UNIVERSITY OF CALIFORNIA SAN DIEGO

Building Reliable Software for Persistent Memory

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

in

Computer Science (Computer Engineering)

by

Lu Zhang

Committee in charge:

Professor Steven Swanson, Chair
Professor Ranjit Jhala
Professor Paul Siegel
Professor Geoffrey Voelker
Professor Jishen Zhao

2019

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

_____

_____

_____

_____

_____

Chair

University of California San Diego

2019

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I want to thank my advisor Professor Steven Swanson for his support as the chair of my committee. He has guided me through the course of my Ph.D. research, and his advice have helped me overcome the challenges throughout the process. I also want to express my gratitude to Professor Ranjit Jhala, Professor Geoffrey Voelker, Professor Jishen Zhao, and Professor Paul Siegel for their valuable comments as my committee members.

I also want to thank all the members of the Non-volatile Systems Laboratory (NVSL) for the discussions about research ideas and insights. I am especially thankful to Jian Xu, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Andy Rudoff (Intel), and Haolan Liu for their help in the papers that we co-authored.

I heartfully thank my parents, wife, and son for their love and support that have given me the mental strength to go through this challenging journey.

Chapter 1 and Chapter 2 contain 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 has appeared in the Proceedings of the 26th ACM Symposium on Operating Systems Principles. The dissertation author is one of the two the primary contributors and second author of this paper. The materials are copyright ©2017 by Association for Computing Machinery.

Chapter 1 and Chapter 3 contain material from "Pangolin: A Fault-Tolerant Persistent Memory Programming Memory" by Lu Zhang and Steven Swanson, which has appeared in the proceedings of the 2019 USENIX Annual Technical Conference. The dissertation author is the primary investigator and first author of this paper. The materials are copyright ©2019 by USENIX Association.

Chapter 1 and Chapter 4 contain content from "Using Static Analysis to Find Non-Volatile Main Memory Programming Bugs" by Lu Zhang, Haolan Liu, Jishen Zhao, and Steven Swanson, which is submitted to the 25th ACM International Conference on Architectural Support for

Programming Languages and Operating Systems (ASPLOS'20). The dissertation author is the primary investigator and first author of this paper.

VITA

2007	B. S. in Electronics Engineering, Shandong University, Ji'nan
2009-2010	Graduate Student Researcher, IMEC, Leuven
2010	M. S. in Embedded Systems, Delft University of Technology, Delft
2010-2012	Hardware Engineer, Intel, Eindhoven
2012-2019	Graduate Student Researcher, University of California, San Diego
2015	Internship, Google, Mountain View
2019	Ph. D. in Computer Science (Computer Engineering), University of California, San Diego

PUBLICATIONS

Lu Zhang, Haolan Liu, Jishen Zhao and Steven Swanson. "Using Static Analysis to Find Non-Volatile Main Memory Programming Bugs", *Submitted to the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, Steven Swanson. "Basic Performance Measurements of the Intel Optane DC Persistent Memory Module", *arXiv article on Performance*, 2019.

Lu Zhang and Steven Swanson. "Pangolin: A Fault-Tolerant Non-Volatile Main Memory Programming Library", *2019 USENIX Annual Technical Conference (ATC)*, 2019.

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", *The 26th ACM Symposium on Operating Systems Principles (SOSP)*, 2017.

Moein Khazraee, Lu Zhang, Luis Vega, and Michael Bedford Taylor "Moonwalk: NRE Optimization in ASIC Clouds", *The 22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017.

Lu Zhang, Luis Vega and Michael Bedford Taylor. "Power Side Channels in Security ICs: Hardware Countermeasures", *arXiv article on Cryptography and Security*, 2016.

ABSTRACT OF THE DISSERTATION

Building Reliable Software for Persistent Memory

by

Lu Zhang

Doctor of Philosophy in Computer Science (Computer Engineering)

University of California San Diego, 2019

Professor Steven Swanson, Chair

Persistent memory (PMEM) technologies preserve data across power cycles and provide performance comparable to DRAM. In emerging computer systems, PMEM will operate on the main memory bus, becoming byte-addressable and cache-coherent. One key feature enabled by persistent memory is to allow software directly accessing durable data using the CPU's load/store instructions, even from the user-space.

However, building reliable software for persistent memory faces new challenges from two aspects: crash consistency and fault tolerance. Maintaining crash consistency requires the ability to recover data integrity in the event of system crashes. Using load/store instructions to access durable data introduces a new programming paradigm, that is prone to new types of

programming errors. Fault tolerance involves detecting and recovering from persistent memory errors, including memory media errors and scribbles from software bugs. With direct access, file systems and user-space applications have to explicitly manage these errors, instead of relying on convenient functions from lower I/O stacks.

We identify unique challenges in improving reliability for PMEM-based software and propose solutions. The thesis first introduces NOVA-Fortis, a fault-tolerant PMEM file system incorporating replication, checksums, and parity for protecting the file system's metadata and the user's file data. NOVA-Fortis is both fast and resilient in the face of corruption due to media errors and software bugs.

NOVA-Fortis only protects file data via the `read()` and `write()` system calls. When an application memory-maps a PMEM file, NOVA-Fortis has to disable file data protection because `mmap()` leaves the file system unaware of updates made to the file. For protecting memory-mapped PMEM data, we present Pangolin, a fault-tolerant persistent object library to protect an application's objects from persistent memory errors.

Writing programs to ensure crash consistency in PMEM remains challenging. Recovery bugs arise as a new type of programming error, preventing a post-crash PMEM file from recovering to a consistent state. Thus, we design two debugging tools for persistent memory programming: PmemConjurer and PmemSanitizer. PmemConjurer is a static analyzer using symbolic execution to find recovery bugs without running a compiled program. PmemSanitizer contains compiler instrumentation and run-time recovery bug analysis, compensating PmemConjurer with multi-threading support and store reordering tests.

Chapter 1

Introduction

Non-volatile memory (NVM) technologies (e.g., battery-backed NVDIMMs [70] and 3D XPoint [69]) provide data persistence with performance comparable to DRAM. Commercial NVM products that can operate on the system's main memory bus alongside DRAMs have debuted[1]. We refer to them as non-volatile main memory (NVMM) or persistent memory (PMEM). They are byte-addressable and cache-coherent. Thus, one key feature of PMEM is to allow software accessing durable data using CPU's load/store instructions, even from the user-space, similar to how software accesses DRAM nowadays. The combination of PMEM and DRAM enables hybrid memory systems that offer the promise of dramatic increases in storage performance and a more flexible programming model.

While the performance advantages of PMEM are enticing, for applications dealing with mission-critical data, a storage system's reliability is at least as important as performance. Building reliable software for persistent memory faces new challenges from two aspects: crash consistency and fault tolerance. Crash consistency is the ability to recover data consistency in the events of system crashes. Using load/store instructions and cache-line operations to access durable data and manage persistence introduces a new programming paradigm and that is prone to new

[1] Intel has released their Optane DC Persistent Memory Modules in April, 2019.

types of programming errors. Fault tolerance involves detecting and recovering from persistent memory faults, including media errors and scribbles from software bugs. With direct access using load/store instructions, file systems and user-space applications have to explicitly manage these errors, instead of relying on convenient functions from lower I/O stacks. Without enhancing applications with fault-tolerance capabilities and testing them for crash-consistency bugs, using persistent memory to store critical data is not reliable.

In this thesis, we investigate software design and debugging techniques that would enhance the reliability of PMEM-based applications. We first focus on how we should redesign existing software components (e.g., file systems) and reuse their interfaces to exploit PMEM's performance characteristics and accommodate the fault-tolerance challenges it presents. Chapter 2 explains the key differences between conventional block-based file systems and PMEM file systems from a reliability perspective. Then, it introduces NOVA-Fortis, a fast and resilient PMEM file system incorporating replication, checksums, and per-page parity for protecting the file system's metadata and the user's file data.

Besides using a file system interface, persistent memory also enables applications to operate on durable data directly from the user-space, bypassing the file system. They can achieve this by memory-mapping (e.g., using `mmap()` in Linux) a file in a PMEM-enabled file system such as NOVA-Fortis. Once the operating system creates address mapping, the user-space application can access the file's content without going through the file system. This direct-access mode from user-space is termed *DAX*. There is a fundamental conflict between DAX-style `mmap()` and file system-based fault tolerance: By design, DAX-`mmap()` leaves the file system unaware of updates made to the file, making it impossible for the file system to update the protection data for the file. NOVA-Fortis' solution is to disable file data protection while the file is mapped and restore it afterward. This provides well-defined protection guarantees but leaves file data unprotected when it is memory-mapped. Moving fault-tolerance to user-space programming libraries solves this problem, but presents challenges since it requires integrating fault tolerance into persistent object

libraries that manage potentially millions of small, heterogeneous objects.

To provide fault tolerance for future DAX-style, object-oriented applications, Chapter 3 presents Pangolin, a persistent object library that uses a combination of metadata replication, per-object checksums, parity, and micro-buffering to protect an application's objects from both media errors and corruption due to software bugs. Pangolin presents programming interfaces similar to `libpmemobj`, a persistent object library maintained by Intel, and automatically enables fault-tolerance features with its function calls. Compared to `libpmemobj`, performance is similar, and Pangolin provides stronger protection, online recovery, and greatly reduced storage overhead (1% instead of 100%).

Writing programs to ensure crash consistency in PMEM remains challenging. We define *recovery bugs* as PMEM-specific programming errors that may cause unrecoverable data inconsistency after a crash. Developing PMEM applications without recovery bugs requires programmers to carefully reason about when and in what order data becomes persistent during the program execution, explicitly insert cache-line flushing and memory ordering instructions at proper locations in the source code, and implement recovery methods to restore an PMEM image to a consistent state after a crash. Moreover, it also demands special memory management mechanisms (e.g., atomic, crash-consistent memory allocation) and transaction algorithms that are unique to PMEM programming. Adding the required functionality introduces pervasive changes to existing programming practices, and the subtleties involved open the door to a wide range of recovery bugs.

These challenges illustrate the need for effective recovery bug detecting tools. However, most of existing PMEM debugging tools, such as PMemCheck [45], PMReorder [81], and PMTest [60], share two major limitations: 1) lack of static analysis and 2) poor multi-threading support. Without static analysis, tools must rely on instrumenting a program and running dynamic analysis on a finite set of test cases. Therefore, the instrumentation effort and test case quality significantly limit the testing coverage. Moreover, existing tools often require programmers to

manually annotate the source code with testing constructs, steepening the learning curve, reducing code readability and maintainability, and reducing portability. Finally, current PMEM debugging tools [45, 60, 81] provide little or no support for inter-thread bug analysis. PMemCheck and PMReorder extend Valgrind [76], which serializes all threads with an emulated CPU, and does not consider interactions between threads. PMTest also lacks inter-thread analyses.

