fsync` to sync the file. Making Redis' core data structure persistent in NVMM improves performance by 17% to 2.4×.

applications, xfs-DAX's journaling overheads hurt append performance. The graph also shows the potential benefit of eliminating AOF (and giving up persistence): It improves throughput by 2.8×, 59%, and 38% for xfs-DAX, ext4-DAX, and NOVA, respectively.

The hash table Redis uses internally is an attractive target for NVMM conversion, since making it persistent would eliminate the need for the AOF. We created a fully-functional persistent version of the hash table in NVMM using PMDK [98] by adopting a copy-on-write mechanism for atomic updates: To insert/update a key-value pair, we allocate a new pair to store the data, and replace the old data by atomically updating the pointer in the hashtable.

The throughput with our persistent hash table is 17% to 2.4× better than using synchronous writes to the AOF, and ∼16% worse than skipping persistence altogether. Implementing the changes required changes to or the addition of 1326 lines of code.

RocksDB RocksDB [43] is a high-performance embedded key-value store based on log-structured merge trees (LSM-trees). RocksDB is composed of two parts: memory component and disk component. The memory compoonent is a sorted data structure that resided in DRAM, called memtable. The memtable absorbs new inserts and provides fast insertion and searches. The disk component is structured into multiple layers with increasing sizes. Each level contains multiple

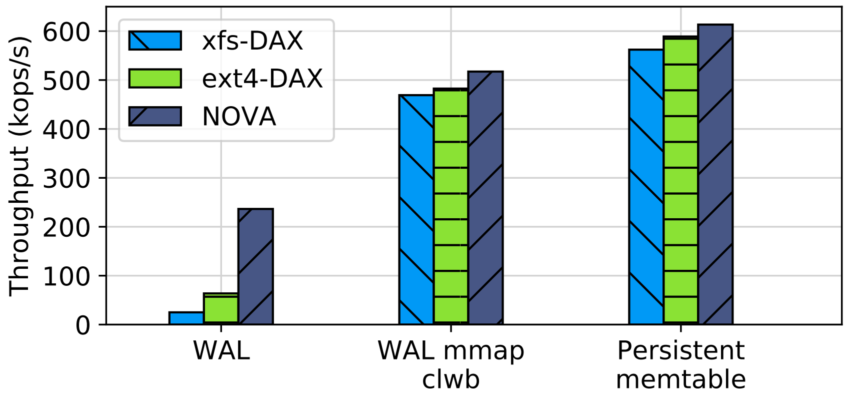

Figure 5.4: **RocksDB SET throughput.** RocksDB inserts new data to the skip-list and appends to the WAL file. WAL appending limits the throughput that NVMM file systems can provide. Using `mmap` and fine-grained clwb improves performance by 3× to 7.5×, and moving skip-list to NVMM and disable WAL improves throughput by another 19% on average.

sorted files, called Sorted Sequence Table (SSTable). When the memtable is full, it is flushed to disk and becomes a SSTable in the first layer. When the number of SSTables in a layer exceeds a threshold, RocksDB merges the SSTables with next layer's SSTables that have overlapping key ranges. This is called compaction, it reduces the number of disk accesses for read operations. When applications write data to a LSM-tree, RocksDB inserts the data to a skip-list in DRAM, and appends the data to a write-ahead log (WAL) file. When the skip-list is full, RocksDB writes it to disk and discards the log file.

RocksDB makes all I/O request sequential, make best use of hard disks' sequential access strength. It supports concurrent writes when old memtable is flushed to disk, and only performs large writes to the disk except for the WAL appending. However, WAL appending and sync operations can still impact performance significantly on NVMM file systems.

Figure 5.4 measures RocksDB SET throughput with 20-byte keys and 100-byte values. RocksDB's default settings perform poorly on xfs-DAX and ext4-DAX, because each append requires journaling for those file systems. NOVA performs better because it avoids this cost.

Once again, moving writes to userspace by DAX-mmaping the file and using `clwb` and memory fence, boost performance: throughput improves by 2.2× - 18.7× and the performance gap between file systems vanishes.

Since the skip-list contains the same information as the WAL file, we could eliminate the WAL file by making the skip-list a persistent data structure. To approximate the impact of this change, the final bar in Figure 5.4 measures the performance of RocksDB with a skip-list in NVMM that uses `clwb` to make updates persistent. This is not a complete implementation of a persistent skip-list (it does not support recovery after failures), but does provide some intuition for how much benefit a persistent skip-list could provide. It is unlikely to improve performance by more than this partial implementation does: 19%.

Other groups have considered a range of complimentary optimizations for LSM-trees. FloDB [15] and TRIAD [14] make optimizations to the different layers of the LSM-tree: they use using highly-concurrent data structures for the layers in memory and reduce write amplification by deferring and batching I/O requests to block devices. WiscKey [75] separates keys and the values and only keeps keys sorted, leveraging both the sequential and random performance characteristics of the SSD device. Similar techniques might apply to NVMMs.

5.1.4 Best Practices

Based on our experiences with these five applications, we can draw some useful conclusions about how applications can profitably exploit NVMMs.

Use DAX-mmap to perform writes in userspace Replacing `write` system calls with stores to an DAX-mmap'd file followed by cache flushes offered large performance gains for very little programmer effort.

Use fine-grained cache flushing instead of `msync` Applications that already use `mmap` and `msync` to access data and ensure consistency, can improve performance significantly by flushing cache lines rather than `msync`'ing pages. However, ensuring that all updated cache lines are flushed correctly can be a challenge.

Use complex persistent data structure judiciously For both of the DRAM data structures we considered making persistent, the programming effort required was significant and likely performance gains were relatively small. This finding leads us to two conclusions: First, it is critical to make building persistent data structures in NVMM as easy as possible. Second, it is wise to estimate the potential performance impact the persistent data structure will have before investing a large amount of programmer effort in developing it [77].

Preallocate files on native NVMM file systems Several of the performance problems we found with adapted NVMM file systems stemmed from storage allocation overheads. Using `fallocate` to pre-allocate file space eliminated them.

Avoid meta-data operations Directory operations (e.g., deleting files) and storage allocation incurred journaling overheads in both xfs and ext4. Avoiding them improves performance, but this is not always possible.

5.1.5 Reducing NVMM file system journaling overhead

Several of the best practices we identify above focus on avoiding metadata operations, since they trigger file system journaling and significantly impact performance. This can be awkward and some metadata operations are unavoidable, so improving their performance would make adapting to NVMMs easier and improve performance.

NOVA's mechanism for performing consistent metadata updates is tailored specifically for NVMMs, but ext4 and xfs' journaling mechanisms were built for disk, and this legacy is evident in their poorer metadata performance. To try to close this performance gap, we replaced ext4's journaling mechanism with a DAX-aware version and measured its impact on performance.

Ext4 uses the journaling block device (JBD2) to perform consistent metadata updates. To ensure atomicity, it always writes entire 4 KB pages, even if the metadata change affects a single byte. Transactions often involve multiple metadata pages. For instance, appending 4 KB data to a file and then calling `fsync` writes one data page and eight journal pages: a header, a commit block, and six pages for inode, inode bitmap and allocator. For a block device, the cost of writing extra blocks is small, since the writes are sequential.