Chapter 4 describes PmemConjurer and PmemSanitizer to overcome the limitations of existing PMEM debugging tools. PmemConjurer is a static analyzer that employs symbolic execution to explore a program's control flow graph and search for recovery bugs. To support inter-thread analysis, PmemSanitizer provides compiler instrumentation that automatically injects dynamic diagnosis code into the program. An instrumented program will execute natively with threading, while PmemSanitizer's runtime library performs inter-thread analysis and store-reordering tests. The tools can detect many reproduced bugs in the PMDK[2] repository, and they also discover eight unknown bugs in the example programs of PMDK, a non-trivial amount given the limited availability of real-life persistent memory prgrams. The PMDK maintainers have accepted all of our patches to fix them.

Finally, Chapter 5 concludes the thesis by summarizing the three projects and highlighting their roles in real-life development of PMEM-based applications.

Acknowledgements

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 has appeared in the Proceedings of the 26th ACM Symposium on Operating Systems Principles. The thesis author is one of the two the primary contributors and second author of the paper.

[2]Persistent Memory Development Kit

This chapter contains material from "Pangolin: A Fault-Tolerant Persistent Memory Programming Memory" by Lu Zhang and Steven Swanson, which has appeared in the proceedings of the 2019 USENIX Annual Technical Conference. The thesis author is the primary investigator and first author of this paper.

This chapter contains material from "Using Static Analysis to Find Non-Volatile Main Memory Programming Bugs" by Lu Zhang, Haolan Liu, Jishen Zhao, and Steven Swanson, which is submitted to the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20). The thesis author is the primary investigator and first author of this paper.

Chapter 2

The NOVA-Fortis File System

Integrating NVMMs into computer systems presents a host of interesting challenges. The most pressing of these focus on how we should redesign existing software components (e.g., file systems) to accommodate and exploit the different performance characteristics, interfaces, and semantics that NVMMs provide.

Several groups have proposed new file systems [13, 104, 22] designed specifically for NVMMs and several Windows and Linux file systems now include at least rudimentary support for them [35, 63, 64]. These file systems provide significant performance gains for data access and support "direct access" (or DAX-style) `mmap()` that allows applications to access a file's contents directly using load and store instructions, a likely "killer app" for NVMMs.

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.

From a reliability perspective, there are four key differences between conventional block-

based file systems and NVMM file systems.

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, something that 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" – 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.

We explore the impact of these differences on file system reliability mechanisms by building *NOVA-Fortis*, an NVMM file system that adds fault-tolerance to NOVA [104] by incorporating replication, checksums, and RAID-style parity protection.

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.

This chapter makes 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 describe a fast replication algorithm called *Tick-Tock* for NVMM data structures that combines atomic update with error detection and recovery.

3. We adapt state-of-the-art techniques for 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× on average. It outperforms reliable, block-based file systems running on NVMM by 3× on average.

To describe NOVA-Fortis, we start by providing a brief primer on NVMM's implications for system designers, existing NVMM file systems, key issues in file system reliability, and the NOVA filesystem (Section 2.1). Then, we describe NOVA-Fortis' (meta)data protection mechanisms in 2.2. Section 2.3 evaluates these mechanisms, and Section 2.4 presents our conclusions.

2.1 Background

NOVA-Fortis targets memory systems that include emerging non-volatile memory technologies along with DRAM. This section first provides a brief survey of NVMM technologies and the opportunities and challenges they present. Then we describe recent work on NVMM file systems and discuss key issues in file system reliability. Finally, we provide a brief primer on NOVA.

2.1.1 Non-volatile Memory Technologies

Modern server platforms have support NVMM in form of NVDIMMs [70, 39] and the Linux kernel includes low-level drivers for identifying physical address regions that are non-volatile, etc. NVDIMMs are commercially available from several vendors in form of DRAM DIMMs that can store their contents to an on-board flash-memory chip in case of power failure with the help of super-capacitors.

NVDIMMs that dispense with flash and battery backup are expected to appear in systems soon. Phase change memory (PCM) [57, 84], resistive RAM (ReRAM) [25, 98], and 3D XPoint memory technology [69] are denser than DRAM, and may enable very large, non-volatile main memories. Their latencies are longer than DRAM, however, making it unlikely that they will

fully replace DRAM as main memory. Other technologies, such as spin-torque transfer RAM (STT-RAM) [51] are faster, but less dense and may find other roles in future systems (e.g., as non-volatile caches [110]). These technologies are all under active development and knowledge about their reliability and performance is evolving rapidly.

The 3D XPoint memory technology has already appeared [43]. In addition, all major memory manufacturers have candidate technologies that could compete with 3D XPoint. Consequently, we expect hybrid volatile/non-volatile memory hierarchies to become common in large systems.

Allowing programmers to build useful data structures with NVMMs requires CPUs to make guarantees about when stores become persistent that programmers can use to guarantee consistency after a system crash [6, 2]. Without these guarantees it is impossible to build data structures in NVMM that are reliable in the face of power failures [103, 12, 102].

NVMM-aware systems provide some form of *persist barrier* that allows programmers to ensure that earlier stores become persistent before later stores. Researchers have proposed several different kinds of persist barriers [13, 78, 53].

For example, under x86 a persist barrier comprises a `clflush` or `clwb` [41] instruction to force cache lines into the system's "persistence domain" and a conventional memory fence to enforce ordering. Once a store reaches the persistence domain, the system guarantees it will reach NVMM, even in the case of crash. NOVA-Fortis and other NVMM file systems assume that these or similar instructions are available.

2.1.2 NVMM File Systems and DAX

Several groups have designed NVMM file systems [13, 21, 22, 63, 104] that address the unique challenges that NVMMs' performance and byte-addressible interface present. One of these, NOVA, is the basis for NOVA-Fortis, and we describe it in more detail in Section 2.1.4.

NVMMs' low latencies make software efficiency much more important than in block-

based storage devices [7, 9].

NVMM-aware CPUs provide a load/store interface with atomic 8-byte operations rather than a block-based interface with block- or sector-based atomicity. NVMM file systems can use these atomic updates to implement features such as complex atomic data and metadata updates, but doing so requires different data structures and algorithms than block-based file systems have employed.

Since NVMMs reside on the processor's memory bus, applications should be able to access them directly via loads and stores. NVMM file systems provide this ability via direct access (or "DAX"). DAX allows read and write system calls to bypass the page cache and access NVMM directly. DAX `mmap()` maps the NVMM physical pages that hold a file directly into an application's address space, so the application can access and modify file data with loads and stores and use persist barriers to enforce ordering constraints. File systems for both Windows [35] and Linux [63, 64] support DAX `mmap()`.

DAX `mmap()` is a likely "killer app" for NVMMs since it 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 [12, 103, 82] to manage it.

Building persistent data structures that are robust in the face of system and application failures is a difficult programming challenge and the first inspiration for the *caveat DAXor* principle. Building data structures in NVMM requires the application (or library) to take responsibility for allocating and freeing persistent memory and avoiding a host of new bugs that can arise in memory systems that combine persistent and transient state [12].

Applications are also responsible for enforcing ordering relationships between stores to ensure that the application can recover from an unexpected system failure. Conventional `mmap()`-based applications use `msync()` for this purpose. `msync()` works with DAX `mmap()`,

but it is expensive, non-atomic, and only operates on pages. Persist barriers are much faster than `msync()` and better-suited to building complex data structures, but they are more difficult to use correctly.

2.1.3 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).

File systems have devised a variety of different techniques to guarantee system-wide consistency of file system data structures, including journaling [100, 22], copy-on-write [8, 86, 13] and log-structuring [87, 88].

Highly-reliable file systems like ZFS [8] and Btrfs [86] 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.

Data Corruption

File systems are subject to a wide array of data corruption mechanisms 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 [91, 5, 90], SSDs [68, 75] and DRAM-based memories [92, 97]. The results of DRAM-based studies 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) (see Section 2.2.1).

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" [54] 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.

We discuss other prior work on file system reliability as it relates to NOVA-Fortis in Section 2.2.8.

2.1.4 The NOVA File System

NOVA-Fortis is based on the NOVA NVMM file system [104]. NOVA's initial design focused on two goals: Fully exposing the performance that NVMMs offer and providing very strong consistency guarantees – all operations in NOVA are atomic. Below, we describe the features of NOVA that are most relevant to our description of NOVA-Fortis.

Each inode in a NOVA file system has a private log that records changes to the inode. Figure 2.1 illustrates the relationship between an inode, its log, and file data. NOVA stores the log as a linked list of 4 KB NVMM pages, so logs are non-contiguous. To perform a file operation that affects a single inode, NOVA appends a log entry to the log and updates the pointer to the log's tail using an atomic, 64-bit store.

For writes, NOVA uses copy-on-write, allocating new pages for the written data. The log entry for a write holds pointers to the newly written pages, atomically replacing them in the file. NOVA immediately reclaims the resulting stale pages.

For complex file operations that involve multiple inodes (e.g., moving a file between

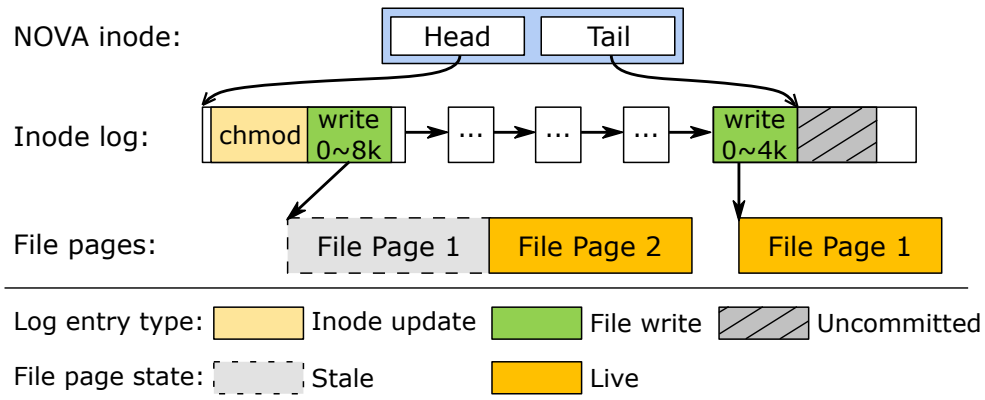

Figure 2.1: NOVA inode log structure - A NOVA inode log records changes to the inode (e.g., the mode change and two file write operations shown above). NOVA stores file data outside the log.

directories), NOVA uses small, fixed-size journals (one per core) to store new tail pointers for all the inodes and update them atomically.

NOVA periodically performs garbage collection on the inode logs by scanning and compacting the log. Since the logs do not contain file data, they are shorter and garbage collection is less critical than in a conventional log-structured file system.

To maximize concurrency, NOVA uses per-CPU structures in DRAM to allocate NVMM pages and inodes. It also caches inode metadata in DRAM to minimize accesses to NVMM (which is projected to be slower than DRAM), and uses one DRAM-based radix tree per file to map file offsets to NVMM pages.

NOVA divides the allocatable NVMM into multiple regions, one region per CPU core. A per-core allocator manages each of the regions, minimizing contention during memory allocation.