JBD2 allows no concurrency between journaled operations, so concurrent threads must synchronize to join the same running transaction, making the journaling a scalability bottleneck [118]. Son et al. [118] and iJournaling [93] have tried to fix ext4's scalability issues by reducing lock contention and adding per-core journal areas to jbd2.

We developed a journaling DAX device (JDD) to perform DAX-style journaling. JDD makes three key improvements to JBD2. First, it journals individual metadata fields rather than entire pages. Second, it provides pre-allocated, per-CPU journaling areas so CPUs can perform journaled operations in parallel. Third, it uses undo logging in the journals: It copies the old values into the journal and performs updates directly to the metadata structures in NVMM.

When an application thread starts a file system operation, JDD starts a transaction on the current CPU's journal. JDD commits the transaction by simply marking the journal invalid and reseting the journal area's tail pointer. In case of a power failure, JDD rolls back partial updates by scanning all the journal areas, copying back the metadata of unfinished transactions to undo the changes.

Figure 5.5 shows JDD's impact on a microbenchmark that performs random 4 KB writes followed by `fsync`, Filebench [2] Varmail (which is metadata-intensive), and the three databases and key value stores we evaluated earlier that perform frequent metadata operations as part of WAL. The JDD improves the microbenchmark performance by 3.7$\times$ and varmail by 40%. For applications that use write-ahead logging, the benefits range from 11% to 2.6$\times$.

We further analyze the latency of JDD on 4 KB append and RocksDB SET operation and show the latency breakdown in Figure 5.6. In ext4-DAX, jbd2 transaction commit (jbd2_commit) occupies 50% of the total latency. JDD eliminates this overhead by performing undo logging so no

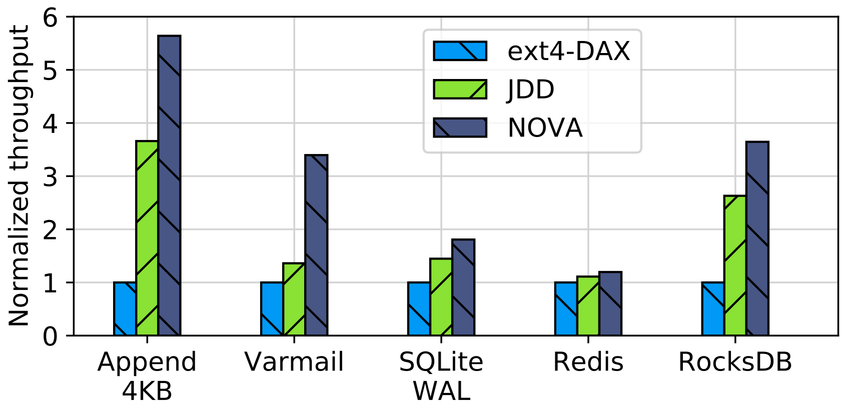

Figure 5.5: **JDD performance.** Fine-grained, DAX-optimized journaling on NVMM improves performance for metadata-intensive applications.

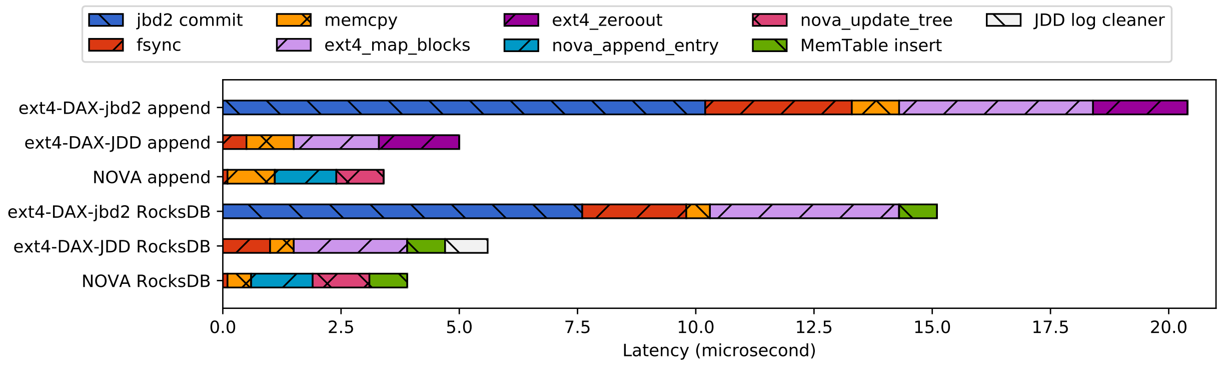

Figure 5.6: **Latency break for 4 KB append and RocksDB SET.** JDD significantly reduces journaling overhead by eliminating jbd2 transaction commit, but still has higher latency than NOVA.

commit is required. JDD also reduces ext4 overheads such as block allocation (ext4_map_blocks). The remaining performance gap between ext4 and NOVA (46%) is due to ext4's more complex design and its need to keep more persistent state in storage media. In particular (as discussed in Section 5.1.1) ext4 keeps its data block and inode allocator state continually up-to-date on disk.

The performance improvement on Redis and SQLite are smaller, because they have higher internal overhead. Redis spends most of its time on TCP transfers between the Redis server and the benchmark application, and SQLite spends over 40% of execution time parsing SQL and performing B-tree operations.

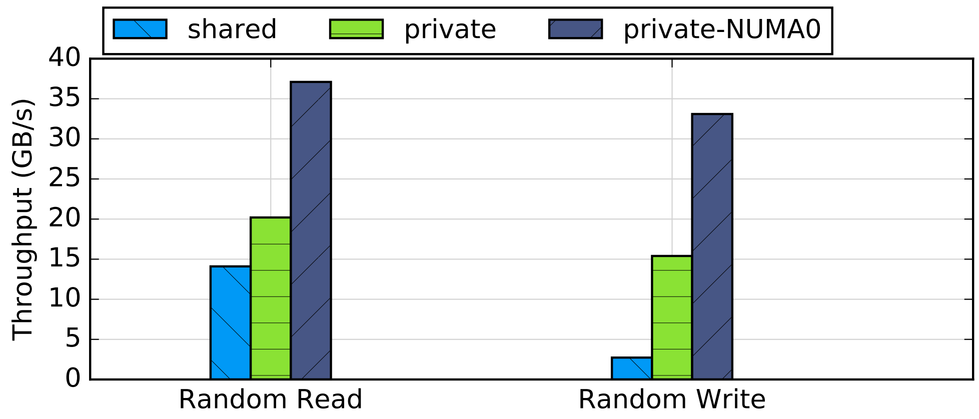

Figure 5.7: **FIO read/write bandwidth on many-core machine.** The FIO read/write bandwidth on a shared file is limited, and binding FIO threads to the pmem device node achieves even higher throughput.

5.1.6 FIO: scalability and NUMA limitation

By designing JDD and reducing journaling overhead, we have improved the metadata performance of DAX file systems. However, data read/write still have performance issues even without involving metadata changes.