After a system crash, NOVA must scan all the logs to rebuild the memory allocator state. Since, there are many logs, NOVA aggressively parallelizes the scan. Recovering a 50 GB NOVA file system takes just over 1/10th of a second [104].

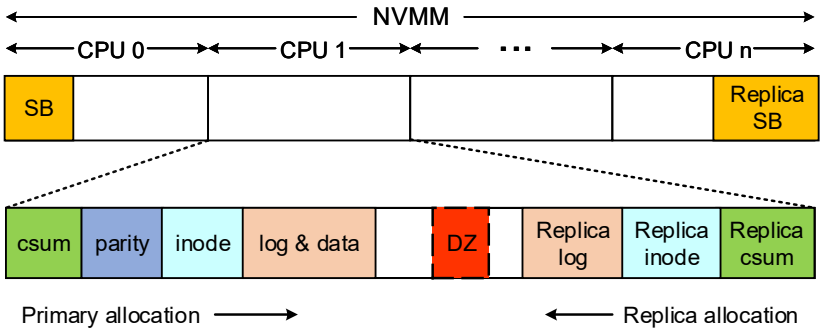

Figure 2.2: NOVA-Fortis space layout - NOVA-Fortis' per-core allocators satisfy requests for primary and replica storage from different directions. They also store data pages and their checksum and parity pages separately.

2.2 Handling Data Corruption in NOVA-Fortis

Like all storage media, NVMM is subject to data and metadata corruption from media failures and software bugs. 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.

2.2.1 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.

This section describes the interface that recent Linux kernels (e.g., Linux 4.10) and Intel

processors provide via the PMEM low-level NVDIMM driver. Porting NOVA-Fortis to other architectures or operating systems may require NOVA-Fortis to adopt a different approach to error detection.

NOVA-Fortis assumes the memory system provides ECC for NVMM that is similar to (or perhaps stronger than) the single-error-correction/double-error-detection (SEC-DED) scheme that conventional DRAM uses. We assume the controller transparently corrects correctable errors, and silently returns invalid data for undetectable errors.

For detectable but uncorrectable errors, Intel's Machine Check Architecture (MCA) [40] 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. Intel processors do not provide a mechanism for detecting store failures, and the memory controller transparently maps around 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 [101].

2.2.2 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.

2.2.3 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 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 2.2 shows the checksum and parity layouts in the NVMM.

When NOVA-Fortis performs a read, it first copies each strip of data into DRAM using `memcpy_mcsafe()`, and calculates its checksum. If this checksum matches either stored copy of the checksum, NOVA-Fortis concludes the file data is correct and updates the mismatched copy of the checksum if needed.

If neither of the checksums match the read data or a media error occurs, NOVA-Fortis attempts to restore the strip using RAID-4 parity, and uses the strip's checksum to determine if the recovery succeeded. If no other strip in the page is corrupt, recovery will succeed and NOVA-Fortis restores the target strip and its checksums. If more than one strip is corrupt, the file page is lost and the read fails.

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 detecting and recovering from both media errors (which appear as `SIGBUS` in user space)

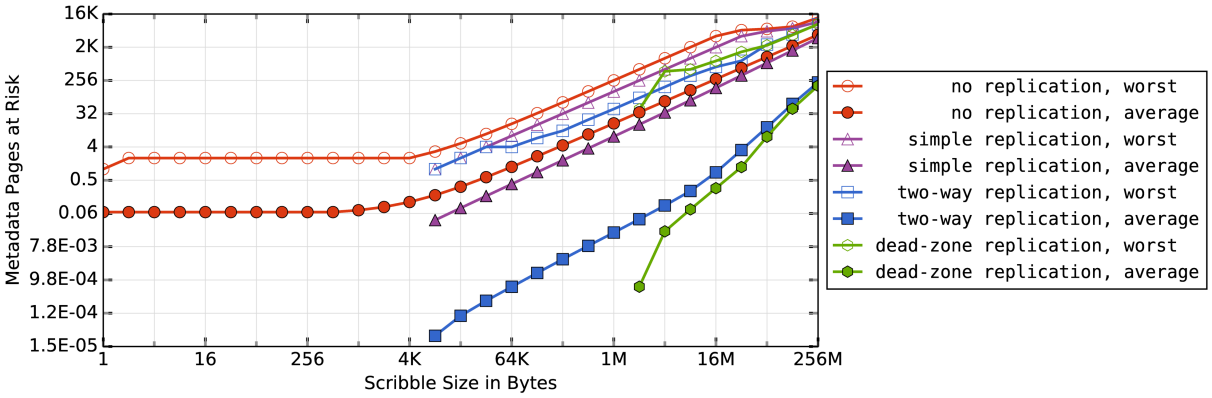

Figure 2.3: Scribble size and metadata bytes at risk - Replicating metadata pages and taking care to allocate the replicas separately improves resilience to scribbles.

and scribbles. This is an interesting challenge but beyond the scope of this chapter. 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.

Design Decisions and Alternatives. Using RAID parity and checksums to protect file data is similar to the approach that ZFS [8] and IRON ext3 [83] take, but we store parity for each page rather than one parity page per file [83], and we maintain per-strip checksums instead of a single checksum for a whole stripe [8].

NOVA-Fortis chooses RAID-4 over RAID-5 because RAID-5 intermingles parity and data and would make DAX `mmap()` impossible since parity bits would end up in the application's address space.

Alternately, NOVA-Fortis could rely on RAIM [65], Chipkill [38], or other advanced ECC mechanisms to protect file data. These techniques would improve reliability, but they are not universally available and cannot protect against scribbles.

2.2.4 Minimizing Vulnerability to Scribbles

Scribbles pose significant risk to NOVA-Fortis' data and metadata, since a scribble can impact large, continuous regions of memory. We are not aware of any systematic study of the prevalence of these errors, but scribbles, lost, and misdirected writes are well-known culprits for file system corruption [54, 109, 26]. In practice, we expect that smaller scribbles are more likely than larger ones, in part since the bugs that result in larger scribbles would be more severe and more likely to be found and fixed.

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.

NOVA-Fortis packs log entries in to log pages, and it must scan the page to recognize each entry. Without protection, losing a single byte can corrupt a whole page. For replicated log pages, a scribble that spans both copies of a byte will corrupt the page. To measure the BAR for a scribble of size N we measure the number of pages each possible N-byte scribble would destroy in an aged NOVA-Fortis file system.

Figure 2.3 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" separates the primary and replica by preferring low addresses for the primary and high addresses for the replica; and "dead-zone replication" extends "two-way" by enforcing a 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. Figure 2.2 shows an example of NOVA-Fortis two-way allocator with dead zone separation. For each pair of mirrored pages, the dead zone forbids the primary and replica from becoming too close, but data pages can reside between them.

To stress the allocator's ability to place the primary and replica far apart, we aged the file system by spawning multiple, multi-threaded, Filebench workloads. When each workload finishes, we remove about half of its files, and then restart the workload. We continue until the file system is 99% full.

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 tries to allocate the primary and replica farther apart, and it 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, but smaller scribbles do not. 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.

2.2.5 Preventing Scribbles

The mechanisms described above let NOVA-Fortis detect and recover from data corruption. NOVA-Fortis can borrow a technique from WAFL [54] and PMFS [22] to prevent scribbles by marking 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.

The WP approach only protects against scribbles from other kernel code. It cannot prevent NOVA-Fortis from corrupting its own data by performing "misdirected writes," a common source of data corruption in file systems [3].

2.2.6 Relaxing Data and Metadata Protection

Many existing file systems can trade off reliability for improved performance (e.g., the data journaling option in Ext4). NOVA-Fortis can do the same: It provides a *relaxed mode* that relaxes atomicity constraints on file data and metadata.

In relaxed mode, write operations modify existing data directly rather than using copy-on-write, and metadata operations modify the most recent log entry for an inode directly rather than appending a new entry. Relaxed mode guarantees metadata atomicity by journaling the modified pieces of metadata. These changes improve performance and we evaluate their impact in Section 2.3.

2.2.7 Protecting DRAM Data Structures

Corruption of DRAM data structures can result in file system corruption [19, 109], 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 allocator state is and

exception and it is vulnerable to corruption. The allocator protects the address and length of each free region with checksums, but it does not protect the pointers that make up the red-black tree that holds them, since we use the kernel's generic red-black tree implementation.

2.2.8 Related and Future Work

Below, we describe proposed "best practices" for file system design and how NOVA-Fortis addresses them. Then, we describe areas of potential improvement for NOVA-Fortis.

Is NOVA-Fortis Ferrous?

InteRnally cONistent (IRON) file systems [83] provide a set of principles to that lead to improved reliability. We designed NOVA-Fortis to embody these principles:

Check error codes. Uncorrectable ECC errors are the only errors that the NVMM memory system delivers to software (i.e., via MCEs). NOVA-Fortis uses `memcpy_mcsafe()` for all NVMM loads and triggers recovery if it detects an MCE. NOVA-Fortis also interacts with the PMEM driver that provides low-level management of NVDIMMs. For these calls, we check and respond to error codes appropriately.

Report errors and limit the scope of failures. NOVA-Fortis reports all unrecoverable errors as `EIO` rather than calling `panic()`.

Use redundancy for integrity checks and distribute redundancy information. NOVA-Fortis' tick-tock replication scheme stores the checksum for each replica with the replica, but it is careful to allocate the primary and replica copies far from one another. Likewise, NOVA-Fortis stores the parity and checksum information for data pages separately from the pages themselves.

Type-aware fault injection. For testing, we built a NOVA-Fortis-specific error injection tool that can corrupt data and metadata structures in specific, targeted ways, allowing us to test NOVA-Fortis' detection and recovery mechanisms.

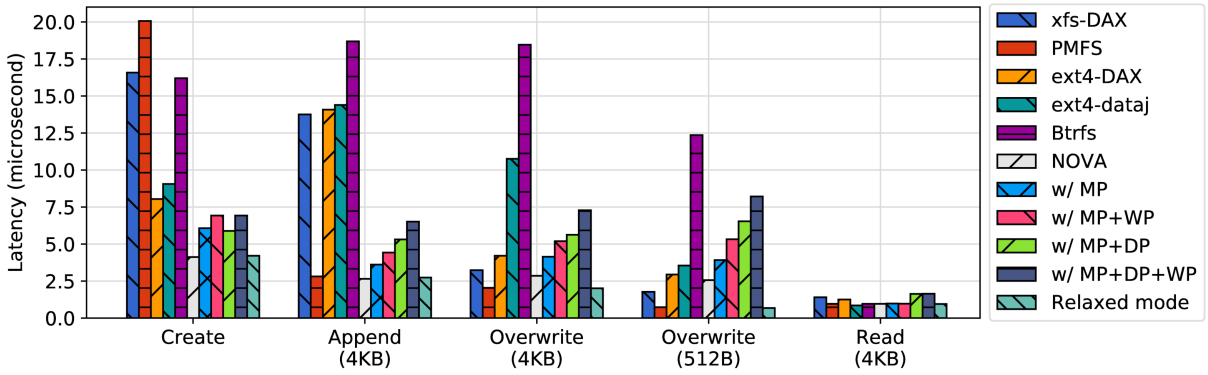

Figure 2.4: File operation latency - NOVA-Fortis' basic file operations are faster than competing file systems except in cases where the other file system provides weaker consistency guarantees and/or data protection.