We use FIO [11] to test the random read/write bandwidth on NOVA file system. The file is pre-allocated so there is no NVMM allocation. The FIO workload runs with 4 KB request size and 20 threads. We test three scenarios: all threads perform I/O to a shared file, each thread operates on its private file, and binding all the threads to NUMA node 0. Figure 5.7 shows the result. When each thread read/write a private file, the overall throughput is higher than read/write to a shared file: the write throughput improved by 5.6× due to inode lock contention. If we bind all the FIO threads to the same NUMA node as the pmem device located, the throughput improves by 84% and 2.15× for read and write, respectively.

The result demonstrates besides metadata journaling, there are still issues about scalability and NUMA effects in the file system. We will investigate more thoroughly in the following section.

5.2 Improving File System Scalability

Concurrent I/O operation is an essential technique to improve the I/O performance of applcations, and many applications perform parallel I/O operations. With faster storage device emerging and the number of CPU cores continue increasing, multi-core scalability has become a key technique, and high-performance databases and key-value stores [117, 126] have started to design their operations with multi-core scalability in mind.

Unfortunately, traditional disk-based file systems are not designed with many-core scalability, because disk is slow and only has a single disk head, and making file system multi-core aware does not improve file system performance. As a result, recent studies on multi-core scalable operating systems use im-memory file systems such as tmpfs to bypass the issue [23, 24]. We expect NVMM file systems to be subject to more onerous scalability demands than block-based filesystems due to the higher performance of the underlying media and the large amount of parallelism that modern memory hierarchies can support [19, 81]. Further, since NVMMs attach to the CPU memory bus, the capacity of NVMM file systems will tend to scale with the number sockets (and cores) in the systems.

Many-core scalability is also a concern for conventional block-based file systems, and researchers have proposed potential solutions. SpanFS [57] shards file and directories across cores at coarse granularity, requiring developers to carefully distribute the files and directories. ScaleFS [19] decouples the in-memory file system from the on-disk file system, and uses pre-core operation logs to achieve high concurrency.

Min et al [81] built a file system scalability test suite called FxMark and used it to identify many scalability problems in both file systems and Linux's VFS layer. We used FxMark to identify scalability problems that affect ext4-DAX, xfs-DAX, and/or NOVA. In this section we identify several operations that have scalability limitations and propose solutions.

5.2.1 Concurrent file read/write

Concurrent `read` and `write` operations to a shared file are a common hot spot in file system performance. Figure 5.9 shows scalability problems for both reads and writes across ext4-DAX, xfs-DAX, and NOVA. The root cause of this poor performance is Linux's read/write semaphore implementation [24, 25, 58, 73]. While in theory concurrent reads should scale, the atomic update required to acquire and release semaphore contends for the shared cacheline and limits performance.

This semaphore protects two things: The file data and the metadata that describes the file layout. To remove this bottleneck in NOVA, we use separate mechanisms to protect the data and metadata.

To protect file data, we leverage NOVA's logs. NOVA maintains one log per inode. Many of the log entries correspond to write operations and hold pointers to the file pages that hold the data for the write. Rather than locking the whole inode, we use reader/writer locks on each log entry to protect the pages it links to.

Figure 5.8 shows the modified NOVA log structure. We remove the inode semaphore from the NOVA I/O path, and add an atomic counter to each NOVA file write entry as a find-grained read/write lock. By default the counter is zero, a positive number means the number of active readers, and -1 means one writer is performing writes. Although this lock resides in NVMM, its state is not logically persistent, so hot locks will reside in processor caches and remain unaffected by NVMM latency. In case of power failure, NOVA scans all the inode logs to rebuild the allocator, and it reset the counter to zero during the scan.

NOVA's approach to tracking file layout makes protecting it simple. NOVA uses an in-DRAM radix tree to map file offsets to write entries in the log. Write operations update the tree and both reads and writes query it. Instead of using a lock we leverage the Linux radix tree implementation that uses read-copy update [78] to provide scalable, lock-free access. When a reader performs the tree lookup, it only needs to disable preemption on current CPU, without

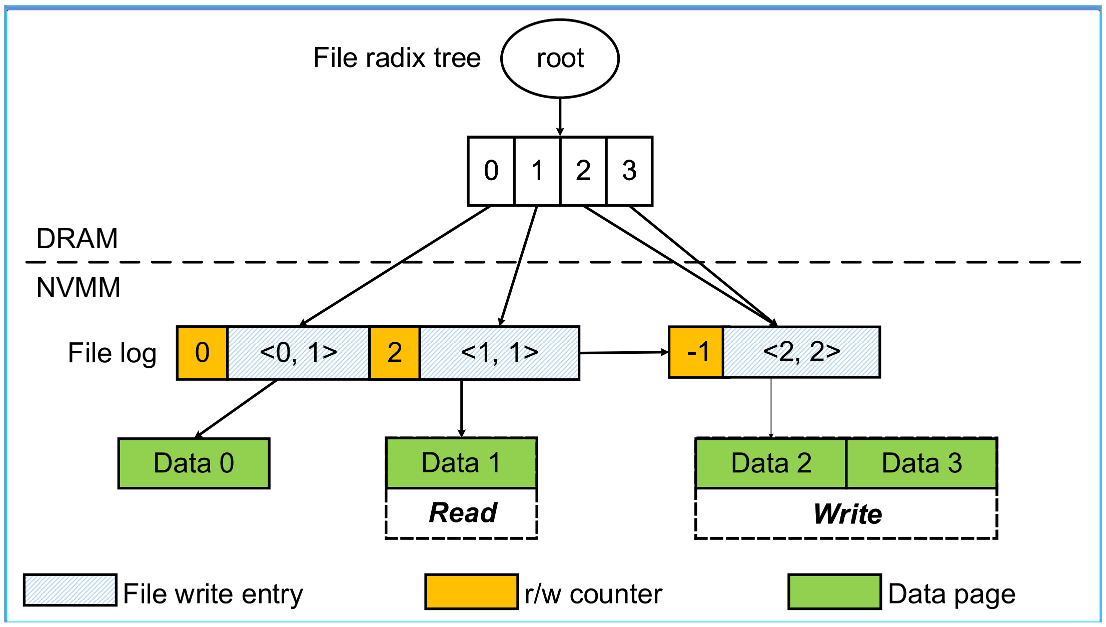

Figure 5.8: **NOVA lock-free read/write.** We remove the inode mutex locking from the I/O path, and add an atomic counter to each NOVA file write entry as a read/write lock. The file write entry describes a write with <page_offset, num_pages> tuple. Two threads are reading data page 1, and one thread is writing data page 2 and 3.

modifying any shared variables. Concurrent writes can run simultaneously if they do not overlap and do not change the radix tree.

Figure 5.9 shows the results (labeled “NOVA-lockfree”) on our 80-core machine. 4 KB read performance scales from 2.9 Mops/s for one thread to 183 Mops/s with 80 threads (63×). The changes improve write performance as well, but, because of NUMA effects, write bandwidth saturates at twenty threads.