Areas for Improvement

There are several additional steps NOVA-Fortis could take to further improve reliability. We do not expect any of them to have a large impact on performance or storage overheads.

Sector or block failures in disks are not randomly distributed [4], and errors in NVMM are also likely to exhibit complex patterns of locality [67, 97]. For instance, an NVMM chip may suffer from a faulty bank, row, or column, leading to a non-uniform error distribution. Or, an entire NVDIMM may fail.

NOVA-Fortis' allocator actively separates the primary and replica copies of metadata structures to eliminate *logical* locality, but it does not account for how the memory system maps physical addresses onto the physical memory. A layout-aware allocator could, for instance, ensure that replicas reside in different banks or different NVDIMMs.

NOVA-Fortis cannot keep running after an unrecoverable MCE (since they cause a `panic()`), but it could recover any corrupted data during recovery. The PMEM driver provides a list of poisoned PRs on boot, and NOVA-Fortis can use this information to locate and recover corrupted file data during mount. Without this step, NOVA-Fortis will still detect poisoned metadata, since reading from a poisoned PR results in a recoverable MCE, and NOVA-Fortis

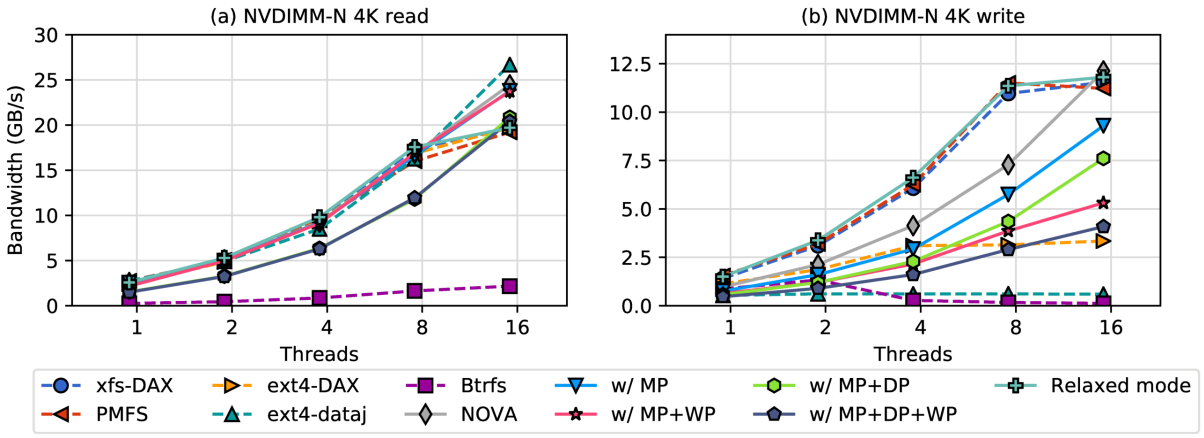

Figure 2.5: NOVA-Fortis random read/write bandwidth on NVDIMM-N - Read bandwidth is similar across all the file systems except Btrfs, and NOVA-Fortis' reliability mechanisms reduces its throughput by between 14% and 19%.

reads all metadata during recovery. Poisoned file data, however, could accumulate over multiple unrecoverable MCEs, increasing the chances of data loss.

Finally, NOVA-Fortis does not scrub data or metadata. PMEM detects media errors on reboot, but if a NOVA-Fortis file system ran continuously for a long time, undetected media errors could accumulate. Undetected scribbles to data and metadata can accumulate during normal operation and across reboots.

2.3 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.

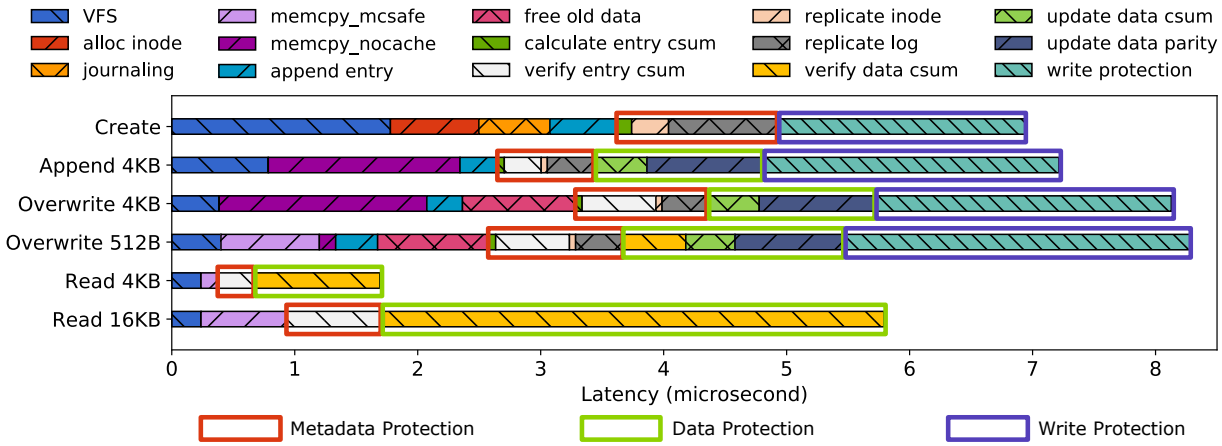

Figure 2.6: NOVA-Fortis latencies for NVDIMM-N - Protecting file data is usually more expensive than protecting metadata because the cost of computing checksums and parity for data scales with access size.

2.3.1 Experimental Setup

We implemented NOVA-Fortis for Linux 4.10, and use the Intel Persistent Memory Emulation Platform (PMEP) [22] to emulate different types of NVMM and study their effects on NVMM file systems. PMEP supports configurable latencies and bandwidth for the emulated NVMM, and emulates `clwb` instruction with microcode. In our tests we configure the PMEP with 32 GB of DRAM and 64 GB of NVMM, and choose two configurations for PMEP's memory emulation system: We use the same read latency and bandwidth as DRAM to emulate fast NVDIMM-N [89], and set read latency to 300 ns and reduce the write bandwidth to 1/8th of DRAM to emulate slower PCM. For both configurations we set `clwb` instruction latency to 40 ns.

We compare NOVA-Fortis against five other file systems. Ext4-DAX, xfs-DAX and PMFS are the three DAX-enabled file systems. None of them provides strong consistency guarantees (i.e., they do not guarantee that all operations are atomic), while NOVA does provide these guarantees. To compare to a file system with stronger guarantees, we also compare to ext4 in data journaling mode (ext4-dataj) and Btrfs running on the NVMM-based block device. Ext4 and xfs keep checksums for metadata, but they do not provide any recovery mechanisms for NVMM

media errors or protection against stray writes.

2.3.2 Performance Impacts

To understand the impact of NOVA-Fortis' reliability mechanisms, we begin by measuring the performance of individual mechanisms and basic file operations. Then we measure their impact on application-level performance.

We compare several version of NOVA-Fortis: We start with our baseline, NOVA, and add metadata protection ("MP"), data protection ("DP"), and write protection ("WP"). "Relaxed mode" weakens consistency guarantees to improve performance and provides no data protection (Section 2.2.6).

2.3.3 Microbenchmarks

We evaluate basic file system operations: `create`, 4 KB `append`, 4 KB `write`, 512 B `write`, and 4 KB `read`. Figure 2.4 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 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

Table 2.1: 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

verifies the data checksum before returning to the user.

Figure 2.6 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, 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 2.5 shows FIO [1] 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.

2.3.4 Macrobenchmarks

We use nine application-level workloads to evaluate NOVA-Fortis: Four Filebench [99] workloads (fileserver, varmail, webproxy, and webserver), two key-value stores (RocksDB [24] and MongoDB [71]), the Exim email server [23], SQLite [96], and TPC-C running on Shore-

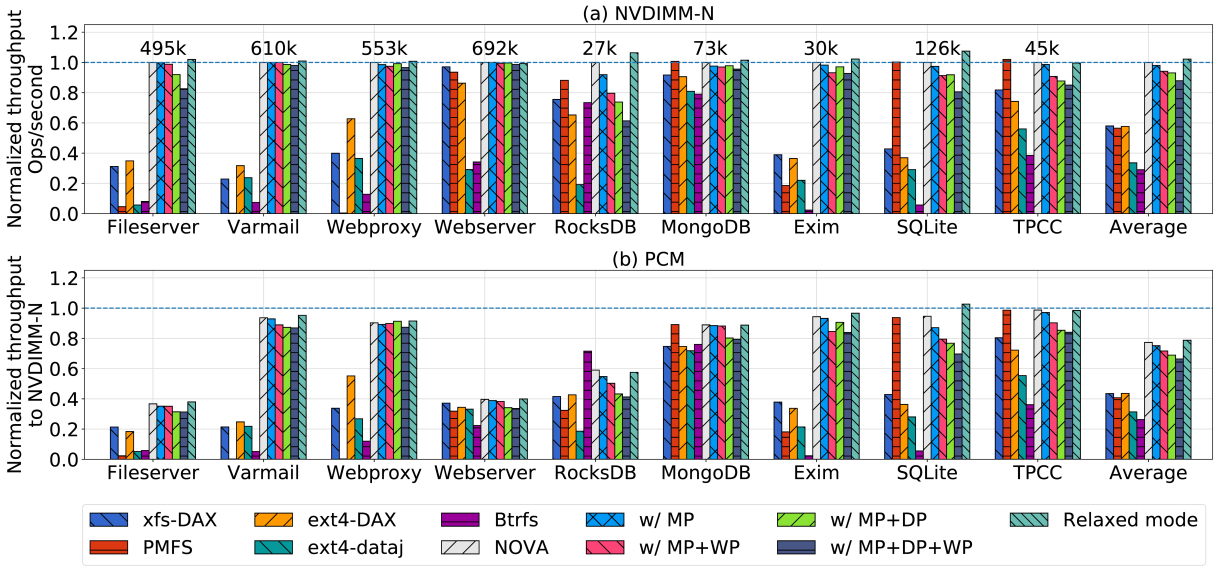

Figure 2.7: Application performance on NOVA-Fortis - Reliability overheads and the benefits of relaxed mode have less impact on applications than microbenchmarks (Figures 2.4 and 2.5).

MT [50]. Fileserver, varmail, webproxy and Exim are metadata-intensive workloads, while other workloads are data-intensive. Table 2.1 summarizes the workloads.

Figure 2.7 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 Rocks-DB 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 Rocks-DB 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.

2.3.5 NVMM Storage Utilization

Protecting data integrity introduces storage overheads. Figure 2.8 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%.

2.4 Conclusion

We have used NOVA-Fortis to explore the unique challenges that improving NVMM file system reliability presents. The solutions that NOVA-Fortis implements provide protection against media errors and corruption due to software errors.

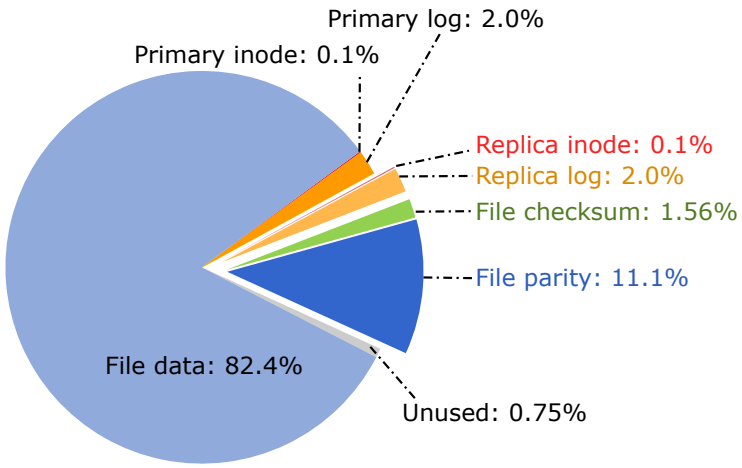

Figure 2.8: NVMM storage utilization - Extra storage required for reliability is highlighted to the right. Protecting data is more expensive that protecting metadata, consuming 12.7% of storage compared to just 2.1% for metadata.

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, despite our use of very fast (XOR parity) and hardware accelerated (CRC) mechanisms. 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.

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 has appeared in the Proceedings of the 26th ACM Symposium on Operating Systems Principles. The dissertation author is one of the two the primary contributors and second author of this paper.

Chapter 3

The Pangolin Library

A key feature of NVMM is support for direct access, or DAX, that lets applications perform loads and stores directly to a file that resides in NVMM. DAX offers the lowest-possible storage access latency and enables programmers to craft complex, customized data structures for specific applications. To support this model, researchers and industry have proposed various persistent object systems [12, 103, 82, 59, 36, 66, 48, 14].

Building persistent data structures presents a host of challenges, particularly in the area of crash consistency and fault tolerance. Systems that use NVMM must preserve crash-consistency in the presence of volatile caches, out-of-order execution, software bugs, and system failures. To address these challenges, many groups have proposed crash-consistency solutions based on hardware [74, 85, 73, 77], file systems [13, 22, 104, 105], user-space data structures and libraries [107, 93, 12, 103, 82, 36, 14], and languages [79, 18].

Fault tolerance has received less attention but is equally important: To be viable as an enterprise-ready storage medium, persistent data structures must include protection from data corruption. Intel processors report uncorrectable memory media errors via a machine-check exception and the kernel forwards it to user-space as a `SIGBUS` signal. To our knowledge, Xu *et al.* [105] were the first to design an NVMM file system that detects and attempts to recover from

these errors. Among programming libraries, only `libpmemobj` provides any support for fault tolerance, but it incurs 100% space overhead, only protects against media errors (not software "scribbles"), and cannot recover corrupted data without taking the object store offline.

Xu *et al.* also highlighted a fundamental conflict between DAX-`mmap()` and file system-based fault tolerance: By design, DAX-`mmap()` leaves the file system unaware of updates made to the file, making it impossible for the file system to update the redundancy data for the file. Their solution is to disable file data protection while the file is mapped and restore it afterward. This provides well-defined protection guarantees but leaves file data unprotected when it is in use.

Moving fault-tolerance to user-space NVMM libraries solves this problem, but presents challenges since it requires integrating fault tolerance into persistent object libraries that manage potentially millions of small, heterogeneous objects.

To satisfy the competing requirements placed on NVMM-based, DAX-mapped object store, a fault-tolerant persistent object library should provide at least the following characteristics:

1. **Crash-consistency**. The library should provide the means to ensure consistency in the face of both system failures and data corruption.

2. **Protection against media and software errors**. Both types of errors are real threats to data stored to NVMM, so the library should provide protection against both.

3. **Low storage overhead**. NVMM is expensive, so minimizing storage overhead of fault tolerance is important.

4. **Online recovery**. For good availability, detection and recovery must proceed without taking the persistent object store offline.

5. **High performance**. Speed is a key benefit of NVMM. If fault-tolerance incurs a large performance penalty, NVMM will be much less attractive.

6. **Support for diverse objects**. A persistent object system must support objects of size ranging from a few cache lines to many megabytes.

This chapter describes *Pangolin*, the first persistent object library to satisfy all these criteria. Pangolin uses a combination of parity, replication, and object-level checksums to provide space-efficient, high-performance fault tolerance for complex NVMM data structures. Pangolin also introduces a new technique for accessing NVMM called *micro-buffering* that simplifies transactions and protects NVMM data structures from programming errors.

We evaluate Pangolin using a suite of benchmarks and compare it to `libpmemobj`, a persistent object library that offers a simple replication mode for fault tolerance. Compared to `libpmemobj`, performance is similar, and Pangolin provides stronger protection, online recovery, and greatly reduced storage overhead (1% instead of 100%).

The rest of this chapter is organized as follows: Section 3.1 provides a primer on NVMM programming and NVMM error handling in Linux. Section 3.2 describes how Pangolin organizes data, manages transactions, and detects and repairs errors. Section 3.3 presents our evaluations. Section 3.4 and Section 3.5 discusses other design options and related works, respectively. Finally, Section 3.6 concludes.

3.1 Background

Pangolin lets programmers build fault-tolerant, crash-consistent data structures in NVMM. This section first introduces NVMM and the DAX mechanism applications use to gain direct access to persistent data. Then, we describe the NVMM error handling mechanisms that Intel processors and Linux provide. Finally, we provide a brief primer on NVMM programming using `libpmemobj` [82], the library on which Pangolin is based.

3.1.1 Non-volatile Main Memory and DAX

Several technologies are poised to make NVMM common in computer systems. 3D XPoint [69] is the closest to wide deployment. Phase change memory (PCM), resistive RAM (ReRAM), and spin-torque transfer RAM (STT-RAM) are also under active development by memory manufacturers. Flash-backed DRAM is already available and in wide use. Linux and Windows both have support for accessing NVMM and using it as storage media.

The performance and cost parameters of NVMM lie between DRAM and SSD. Its write latency is longer than DRAM, but it will cost less per bit. From the storage perspective, NVMM is faster but more expensive than SSD.

The most efficient way to access NVMM is via direct access (DAX) [42] memory mapping (i.e., DAX-`mmap()`). To use DAX-`mmap()`, applications map pages of a file in an NVMM-aware file system into their address space, so the application can access persistent data from the user-space using load and store instructions, without the file system intervening.

3.1.2 Handling NVMM Media Errors

To recover from data corruption, Pangolin relies on error detection and media management facilities that the processor and operating system provide together. Below, we describe these facilities available on Intel and Linux platforms. Windows provides similar mechanisms.

Hardware Error Correction. Memory controllers for commercially available NVMMs (i.e., battery-backed DRAM and 3D XPoint) implement error-correction code (ECC) in hardware to detect and correct media errors when they can, and they report uncorrectable (but detectable) errors with a machine check exception (MCE) [41] that the operating system can catch and attempt to handle.

Pangolin provides a layer of protection in addition to the ECC hardware provides, but it does not require hardware ECC. Pangolin uses checksums to detect errors that hardware cannot

detect. This mechanism also catches software bugs (which are invisible to hardware ECC). ECC does, however, improve performance by transparently handling many media errors.

Regardless of the ECC algorithm hardware provides, field studies of DRAM and SSDs [92, 37, 97, 90, 68, 75] have shown that detectable but uncorrectable media errors occur frequently enough to warrant additional software protection. Furthermore, file systems [109, 105, 54] apply checksums to their data structures to protect against scribbles.

Repairing Errors. When the hardware detects an uncorrectable error, the Linux kernel marks the region surrounding the failed load as "poisoned," and future loads from the region will fail with a bus error. Pangolin assumes an error poisons a 4 KB page since Linux currently manages memory failures at page granularity.

If a running application causes an MCE (by loading from a poisoned page), the kernel sends it a `SIGBUS` and the application can extract the affected address from the data structure describing the signal.

The software can repair the poisoned page by writing new data to the region. In response, the operating system and NVDIMM firmware work together to remap the poisoned addresses to functioning memory cells. The details of this process are part of the Advanced Configuration and Power Interface (ACPI) [101] for NVDIMMs.

Recent kernel patches [15, 16, 17, 62] and NVMM library [82] provide utilities for user-space applications to restore lost data by re-writing affected pages with recovered contents (if available).

3.1.3 NVMM Programming

In this section, we describe `libpmemobj`'s programming model. `Libpmemobj` is a well-supported, open-source C library for programming with DAX-mapped NVMM. It provides facilities for memory management and software transactions that let applications build a persistent object store. Pangolin's interface and implementation are based on `libpmemobj` from PMDK.

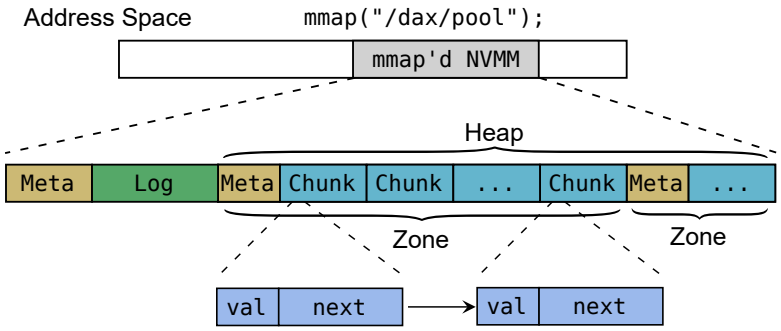

Figure 3.1: DAX-mapped NVMM as an object store - `Libpmemobj` divides the mapped space into zones and chunks for memory management.

Linux exposes NVMM to the user-space as memory-mapped files (Figure 3.1). `Libpmemobj` (and Pangolin) refer to the mapped file as a *pool* of persistent objects. Each pool spans a continuous range of virtual addresses.

```
1  PMEMobjpool *pool = pmemobj_open("/dax/pool");
2  ...
3  struct node *n = pmemobj_direct(node_oid);
4  n->val = value;
5  pmemobj_persist(pool, &n->val, 8);
6  ...
7  TX_BEGIN(pool) {
8    n = pmemobj_direct(node_oid);
9    pmemobj_tx_add_range(node_oid, 0, sizeof(*n));
10   n->next = pmemobj_tx_alloc(...);
11 } TX_ONABORT {
12   /* handling transaction aborts */
13 } TX_END
14 ...
15 pmemobj_close(pool);
```

Listing 3.1: A `libpmemobj` program - First modify a node value in a linked list, and later allocate and link a new node from the pool.

Within a pool, `libpmemobj` reserves a metadata region that contains information such as the pool's identification (64-bit UUID) and the offset to a "root object" from which all other live objects are reachable. Next, is an area reserved for transaction logs. `Libpmemobj` uses redo logging for its metadata updates and undo logging for application object updates. Transaction logs reside in one of two locations depending on their sizes. Small log entries live in the provisioned "Log" region, as shown in Figure 3.1. Large ones overflow into the "Heap" storage area.