Adding fine-grain locking for ranges of a file is possible for ext4-DAX and xfs-DAX, and it would improve performance when running on any storage device.

Using the radix tree to store file layout information would be more challenging, since ext4 and xfs make updates to file layout information immediately persistent in the file’s inode and indirect blocks. This is necessary to avoid reading the data from disk when the file is opened, which would be slow on block device. Since NVMM is much faster, NOVA can afford to scan the inode’s log on `open` to construct the radix tree in DRAM.

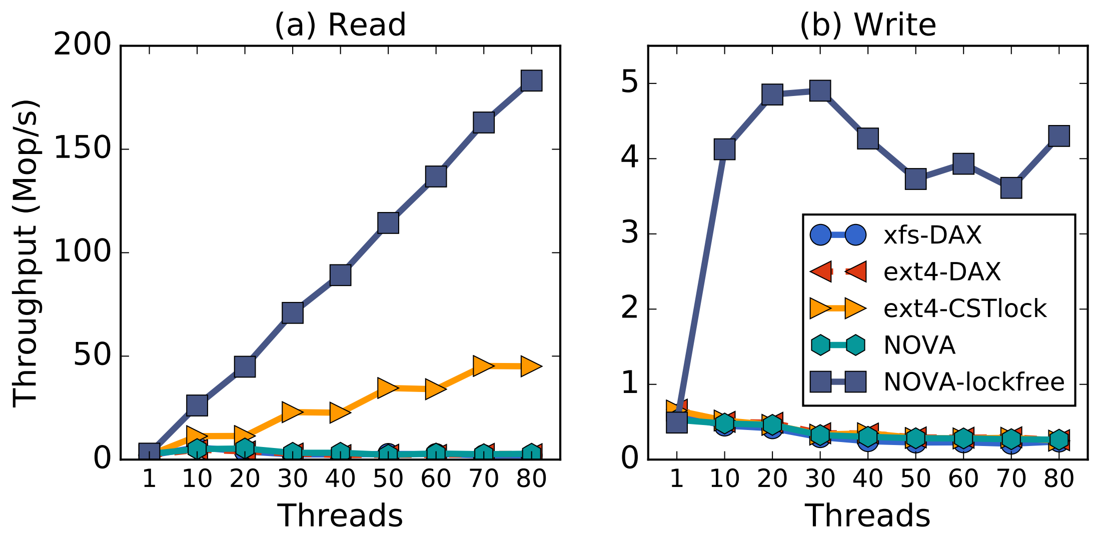

Figure 5.9: **Concurrent 4 KB read/write throughput.** Linux file systems use the inode reader/writer semaphore to synchronize concurrent reads and writes to shared files. However, that semaphore is not scalable and prevents concurrent reads from scaling. Using more scalable semaphores or fine-grain locking combined lock-free data structures can improve both read and write scalability.

An alternative solution for ext4 and xfs would be to replace VFS's per-inode reader/write semaphore with a CST semaphore [58] (or some other more scalable semaphore). The ext4-CSTlock line in the figure shows the impact on ext4-DAX: Performance scales from 2.1 Mops/s for one thread to 45 Mops/s for eighty threads (21×). The gains are not as large as approach we implemented in NOVA, and they only apply to reads. Both of these approaches could coexist.

5.2.2 Directory Accesses

Scalable directory operations are critical in multi-program, data intensive workloads. Figure 5.10 shows that, surprisingly, creating files in private directories only scales to twenty cores. The root cause is VFS taking a spinlock to add the new inode to the superblock's inode list, and maintains a global inode cache. The inode list includes all live inodes so that the file system can evict all of them on unmount. The global inode cache is an open-chaining hash table with 1,048,576 slots, providing mapping from inode number to inode addresses. The inode cache

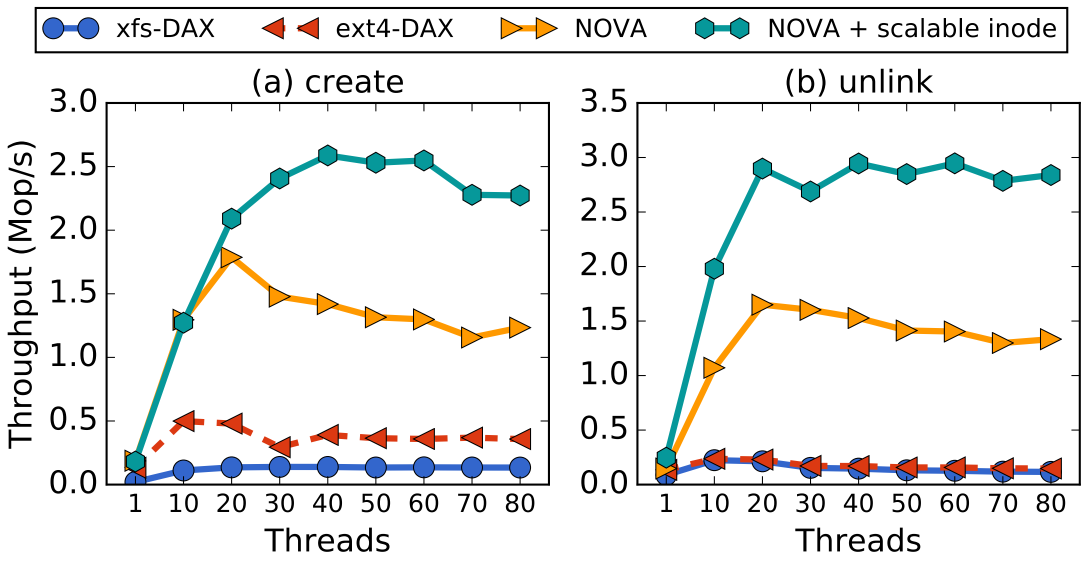

Figure 5.10: **Concurrent `create/unlink` throughput.** The `create` and `unlink` operations are not scalable even if performed in isolated directories, because Linux protects the global inode lists and inode cache a single spinlock. Moving to per-cpu structures and fine-grain locks improves throughput by 2× with 80 cores.

is protected by a single spinlock and all the mounted file systems share this cache, making it a scalability bottleneck.

To improve scalability for the inode list, we break it into per-CPU lists and protect each with a private spinlock. For inode cache table, We modify NOVA to use its own per-core inode cache table. The table is distributed across the cores, each core maintains a radix tree that provides lock-free lookups, and threads on different cores can perform inserts concurrently. In Figure 5.10, the "NOVA + scalable inode" line shows the resulting improvements in scaling.

Updates to shared directories also scale poorly due to VFS locking. For every directory operation, VFS takes the inode mutexes of all the affected inodes, so operations in the shared directories are serialized. The `rename` operation is globally serialized at a system level in the Linux kernel for consistent updates of the dentry cache. Fixing these problems is beyond the scope of this thesis, but recent work has addressed them [125, 19].