The rest of the pool is the persistent heap. `Libpmemobj`'s NVMM allocator (a persistent variant of `malloc`/`free`) manages it. The allocator divides the heap's space into several "zones" as shown in Figure 3.1. A zone contains metadata and a sequence of "chunks." The allocator divides up a chunk for small objects and coalesces adjacent chunks for large objects. By default, a zone is 16 GB, and a chunk is 256 KB.

Listing 3.1 presents an example to highlight the key concepts of NVMM programming. The code performs two independent operations on a persistent linked list: one is to modify a node's value, and another is to allocate and link a new node.

This example demonstrates two styles of crash-consistent NVMM programming: *atomic*-style (lines 3-5) for a simple modification that is 8 bytes or smaller, and *transactional*-style (lines 7-13) for arbitrary-sized NVMM updates.

Building data structures in NVMM using `libpmemobj` (or any other persistent object library) differs from conventional DRAM programming in several ways:

Memory Allocation. `Libpmemobj` provides crash-consistent NVMM allocation and deallocation functions: `pmemobj_tx_alloc`/`pmemobj_tx_free`. They let the programmer specify object type and size to allocate and prevent orphaned regions in the case of poorly-time crashes.

Addressing Scheme. Persistent pointers within a pool must remain valid regardless of at what virtual address the pool resides. `Libpmemobj` uses a `PMEMoid` data structure to address an object within a pool. It consists of a 64-bit file ID and a 64-bit byte offset relative to the start of the file. The `pmemobj_direct()` function translates a `PMEMoid` into a native pointer for use in

load or store instructions.

Failure-atomic Updates. Modern x86 CPUs only guarantee that 8-byte, aligned stores atomically update NVMM [44]. If applications need larger atomic updates, they must manually construct software transactions. `Libpmemobj` provides undo log-based transactions. The application executes stores to NVMM between the `TX_BEGIN` and `TX_END` macros, and snapshots (`pmemobj_tx_add_range`) a range of object data before modifying it in-place.

Persistence Ordering. Intel CPUs provide cache flush/write-back (e.g., `CLFLUSH(OPT)` and `CLWB`) and memory ordering (e.g., `SFENCE`) instructions to make guarantees about when stores become persistent. In Listing 3.1, the `pmemobj_persist` function and `TX` macros integrate these instructions to flush modified object ranges.

`Libpmemobj` supports a replicated mode that requires a replica pool, doubling the storage the object store requires. `Libpmemobj` applies updates to both pools to keep them synchronized.

Replicated `libpmemobj` can detect and recover from media errors only when the object store is offline, and it cannot detect or recover from data corruption caused by errant stores to NVMM – so-called "scribbles," that might result from a buffer overrun or dereferencing a wild pointer.

3.2 Pangolin Design

Pangolin allows programmers to build complex, crash-consistent persistent data structures that are also robust in the face of media errors and software "scribbles" that corrupt data. Pangolin satisfies all of the criteria listed in the introduction of this chapter. This section describes its architecture and highlights the key challenges that Pangolin addresses to meet those requirements. In particular, Pangolin provides the following features unseen in prior works.

- It provides fast, space-efficient recovery from media errors and scribbles.

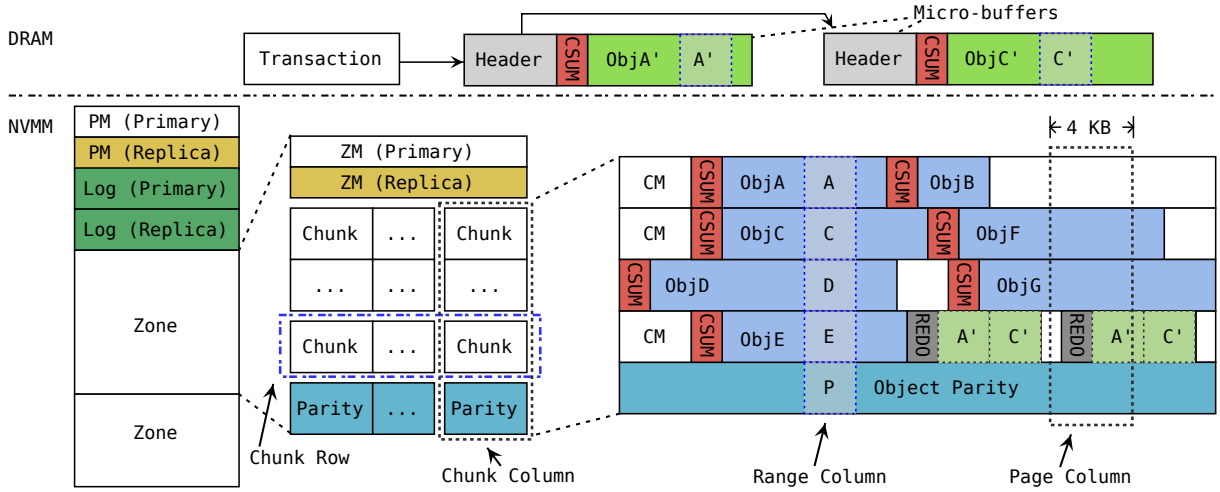

Figure 3.2: Data protection scheme in Pangolin - Pangolin protects pool metadata (PM), zone metadata (ZM), and chunk metadata (CM).

- It uses checksums to protect object integrity and supports incremental checksum updates.
- It integrates parity and checksum updates into an NVMM transaction system.
- It periodically scrubs data to identify corruption.
- It detects and recovers from media errors and scribbles online.

Pangolin guarantees that it can recover from the loss of any single 4 KB page of data in a pool. In many cases, it can recover from the concurrent loss of multiple pages.

We begin by describing how Pangolin organizes data to protect user objects, library metadata, and transaction logs using a combination of parity, replication, and checksums. Next, we describe micro-buffers and explain how they allow Pangolin to preserve a simple programming interface and protect against software scribbles. Then, we explain how Pangolin detects and prevents NVMM corruption and elaborate on Pangolin's transaction implementation with support for efficient, concurrent updates of object parity. Finally, we discuss how Pangolin restores data integrity after corruption and crashes.

3.2.1 Pangolin's Data Organization

Pangolin uses replication for its internal metadata and RAID-style parity for user objects to provide redundancy for corruption recovery. The MCE mechanism described in Section 3.1.2 and object checksums in Pangolin detect corruption.

Pangolin views a zone's chunks as a two-dimensional array as shown in the middle of Figure 3.2. Each *chunk row* contains multiple, contiguous chunks and the chunks "wrap around" so that the last chunk of a row and the first chunk of the next are adjacent. Pangolin reserves the last chunk row for parity.

In our description of Pangolin, we define a *page column* as a one page-wide, aligned column that cuts across the rows of a zone. A *range column* is similar, but can be arbitrarily wide (no more than a chunk row's size).

Initializing a parity-coded NVMM pool requires zeroing out all the bytes in the file. This is a one-time overhead when creating a pool file and does not affect run-time performance. We report this latency in Section 3.3.

To detect corruption in user objects, Pangolin adds a 32-bit checksum to the object's header. The header also contains the object's size (64-bit) and type (32-bit). The compiler determines type values according to user-defined object types. Pangolin inherits this design from `libpmemobj` and changes the type identifier from 64-bit to 32-bit for the checksum.

Pangolin's object placement is independent of chunk and row boundaries. Objects can be anywhere within a zone, and they can be of any size (up to the zone size).

In addition to user objects, the library maintains metadata for the pool, zones, and chunks, including allocation bitmaps. Pangolin checksums these data structures to detect corruption and replicates the pool's and zones' metadata for fault tolerance. These structures are small (less than 0.1% for pools larger than 1 GB), so replicating them is not expensive. Pangolin uses zone parity to support recovery of chunk metadata.

Pangolin checksums transaction logs and replicates them for redundancy. It treats log

Table 3.1: The Pangolin API - Pangolin's interface mirrors `libpmemobj`'s except that Pangolin does not allow direct writing to NVMM.

Function	Semantics
`pgl_tx_begin()/commit()/end(), etc.`	Control the lifetime of a Pangolin transaction.
`pgl_tx_alloc()/free()`	Allocate or deallocate an NVMM object.
`pgl_tx_open(PMEMoid oid, ...)`	Create a thread-local micro-buffer for an NVMM object. Verify (and restore) the object integrity, and return a pointer to the micro-buffered user object.
`pgl_tx_add_range(PMEMoid oid, ...)`	Invoke `pgl_tx_open` and then mark a range of the object that will be modified.
`pgl_get(PMEMoid oid)`	Get access to an object, either directly in NVMM or in its micro-buffer, depending on the transaction context. By default, it does not verify the checksum.
`pgl_open(PMEMoid oid, ...)`	Create a micro-buffer for an NVMM object without a transaction. Check the object integrity, and return a pointer to the micro-buffered user object.
`pgl_commit(void *uobj)`	Automatically start a transaction and commit the modified user object in a micro-buffer to NVMM.

entries in zone storage as zeros during parity calculations. This prevents parity update contention between log entries and user objects (see Section 3.2.5).

Fault Tolerance Guarantees. Pangolin can tolerate a single 4 KB media error anywhere in the pool, regardless of whether it is a data page or a parity page. Based on the bad page's address Pangolin can locate its page column and restore its data using other healthy pages.

Faults affecting two pages of the same page column may cause data loss if the corrupted ranges overlap. If an application demands more robust fault tolerance, it can increase the chunk row size, reducing the number of rows and, consequently, the likelihood that two corrupt pages overlap.

Pangolin can recover from scribbles (contiguous overwrites caused by software errors) on NVMM data up to a chunk-row size. By default, Pangolin uses 100 chunk rows, and parity consumes $\sim$1% of a pool's size (e.g., 1 GB for a 100 GB pool).

3.2.2 Micro-buffering for NVMM Objects

Pangolin introduces micro-buffering to hide the complexity of updating checksums and parity when modifying NVMM objects. Adding checksums to objects and protecting them with parity makes updates more complex, since all three – object data, checksum, and parity –

must change at once to preserve consistency. This challenge is especially acute for the atomic programming model as shown in Listing 3.1 (line 3-5) because a single 8-byte NVMM write cannot host all these updates.

Micro-buffering creates a shadow copy of an NVMM object in DRAM, which separates an object's transient and persistent versions (Figure 3.2). In Listing 3.2, `pgl_open` creates a micro-buffer for the node object by allocating a DRAM buffer and copying the node's data from NVMM. It also verifies the object's checksum and performs corruption recovery if necessary.

The application can modify the micro-buffered object without concern for its checksum, parity, and crash-consistency because changes exist only in the micro-buffer. When the updates finish, `pgl_commit` starts a transaction that atomically updates the NVMM object, its checksum, and parity (described below). Compared to line 3-5 of Listing 3.1, Pangolin retains the simple, atomic-style programming model for modifying a single NVMM object, and it supports updates within an object beyond 8 bytes.