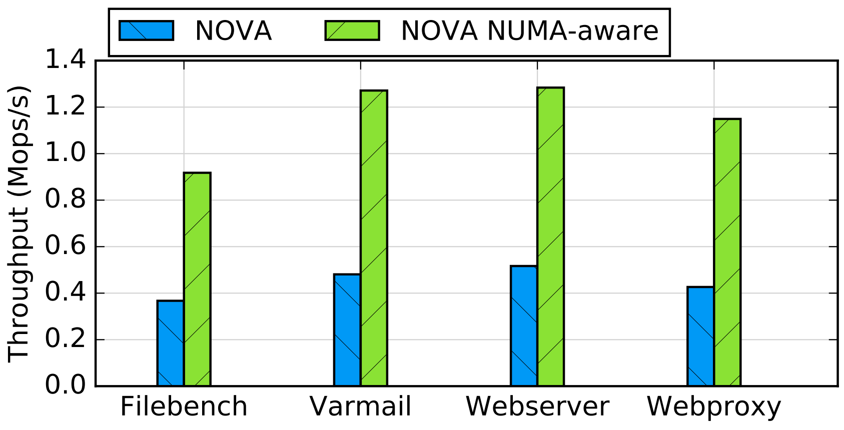

Figure 5.11: **NUMA-awareness in the file system.** Since NVMM is memory, NUMA effects impact performance. Providing simple controls over where the file system allocates NVMM for a file lets application run threads near the data they operate on, leading to higher performance.

5.2.3 NUMA Scalability

Each socket in a NUMA system has its local memory and can access other sockets' remote memory. Such systems present a uniform programming model that all memory is globally visible, but access latency and bandwidth vary depending on whether a core is accessing local or remote memory.

Intelligently allocating memory in NUMA systems is critical to maximizing performance. Since a key task of NVMM file systems is allocating memory, these file systems should be NUMA-aware. Otherwise, poor data placement decisions will lead to poor performance [40], since an application may create and write to a file when running on one NUMA node, but re-execute on another NUMA node and accesses the file remotely.

We have added NUMA-aware features to NOVA to understand the impact they can have. We created a new `ioctl` that can set and query the preferred NUMA node for the file. A NUMA node represents a set of processors and memory regions that close to one another in terms of memory access latency. The file system will try to use that node to allocate all the metadata and data pages for that file. A thread can use this ioctl along with Linux's CPU affinity mechanism to bind itself to the NUMA node where the file's data and metadata resides.

Figure 5.11 shows the result of Filebench workloads with fifty threads. The NVMM is attached to NUMA node 0. Without the new mechanism, threads are spread across all the NUMA nodes, and some of them are accessing NVMM remotely. Binding threads to the NUMA node that holds the file they are accessing improves performance by 2.6× on average.

5.3 Summary

We have examined the performance of NVMM-aware storage software stacks to better understand the trade-offs between adapted and native NVMM file systems and to understand how applications benefit from migrating to NVMMs. We find that by making relatively small changes these applications can achieve substantial performance gains, and demonstrate how NVMM-aware journaling can boost performance for applications. Finally, we describe and resolve several scalability bottlenecks in current NVMM file systems and the Linux storage stack.

Acknowledgments

This chapter contains material from "Finding and Fixing Performance Pathologies in Persistent Memory Software Stacks", by Jian Xu, Juno Kim and Steven Swanson, which is submitted to the 13th USENIX Symposium on Operating Systems Design and Implementation, (OSDI 2018). The dissertation author is the primary investigator and first author of this paper.

The author also would like to thank Thierry Fevrier and Anton Gavriliuk from Hewlett Packard Enterprise for their help and hardware access.

Chapter 6

Conclusion

Non-volatile main memory (NVMM) technologies promise vast improvements in storage performance, but adapting the software storage stack for data stored in NVMM raises a host of challenges. Existing NVMM file systems can compromise the NVMM performance with their high software overheads, and fail to provide the strong consistency and integrity guarantees that applications require.

This thesis first presents NOVA, a log-tructured file system designed to maximize performance on hybrid memory systems while providing strong consistency guarantees. NOVA proposed a novel technique called per-inode logging to exploit the fast random accesses that NVMMs provide. NOVA resolved the performance issue of conventional LFSs and provides metadata and data atomicity with low overhead. Experimental results show NOVA excels in metadata-intensive and write-intensive workloads, outperforming existing NVMM file systems by a wide margin.

Then we extend NOVA with fault-tolerance features and present NOVA-Fortis, a reliable file system that is both fast and robust against media errors and software bugs. NVMM has very different characteristics from disks and SSDs, and existing fault-tolerance features may not directly apply to NVMMs. We identified and proposed solutions for the unique challenges when developing NOVA-Fortis, and evaluated the performance and storage overheads of these techniques.

Finally, we investigated existing applications and analyzed their access patterns to the file system. We resolved several performance inefficiencies from both application and file system perspective. We also analyzed and fixed the file system scalabilty and NUMA impact issues on a many-core machine.

Bibliography

[1] BtrFS official site. https://btrfs.wiki.kernel.org/index.php/Main_Page.

[2] Filebench. https://github.com/filebench/filebench/wiki.

[3] MySQL. https://www.mysql.com/.

[4] NILFS2 official site. http://nilfs.sourceforge.net/en/.

[5] 3d xpoint. https://newsroom.intel.com/news-releases/intel-and-micron-produce-breakthrough-memory-technology/.

[6] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A protoype phase change memory storage array. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Storage and File Systems*, HotStorage'11, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[7] M. Andrei, C. Lemke, G. Radestock, R. Schulze, C. Thiel, R. Blanco, A. Meghlan, M. Sharique, S. Seifert, S. Vishnoi, D. Booss, T. Peh, I. Schreter, W. Thesing, M. Wagle, and T. Willhalm. SAP HANA Adoption of Non-volatile Memory. *Proc. VLDB Endow.*, 10(12):1754–1765, Aug. 2017.

[8] D. Apalkov, A. Khvalkovskiy, S. Watts, V. Nikitin, X. Tang, D. Lottis, K. Moon, X. Luo, E. Chen, A. Ong, A. Driskill-Smith, and M. Krounbi. Spin-transfer Torque Magnetic Random Access Memory (STT-MRAM). *J. Emerg. Technol. Comput. Syst.*, 9(2):13:1–13:35, May 2013.

[9] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau. *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.80 edition, May 2014.

[10] J. Arulraj, A. Pavlo, and S. R. Dulloor. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 707–722, New York, NY, USA, 2015. ACM.

[11] J. Axboe. Flexible I/O Tester, 2017. https://github.com/axboe/fio.

[12] L. N. Bairavasundaram, G. R. Goodson, S. Pasupathy, and J. Schindler. An Analysis of Latent Sector Errors in Disk Drives. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '07, pages 289–300, New York, NY, USA, 2007. ACM.

[13] L. N. Bairavasundaram, M. Rungta, N. Agrawa, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. Analyzing the Effects of Disk-Pointer Corruption. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*, pages 502–511, June 2008.

[14] O. Balmau, D. Didona, R. Guerraoui, W. Zwaenepoel, H. Yuan, A. Arora, K. Gupta, and P. Konka. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 363–375, Santa Clara, CA, 2017. USENIX Association.