Each micro-buffer's header contains information such as its NVMM address, modified ranges, and status flags (e.g., allocated or modified). We elaborate on Pangolin's programming interface and how to construct complex transactions with micro-buffering in Section 3.2.4.

```
1 struct node *n = pgl_open(node_oid);
2 n->val = value;
3 pgl_commit(pool, n);
```

Listing 3.2: A Pangolin transaction for a single-object.

Another important consideration for micro-buffering is to prevent misbehaving software from corrupting NVMM. If an application's code can directly write to NVMM, as `libpmemobj` allows to, software bugs such as buffer overflows and using dangling pointers can easily cause NVMM corruption. Conventional debugging tools for memory safety, such as Valgrind [76] and AddressSanitizer [94], insert inaccessible bytes between objects as "redzones" to trap illegal accesses. This approach fails to work for directly accessed NVMM objects because once they

are allocated, there is no guarantee for spacing between them, and thus, redzones may land on a nearby, accessible object. One viable approach to using these tools is to let the NVMM allocator insert redzones. However, the presence of redzone bytes will pollute the pool and may exacerbate fragmentation.

Using micro-buffers isolates transient writes from persistent data, and since micro-buffers are dynamically allocated using `malloc()`, they are compatible with existing memory debugging tools. Without using debugging tools, Pangolin also protects micro-buffers by inserting a 64-bit "canary" word in each micro-buffer's header and checks its integrity before writing back to NVMM. On transaction commit, if Pangolin detects a canary mismatch, it aborts the transaction to avoid propagating the corruption to NVMM. Pangolin uses checksums to detect corruptions that may bypass the canary protection.

3.2.3 Detecting NVMM Corruption

Pangolin uses three mechanisms to detect NVMM corruption. First, it installs a handler for `SIGBUS` (see Section 3.1.2) that fires when the Linux kernel receives an MCE. A signal handler has access to the address the offending load accessed, and Pangolin can determine what kind of data (i.e., metadata or a user object) lives there and recover appropriately. This mechanism detects media failures, but it cannot discover corrupted data caused by software "scribbles."

To detect scribbles, Pangolin verifies the integrity of user objects using their checksums. Verifying checksums on every access can be expensive. To limit this cost, by default Pangolin only verifies checksums during micro-buffer creation before any object is modified in a transaction. This keeps Pangolin from recalculating a new checksum based on corrupt data. For read-only objects that are accessed by `pgl_get` without micro-buffering, by default Pangolin does not verify checksums. To protect them, Pangolin provides two alternative operation modes: "Scrub" mode runs a scrubbing thread that verifies and restores the whole pool's data integrity when a preset number of transactions have completed, and "Conservative" mode verifies the checksum for every

object access (including `pgl_get`). We evaluate the impact of different checksum verification policies in Section 3.3.

Finally, Linux keeps track of known bad pages of NVMM across reboots. When opening a pool or during its scrubbing, Pangolin can extract this information and recover the data in the reported pages (not currently implemented).

3.2.4 Fault-Tolerant Transactions

Failure-atomic transactions are central to Pangolin's interface, and they must include verification of data integrity and updates to the checksums and parity data that protect objects. Table 3.1 summarizes Pangolin's core functions.

Pangolin supports arbitrary-sized transactions and we have made similar APIs and macros as `libpmemobj`'s. The program in Listing 3.1 can be easily transformed to Pangolin using equivalent functions. One subtle difference is in the handling of atomic-style updates, as shown in Listing 3.2.

In Pangolin, each thread can execute one transaction or nested transactions (same as `libpmemobj`). Concurrent transactions can execute if each one is associated with a different thread. Currently, Pangolin does not allow concurrent transactions to modify the same NVMM object. Concurrently modifying a shared object may cause data inconsistency if one transaction has to abort. `Libpmemobj` has the same limitation [45].

Each transaction manages its own micro-buffers using a thread-local hashmap [58], indexed by an NVMM object's `PMEMoid`. Therefore, in a transaction, calling `pgl_tx_open` for the same object either creates or retrieves its micro-buffer. Multiple micro-buffers opened in one transaction form a linked list as shown in Figure 3.2. Micro-buffers for one transaction are not visible in other transactions, providing isolation.

If a transaction modifies an object, Pangolin copies it to a micro-buffer, performs the changes there, and then propagates the changes to NVMM during commit. Since changes occur

in DRAM (which does not require undo information), Pangolin implements redo logging.

At transaction commit, Pangolin recomputes the checksums for modified micro-buffers, creates and replicates redo log entries for the modified parts of the micro-buffers and writes these ranges back to NVMM objects. Then, it updates the affected parity bits (see Section 3.2.5) and marks the transaction committed. Finally, Pangolin garbage-collects its logs and closes thread-local micro-buffers.

If a transaction aborts, either due to unrecoverable data corruption or other run-time errors, Pangolin discards the transaction's micro-buffers without touching NVMM.

A transaction can also allocate and deallocate objects. Pangolin uses redo logging to record NVMM allocation and free operations, just as `libpmemobj` does.

For read-only workloads, repeatedly creating micro-buffers and verifying object checksums can be very expensive. Therefore, Pangolin provides `pgl_get` to gain direct access to an NVMM object without verifying the object's checksum. The application can verify an object's integrity manually as needed or rely on Pangolin's periodic scrubbing mechanism. Inside a transaction context, `pgl_get` returns a pointer to the object's micro-buffer to preserve isolation.

3.2.5 Parity and Checksum Updates

Objects in different rows can share the same range of parity, and we say these objects *overlap*. Object overlap leads to a challenge for updating the shared parity because updates from different transactions must serialize but naively locking the whole parity region sacrifices scalability.

For instance, using *ObjA* and *ObjC* in Figure 3.2, suppose two different transactions modify them, replacing A with A' and C with C', respectively. After both transactions update P, the parity should have the value $P' = A' \oplus C' \oplus D \oplus E$ regardless of how the two transaction commits interleave.

Pangolin uses a combination of two techniques that exploit the commutativity of XOR

and fine-grained locking to preserve correctness and scalability.

Atomic parity updates. The first approach uses the atomic XOR instruction (analogous to an atomic increment) that modern CPUs provide to perform incremental parity updates for changes to each overlapping object.

In our example, we can compute two parity patches: $\Delta A = A \oplus A'$, $\Delta C = C \oplus C'$ and then rewrite P' as $P \oplus \Delta A \oplus \Delta C$. Since XOR commutes and is a bit-wise operation, the two threads can perform their updates without synchronization.

Hybrid parity updates. Atomic XOR is slower than normal or vectorized XOR. For small updates, the latency difference between them is not significant, and Pangolin prefers atomic XOR instructions to update parity without the need for locks. But for large parity updates, atomic XOR can be inefficient. Therefore, Pangolin's hybrid parity scheme switches to vectorized XOR for large transfers.

To coordinate large and small parity updates, Pangolin uses *parity range-locks*, that work similarly as reader/writer locks (or shared mutex): Small writes take shared ownership of a range lock and update parity with atomic XOR instructions. Large updates using vectorized XORs take exclusive ownership of a range-lock, and only one thread can modify parity in a locked range. If one update involves multiple range-locks, serialization happens on a per-range-lock basis.

The managed size of a parity range-lock depends on the performance trade-off between Pangolin's parity mode and `libpmemobj`'s replication mode. We discuss this in Section 3.3.

Pangolin refreshes an object's checksum in its micro-buffer before updating parity, and it considers the checksum field as one of the modified ranges of the object. Checksums like CRC32 requires recomputing the checksum using the whole object. This can become costly with large objects. Thus, Pangolin uses Adler32 [54], a checksum that allows incremental updates, to make the cost of updating an object's checksum proportional to the size of the modified range rather than the object size.

We implement Pangolin's parity and checksum updates using the Intelligent Storage

Acceleration Library (ISA-L) [46], which leverages SIMD instructions of modern CPUs for these data-intensive tasks.

Protections for other transaction systems. Other NVMM persistent object systems could apply Pangolin's techniques for parity and checksum updates. For example, consider an undo logging (as opposed to Pangolin's redo logging) system that first stores a "snapshot" copy of an object in the log before modifying the original in-place. In this case, the system could compute a parity patch using the XOR result between the logged data (old) and the object's data (new). Then, it can apply the parity patch using the hybrid method we described in this section.

3.2.6 Recovering from Faults

In this section, we discuss how Pangolin recovers data integrity from both NVMM corruption and system crashes.

Corruption recovery. Pangolin uses the same algorithm to recover from errors regardless of how it detects them (i.e., via `SIGBUS` or a checksum mismatch).

The first step is to pause the current thread's transaction, and to wait until all other outstanding transactions have completed. Meanwhile, Pangolin prevents the initialization of new transactions by setting the pool's "freeze" flag. This is necessary because, during transaction committing, parity data may be inconsistent.

Once the pool is frozen, Pangolin uses the parity bits and the corresponding parts of each row in the page column to recover the missing data.

Pangolin preserves crash-consistency during repair by making persistent records of the bad pages under recovery. Recovery is idempotent, so it can simply re-execute after a crash.

Pangolin's current implementation only allows one thread to perform any online corruption recovery, and if the thread is executing a transaction, online recovery only works if the thread has not started committing. If two threads encounter faults simultaneously, Pangolin kills the

application and performs post-crash recovery (see below) when it restarts. Supporting multi-threaded online recovery, and allowing it to work when threads have partially written NVMM is possible, but it requires complex reasoning about how to restore the data and its parity to a consistent state.

Crash recovery. Pangolin handles recovery from a crash using its redo logs. It must also protect against the possibility that the crash occurred during a parity update.

To commit a transaction, Pangolin first ensures its redo logs are persistent and replicated, and then updates the NVMM objects and their parity. If a crash happens before redo logs are complete, Pangolin discards the redo logs on reboot without touching the objects or parity. If redo logs exist, Pangolin replays them to update the objects and then recomputes any affected parity ranges using the data written during replay (which is now known to be correct) and the data from the other rows.

Pangolin does not log parity updates because it would double the cost of logging. This does raise the possibility of data loss if a crash occurs during a parity update and a media error then corrupts data of the same page column before recovery can complete. This scenario requires the simultaneous loss of two pages in the same page column due to corruption and a crash, which we expect to be rare.

3.3 Evaluation

In this section, we evaluate Pangolin's performance and the overheads it incurs by comparing it to normal `libpmemobj` and its replicated version. We start with our experimental setup and then consider its storage requirements, latency impact, scalability, application-level performance, and corruption recovery.