[15] O. Balmau, R. Guerraoui, V. Trigonakis, and I. Zablotchi. FloDB: Unlocking Memory in Persistent Key-Value Stores. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 80–94, New York, NY, USA, 2017. ACM.

[16] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, New York, NY, USA, 2000. ACM.

[17] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Implications of CPU Caching on Byte-addressable Non-volatile Memory Programming. Technical report, HP Technical Report HPL-2012-236, 2012.

[18] M. S. Bhaskaran, J. Xu, and S. Swanson. BankShot: Caching Slow Storage in Fast Non-Volatile Memory. In *First Workshop on Interactions of NVM/Flash with Operating Systems and Workloads*, INFLOW '13, 2013.

[19] S. S. Bhat, R. Eqbal, A. T. Clements, M. F. Kaashoek, and N. Zeldovich. Scaling a File System to Many Cores Using an Operation Log. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 69–86, New York, NY, USA, 2017. ACM.

[20] A. Bityutskiy. JFFS3 design issues. www.linux-mtd.infradead.org/tech/JFFS3design.pdf, 2005.

[21] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems, 2007.

[22] L. Bouganim, B. Jónsson, and P. Bonnet. uFLIP: Understanding Flash IO Patterns. *arXiv preprint arXiv:0909.1780*, 2009.

[23] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An Operating System for Many

Cores. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[24] S. Boyd-Wickizer, A. T. Clements, Y. Mao, A. Pesterev, M. F. Kaashoek, R. Morris, and N. Zeldovich. An Analysis of Linux Scalability to Many Cores. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association.

[25] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, pages 119–130, 2012.

[26] M. J. Breitwisch. Phase change memory. *Interconnect Technology Conference, 2008. IITC 2008. International*, pages 219–221, June 2008.

[27] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A High-performance Storage Array Architecture for Next-generation, Non-volatile Memories. In *Proceedings of the 43nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 43, pages 385–395, New York, NY, USA, 2010. ACM.

[28] A. M. Caulfield, T. I. Mollov, L. 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*, New York, NY, USA, March 2012. ACM.

[29] A. M. Caulfield and S. Swanson. QuickSAN: A Storage Area Network for Fast, Distributed, Solid State Disks. In *ISCA '13: Proceedings of the 40th Annual International Symposium on Computer architecture*, 2013.

[30] C. Chen, J. Yang, Q. Wei, C. Wang, and M. Xue. Optimizing File Systems with Fine-grained Metadata Journaling on Byte-addressable NVM. *ACM Trans. Storage*, 13(2):13:1–13:25, May 2017.

[31] F. Chen, D. A. Koufaty, and X. Zhang. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. *ACM SIGMETRICS Performance Evaluation Review*, 37(1):181–192, 2009.

[32] J. Chen, Q. Wei, C. Chen, and L. Wu. FSMAC: A file system metadata accelerator with non-volatile memory. In *Mass Storage Systems and Technologies (MSST), 2013 IEEE 29th Symposium on*, pages 1–11. IEEE, 2013.

[33] S. Chen and Q. Jin. Persistent B$^+$-trees in Non-volatile Main Memory. *Proc. VLDB Endow.*, 8(7):786–797, Feb. 2015.

[34] D. Chinner. xfs: updates for 4.2-rc1, 2015. http://oss.sgi.com/archives/xfs/2015-06/msg00478.html.

[35] Y. Choi, I. Song, M.-H. Park, H. Chung, S. Chang, B. Cho, J. Kim, Y. Oh, D. Kwon, J. Sunwoo, J. Shin, Y. Rho, C. Lee, M. G. Kang, J. Lee, Y. Kwon, S. Kim, J. Kim, Y.-J. Lee, Q. Wang, S. Cha, S. Ahn, H. Horii, J. Lee, K. Kim, H. Joo, K. Lee, Y.-T. Lee, J. Yoo, and G. Jeong. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 46–48, Feb 2012.

[36] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 105–118, New York, NY, USA, 2011. ACM.

[37] 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*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[38] T. Do, T. Harter, Y. Liu, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. HARDFS: Hardening HDFS with Selective and Lightweight Versioning. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*, pages 105–118, San Jose, CA, 2013. USENIX.

[39] M. Dong and H. Chen. Soft Updates Made Simple and Fast on Non-volatile Memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 719–731, Santa Clara, CA, 2017. USENIX Association.

[40] M. Dong, Q. Yu, X. Zhou, Y. Hong, H. Chen, and B. Zang. Rethinking Benchmarking for NVM-based File Systems. In *Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems*, APSys '16, pages 20:1–20:7, New York, NY, USA, 2016. ACM.

[41] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[42] Exim Internet Mailer, 2017. http://www.exim.org.

[43] Facebook. RocksDB, 2017. http://rocksdb.org.

[44] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara, and G. Hush. A 16Gb ReRAM with 200MB/s write and 1GB/s read in 27nm technology. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2014 IEEE International*, pages 338–339, Feb 2014.

[45] FAL Labs. Tokyo Cabinet: a modern implementation of DBM, 2010. http://fallabs.com/tokyocabinet.

[46] J. Gray. The Transaction Concept: Virtues and Limitations (Invited Paper). In *Proceedings of the Seventh International Conference on Very Large Data Bases - Volume 7*, VLDB '81, pages 144–154. VLDB Endowment, 1981.

[47] R. HAGMANN. Reimplementing the cedar file system using logging and group commit. In *Proc. 11th ACM Symposium on Operating System Principles Austin, TX*, pages 155–162, 1987.

[48] R. Harris. Windows leaps into the NVM revolution, 2016. http://www.zdnet.com/article/windows-leaps-into-the-nvm-revolution/.

[49] Hewlett Packard Enterprise. HPE Scalable Persistent Memory, 2018. https://www.hpe.com/us/en/servers/persistent-memory.html.

[50] D. Hitz, J. Lau, and M. A. Malcolm. File system design for an NFS file server appliance. In *USENIX Winter*, pages 235–246, 1994.

[51] Intel. NVDIMM Namespace Specification, 2015. http://pmem.io/documents/NVDIMM_Namespace_Spec.pdf.

[52] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3, Chapter 15, 2016. https://software.intel.com/sites/default/files/managed/a4/60/325384-sdm-vol-3abcd.pdf, Version December 2016.

[53] Intel. Intel Architecture Instruction Set Extensions Programming Reference, 2017. https://software.intel.com/sites/default/files/managed/0d/53/319433-022.pdf.

[54] S. G. International. XFS: A High-performance Journaling Filesystem. http://oss.sgi.com/projects/xfs.

[55] S. Jeong, K. Lee, J. Hwang, S. Lee, and Y. Won. AndroStep: Android Storage Performance Analysis Tool. In *Software Engineering (Workshops)*, volume 13, pages 327–340, 2013.

[56] S. Jeong, K. Lee, S. Lee, S. Son, and Y. Won. I/O Stack Optimization for Smartphones. In *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 309–320, San Jose, CA, 2013. USENIX.

[57] J. Kang, B. Zhang, T. Wo, W. Yu, L. Du, S. Ma, and J. Huai. SpanFS: A Scalable File System on Fast Storage Devices. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 249–261, Santa Clara, CA, 2015. USENIX Association.

[58] S. Kashyap, C. Min, and T. Kim. Scalable numa-aware blocking synchronization primitives. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 603–615, Santa Clara, CA, 2017. USENIX Association.

[59] J. Katcher. Postmark: A new file system benchmark. Technical report, Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, 1997.

[60] T. Kawahara. Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing. *Design & Test of Computers, IEEE*, 28(1):52–63, Jan 2011.

[61] R. Kesavan, R. Singh, T. Grusecki, and Y. Patel. Algorithms and Data Structures for Efficient Free Space Reclamation in WAFL. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, 2017.

[62] W.-H. Kim, J. Kim, W. Baek, B. Nam, and Y. Won. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 385–398, New York, NY, USA, 2016. ACM.

[63] H. Kumar, Y. Patel, R. Kesavan, and S. Makam. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*, pages 197–212, Santa Clara, CA, 2017. USENIX Association.

[64] Y. Kwon, H. Fingler, T. Hunt, S. Peter, E. Witchel, and T. Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 460–477, New York, NY, USA, 2017. ACM.

[65] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[66] C. Lee, D. Sim, J. Hwang, and S. Cho. F2FS: A New File System for Flash Storage. In *13th USENIX Conference on File and Storage Technologies*, FAST '15, pages 273–286, Santa Clara, CA, Feb. 2015. USENIX Association.

[67] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies*, FAST '13, pages 73–80, San Jose, CA, 2013. USENIX.

[68] E. K. Lee and C. A. Thekkath. Petal: Distributed virtual disks. *SIGOPS Oper. Syst. Rev.*, 30(5):84–92, Sept. 1996.

[69] W. Lee, K. Lee, H. Son, W.-H. Kim, B. Nam, and Y. Won. WALDIO: Eliminating the Filesystem Journaling in Resolving the Journaling of Journal Anomaly. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, pages 235–247, Santa Clara, CA, 2015. USENIX Association.

[70] P. H. Lensing, T. Cortes, and A. Brinkmann. Direct Lookup and Hash-based Metadata Placement for Local File Systems. In *Proceedings of the 6th International Systems and Storage Conference*, SYSTOR '13, pages 5:1–5:11, New York, NY, USA, 2013. ACM.

[71] Trees I: Radix trees. https://lwn.net/Articles/175432/.

[72] XFS tests. http://oss.sgi.com/cgi-bin/gitweb.cgi?p=xfs/cmds/xfstests.git;a=summary.

[73] R. Liu, H. Zhang, and H. Chen. Scalable Read-mostly Synchronization Using Passive Reader-Writer Locks. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 219–230, Philadelphia, PA, 2014. USENIX Association.

[74] D. E. Lowell and P. M. Chen. Free Transactions with Rio Vista. In *SOSP '97: Proceedings of the Sixteenth ACM Symposium on Operating Systems Principles*, pages 92–101, New York, NY, USA, 1997. ACM.

[75] L. Lu, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. WiscKey: Separating Keys from Values in SSD-conscious Storage. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 133–148, Santa Clara, CA, 2016. USENIX Association.

[76] Y. Lu, J. Shu, and W. Wang. ReconFS: A Reconstructable File System on Flash Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 75–88, Berkeley, CA, USA, 2014. USENIX Association.

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

[78] P. E. McKenney, J. Appavoo, A. Kleen, O. Krieger, R. Russell, D. Sarma, and M. Soni. Read-copy update. In *AUUG Conference Proceedings*, page 175. AUUG, Inc., 2001.

[79] A. Memaripour, A. Badam, A. Phanishayee, Y. Zhou, R. Alagappan, K. Strauss, and S. Swanson. Atomic In-place Updates for Non-volatile Main Memories with Kamino-Tx. In *Proceedings of the Twelfth European Conference on Computer Systems*, EuroSys '17, pages 499–512, New York, NY, USA, 2017. ACM.

[80] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. A Large-Scale Study of Flash Memory Failures in the Field. In *Proceedings of the 2015 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '15, pages 177–190, New York, NY, USA, 2015. ACM.

[81] C. Min, S. Kashyap, S. Maass, and T. Kim. Understanding Manycore Scalability of File Systems. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, pages 71–85, Denver, CO, 2016. USENIX Association.

[82] S. Mittal, J. S. Vetter, and D. Li. A Survey Of Architectural Approaches for Managing Embedded DRAM and Non-Volatile On-Chip Caches. *IEEE Transactions on Parallel and Distributed Systems*, 26(6):1524–1537, June 2015.

[83] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-granularity Locking and Partial Rollbacks Using Write-ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[84] MongoDB, Inc. MongoDB, 2017. https://www.mongodb.com.

[85] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert. Consistent, Durable, and Safe Memory Management for Byte-addressable Non Volatile Main Memory. In *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, TRIOS '13, pages 1:1–1:17, New York, NY, USA, 2013. ACM.

[86] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 135–148, New York, NY, USA, 2017. ACM.

[87] D. Narayanan and O. Hodson. Whole-system Persistence with Non-volatile Memories. In *Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*. ACM, March 2012.

[88] I. Narayanan, D. Wang, M. Jeon, B. Sharma, L. Caulfield, A. Sivasubramaniam, B. Cutler, J. Liu, B. Khessib, and K. Vaid. SSD Failures in Datacenters: What? When? And Why? In *Proceedings of the 9th ACM International on Systems and Storage Conference*, SYSTOR '16, pages 7:1–7:11, New York, NY, USA, 2016. ACM.

[89] H. Noguchi, K. Ikegami, K. Kushida, K. Abe, S. Itai, S. Takaya, N. Shimomura, J. Ito, A. Kawasumi, H. Hara, and S. Fujita. A 3.3ns-access-time 71.2uW/MHz 1Mb embedded STT-MRAM using physically eliminated read-disturb scheme and normally-off memory architecture. In *Solid-State Circuits Conference (ISSCC), 2015 IEEE International*, pages 1–3, Feb 2015.

[90] G. Oh, S. Kim, S.-W. Lee, and B. Moon. SQLite Optimization with Phase Change Memory for Mobile Applications. *Proc. VLDB Endow.*, 8(12):1454–1465, Aug. 2015.

[91] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The Log-Structured Merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.

[92] Berkeley DB. http://www.oracle.com/technology/products/berkeley-db/index.html.

[93] D. Park and D. Shin. iJournaling: Fine-Grained Journaling for Improving the Latency of Fsync System Call. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 787–798, Santa Clara, CA, 2017. USENIX Association.

[94] S. Pelley, P. M. Chen, and T. F. Wenisch. Memory persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture*, ISCA '14, pages 265–276, Piscataway, NJ, USA, 2014. IEEE Press.

[95] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Crafting Crash-Consistent Applications. In *11th USENIX Symposium on Operating*

Systems Design and Implementation (OSDI 14), pages 433–448, Broomfield, CO, Oct. 2014. USENIX Association.

[96] PMEM: the persistent memory driver + ext4 direct access (DAX). https://github.com/01org/prd.

[97] pmem.io. NVM Library, 2017. http://pmem.io/nvml.

[98] pmem.io. Persistent Memory Development Kit, 2017. http://pmem.io/pmdk.

[99] Persistent Memory File System. https://github.com/linux-pmfs/pmfs.

[100] Linux POSIX file system test suite. https://lwn.net/Articles/276617/.

[101] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 24–33, New York, NY, USA, 2009. ACM.

[102] S. Raoux, G. Burr, M. Breitwisch, C. Rettner, Y. Chen, R. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L. Lung, and C. Lam. Phase-change random access memory: A scalable technology. *IBM Journal of Research and Development*, 52(4.5):465–479, July 2008.

[103] redislabs. Redis, 2017. https://redis.io.

[104] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)*, 10(1):26–52, 1992.

[105] M. Rosenblum and J. K. Ousterhout. The Design and Implementation of a Log-structured File System. *ACM Trans. Comput. Syst.*, 10(1):26–52, Feb. 1992.

[106] A. Sainio. NVDIMM: Changes are Here So What's Next? In *In-Memory Computing Summit 2016*, 2016.

[107] R. Santana, R. Rangaswami, V. Tarasov, and D. Hildebrand. A Fast and Slippery Slope for File Systems. In *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, pages 5:1–5:8, New York, NY, USA, 2015. ACM.

[108] M. Satyanarayanan, H. H. Mashburn, P. Kumar, D. C. Steere, and J. J. Kistler. Lightweight recoverable virtual memory. In *SOSP '93: Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 146–160, New York, NY, USA, 1993. ACM.

[109] B. Schroeder, S. Damouras, and P. Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 6–6, Berkeley, CA, USA, 2010. USENIX Association.

[110] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an mttf of 1,000,000 hours mean to you? In *USENIX Conference on File and Storage Technologies (FAST)*, 2007.

[111] B. Schroeder, R. Lagisetty, and A. Merchant. Flash Reliability in Production: The Expected and the Unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 67–80, Santa Clara, CA, 2016. USENIX Association.

[112] B. Schroeder, E. Pinheiro, and W.-D. Weber. DRAM Errors in the Wild: A Large-scale Field Study. In *ACM SIGMETRICS*, 2009.

[113] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti. An Empirical Study of File Systems on NVM. In *Proceedings of the 2015 IEEE Symposium on Mass Storage Systems and Technologies (MSST15)*, 2015.

[114] M. Seltzer, K. Bostic, M. K. McKusick, and C. Staelin. An implementation of a log-structured file system for UNIX. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, pages 3–3. USENIX Association, 1993.

[115] M. Seltzer, K. A. Smith, H. Balakrishnan, J. Chang, S. McMains, and V. Padmanabhan. File System Logging Versus Clustering: A Performance Comparison. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 21–21, Berkeley, CA, USA, 1995. USENIX Association.

[116] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh. Failure-Atomic Slotted Paging for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 91–104, New York, NY, USA, 2017. ACM.

[117] Shore-MT. http://research.cs.wisc.edu/shore-mt/.

[118] Y. Son, S. Kim, H. Y. Yeom, and H. Han. High-performance transaction processing in journaling file systems. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*, pages 227–240, Oakland, CA, 2018. USENIX Association.

[119] SQLite. SQLite, 2017. https://www.sqlite.org.

[120] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, 2015.

[121] V. Sridharan and D. Liberty. A study of DRAM failures in the field. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, Nov 2012.

[122] D. B. Strukov, G. S. Snider, D. R. Stewart, and R. S. Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[123] K. Suzuki and S. Swanson. The Non-Volatile Memory Technology Database (NVMDB). Technical Report CS2015-1011, Department of Computer Science & Engineering, University of California, San Diego, May 2015. http://nvmdb.ucsd.edu.

[124] Symas. Lightning Memory-Mapped Database (LMDB), 2017. https://symas.com/lmdb/.

[125] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter. How to Get More Value from Your File System Directory Cache. In *Proceedings of the 25th Symposium on Operating Systems Principles*, SOSP '15, pages 441–456, New York, NY, USA, 2015. ACM.

[126] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 18–32, New York, NY, USA, 2013. ACM.

[127] UEFI Forum. Advanced configuration and power interface specification, 2017. http://www.uefi.org/sites/default/files/resources/ACPI_6_2.pdf.

[128] S. Venkataraman, N. Tolia, P. Ranganathan, and R. Campbell. Consistent and durable data structures for non-volatile byte-addressable memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST '11, pages 5–5, San Jose, CA, USA, February 2011. USENIX Association.

[129] 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*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

[130] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *ASPLOS '11: Proceeding of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems*, New York, NY, USA, 2011. ACM.

[131] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. L. Moal, T. Bunker, J. Xu, S. Swanson, and Z. Bandić. DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST '14, pages 309–315, Santa Clara, CA, 2014. USENIX.

[132] J. Wang and Y. Hu. WOLF-A Novel Reordering Write Buffer to Boost the Performance of Log-Structured File Systems. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST '02, pages 47–60, Monterey, CA, 2002. USENIX.

[133] W. Wang, Y. Zhao, and R. Bunt. HyLog: A High Performance Approach to Managing Disk Layout. In *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, volume 4 of *FAST '04*, pages 145–158, San Francisco, CA, 2004. USENIX.

[134] M. Wilcox. Add support for NV-DIMMs to ext4, 2014. https://lwn.net/Articles/613384/.

[135] M. Wu and W. Zwaenepoel. eNVy: A Non-volatile, Main Memory Storage System. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-VI, pages 86–97, New York, NY, USA, 1994. ACM.

[136] X. Wu and A. L. N. Reddy. SCMFS: A File System for Storage Class Memory. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 39:1–39:11, New York, NY, USA, 2011. ACM.

[137] J. Xu and S. Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*, pages 323–338, Santa Clara, CA, Feb. 2016. USENIX Association.

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

[139] J. Yang, D. B. Minturn, and F. Hady. When poll is better than interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, pages 3–3, Berkeley, CA, USA, 2012. USENIX.

[140] J. Yang, Q. Wei, C. Chen, C. Wang, K. L. Yong, and B. He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *13th USENIX Conference on File and Storage Technologies*, FAST '15, pages 167–181, Santa Clara, CA, Feb. 2015. USENIX Association.

[141] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. End-to-end data integrity for file systems: A zfs case study. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, FAST'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association.

[142] Y. Zhang and S. Swanson. A Study of Application Performance with Non-Volatile Main Memory. In *Proceedings of the 2015 IEEE Symposium on Mass Storage Systems and Technologies (MSST15)*, 2015.

[143] J. Zhao, S. Li, D. H. Yoon, Y. Xie, and N. P. Jouppi. Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-46, pages 421–432, New York, NY, USA, 2013. ACM.

[144] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[145] R. Zwisler. Add support for new persistent memory instructions. https://lwn.net/Articles/619851/.