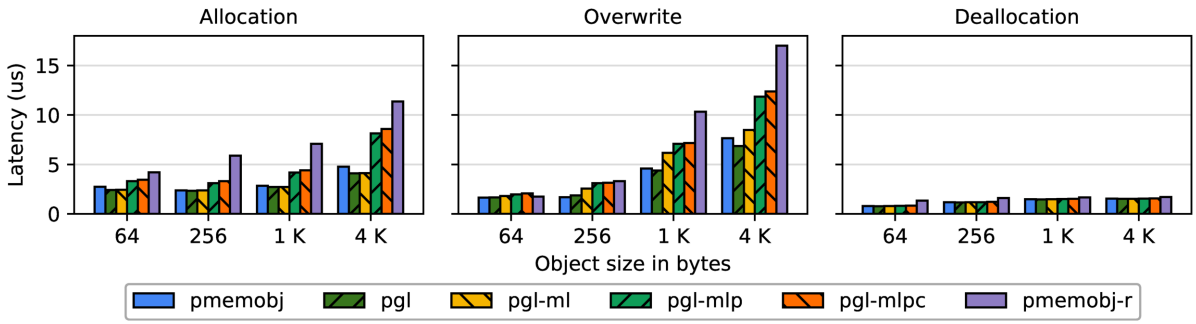

Figure 3.3: Transaction performance - Each transaction allocates, overwrites, or frees one object of varying sizes. Pangolin's latencies are similar to Pmemobj's.

3.3.1 Evaluation Setup

We perform our evaluation on a dual-socket platform with Intel's Optane DC Persistent Memory [47]. The CPUs are 24-core engineering samples of the Cascade Lake generation. Each socket has 192 GB DDR4 DRAM and 1.5 TB NVMM. We configure the persistent memory modules in *AppDirect* mode and run experiments on one socket using its local DRAM and NVMM. A recent report [49] studying this platform provides more architectural details.

The CPU provides the `CLWB` instruction for writing-back cache lines to NVMM, non-temporal store instructions to bypass caches, and the `SFENCE` instruction to ensure persistence and memory ordering. It also has atomic XOR and AVX instructions that our parity and checksum computations use.

The evaluation machine runs Fedora 27 with a Linux kernel version 4.13 built from source with the NOVA [104] file system. We run experiments with both Ext4-DAX [63] and NOVA, and applications use `mmap()` to access NVMM-resident files. The performance is similar on the two file systems because DAX-`mmap()` essentially bypasses them.

On our evaluation machine, we found that updating parity with atomic XORs becomes worse than `libpmemobj`'s replication mode when the modified parity range is greater than 8 KB, so we set 8 KB as the threshold to switch between those parity calculation strategies (see

Table 3.2: Library configurations for evaluation - In the figures, we abbreviate Pangolin as `pgl`.

Pmemobj	`libpmemobj` baseline from PMDK v1.5
Pangolin	Pangolin baseline w/ micro-buffering only
Pangolin-ML	Pangolin + metadata and redo log replication
Pangolin-MLP	Pangolin-ML + object parity
Pangolin-MLPC	Pangolin-MLP + object checksums
Pmemobj-R	`libpmemobj` w/ one replication in another file

Section 3.2.5).

Table 3.2 describes the operation modes for our evaluations. The Pangolin baseline implements transactions with micro-buffering. It uses buffer canaries to prevent corruption from affecting NVMM, but it does not have parity or checksum for NVMM data.

We evaluate versions of Pangolin that incrementally add metadata and log replication ("+ML"), object parity ("+MLP"), and checksums ("+MLPC"). We combine the impact of metadata updates with log replication because metadata updates are small and cheap in our evaluation.

Pmemobj-R is the replication mode of `libpmemobj` that mirrors updates to a replica pool during transaction commit. Comparing Pangolin-MLP and Pmemobj-R is especially useful because the two configurations protect against the same types of data corruption: media errors but not scribbles.

3.3.2 Memory Requirements

We discuss and evaluate Pangolin's memory requirements for both NVMM and DRAM.

NVMM All our Pangolin experiments use a single pool of 100 GB that contains 6× 16 GB zones. Pangolin replicates all the pool's metadata in the same file, which occupies a fixed ∼20 MB. The rest of the space is for user objects and their protection data. By default, Pangolin uses 100 chunk rows, so each zone has about 160 MB parity, and that totally occupies ∼1% of the pool's capacity. Pmemobj-R uses a second 100 GB file as the replica, doubling the cost of

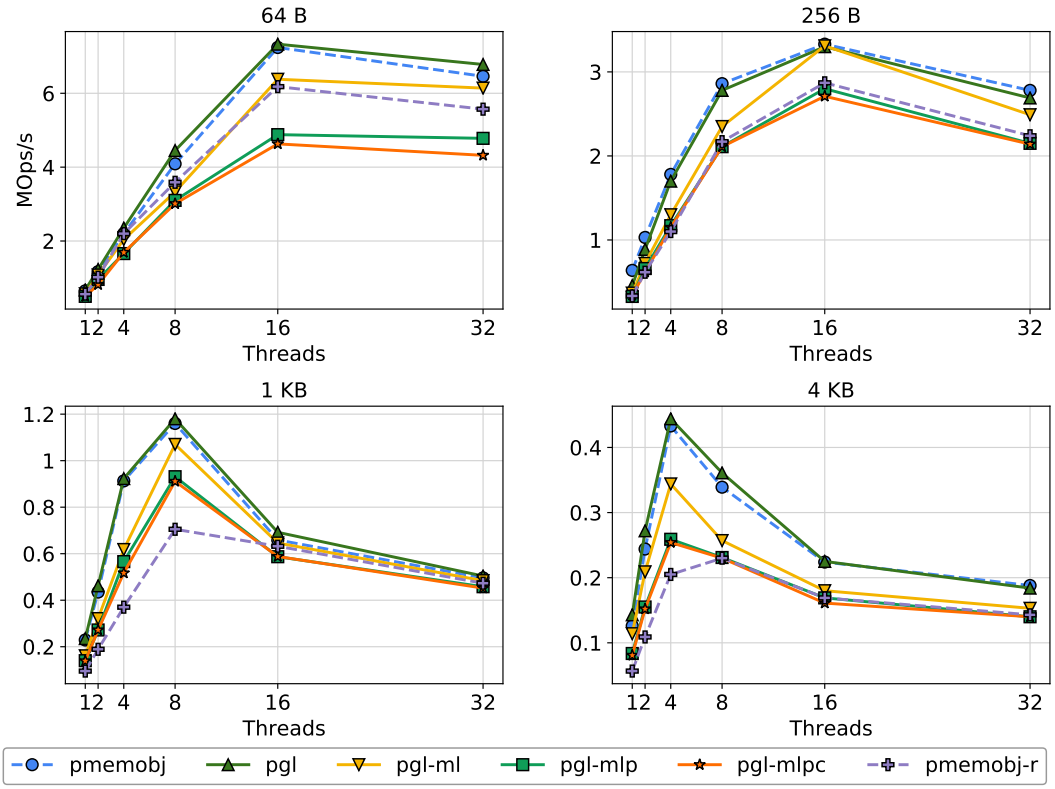

Figure 3.4: Scalability - Concurrent workloads randomly overwrite objects of varying sizes.

NVMM space requirement.

When using parity, Pangolin has to zero out the whole pool to ensure all zones are initially parity-coded. This takes about 130 seconds. It is a one-time overhead for creating the pool and excluded from the following evaluations.

DRAM Pangolin uses `malloc()`'d DRAM to construct micro-buffers. The required DRAM space is proportional to ongoing transaction sizes. Table 3.3 summarized the transaction sizes for the evaluated key-value store data structures. Pangolin automatically recycles them on transaction commits. In our evaluation experiments, micro-buffering never exceeds using 50 MB of DRAM.

3.3.3 Transaction Performance

Figure 3.3 illustrates the transaction latencies for three basic operations on an NVMM object store: object allocation, overwrite, and deallocation. Each transaction operates on one object, and we vary the size of the object.

For allocation, latency grows with object size for all five configurations, due to constructing the object and cache line write-back latency. Pangolin incurs 2% - 13% lower latencies than Pmemobj due to its use of non-temporal stores for write backs. An allocation operation does not involve object logging, so Pangolin-ML shows performance similar to Pangolin. Pangolin-MLP adds overhead to update the parity data. It outperforms Pmemobj-R by between 1.2× and 1.9×. We found this is because updating parity using atomic XORs and `CLWB`s incurs less latency than mirroring data in a separate file, as Pmemobj-R does.

Adding checksum (Pangolin-MLPC) incurs less than 7% overhead compared to Pangolin-MLP. Parity's impact is larger than checksum's because updating a parity range demands values from three parts: the micro-buffer, the NVMM object, and the old parity data, while computing a checksum only needs data in a DRAM-based micro-buffer. Moreover, Pangolin needs to flush the modified parity range to persistence, which is the same size as the object. In contrary, updating a checksum only writes back a single cache line that contains the checksum value.

Overwriting an NVMM object involves transaction logging for crash consistency. Pangolin and Pmemobj store the same amount of logging data in NVMM, although they use redo logging and undo logging for this purpose, respectively. Since log entry size is proportional to an object's modified size, which is the whole object in this evaluation, this cost grows with the object. With Pangolin, log replication accounts for between 7% to 25% of the latency. Parity updates consume between 8% to 27% of the extra latency, depending on object size, and checksum updates account for less than 5%. Pangolin-MLP's performance for overwrites is 12% worse than Pmemobj-R for 64 B object updates and is between 1.1× and 1.5× better than Pmemobj-R for objects larger than 64 B.

Table 3.3: Data structure and transaction sizes - "Insert" and "Remove" show average transaction sizes for insertions and removals, respectively. "New" and "Mod" indicate average allocated and modified sizes.

		ctree	rbtree	btree	skiplist	rtree	hashmap
Object size		56	80	304	408	4136	10 M (table), 40 (entry)
Insert	New	56 (1.00)	80 (1.00)	65.9 (0.22)	408 (1.00)	4502 (1.09)	60.9 (1.00)
	Mod	127.6 (3.28)	330.2 (5.13)	381.2 (1.47)	33.9 (2.50)	200.0 (5.05)	331.1 (4.21)
Remove	New	0	0	0	0	184.1 (0.05)	10.5 (1×10^{-5})
	Mod	28.0 (0.50)	202.8 (2.65)	268.3 (0.90)	16.9 (0.75)	98.6 (2.52)	254.3 (2.16)

Deallocation transactions only modify metadata, so their latencies do not change much.

3.3.4 Scalability

Figure 4.8 measures Pangolin's scalability by randomly overwriting existing NVMM objects and varying the object sizes and the number of concurrent threads.

Pangolin uses reader/writer locks to implement the hybrid parity update scheme described in Section 3.2.5. The number of rows in a zone and the zone size determine the granularity of these locks: For a fixed zone size, more rows means fewer columns and fewer parity range-locks.

There is no lock contention in the results because the transactions use atomic XOR instructions and can execute concurrently (only taking the reader locks). Our configuration with 1% parity (160 MB parity per 16 GB zone) has 20 K range-locks per zone, so the chance of lock contention is slim even with large updates (more than 8 KB) and many cores.

The graphs also show how each Pangolin's fault-tolerance mechanisms affect performance. Pangolin's throughput is very close to Pmemobj. Pangolin-MLP mostly outperforms Pmemobj-R for object updates that are 256 B or larger, up to 1.5×. But for 64 B object updates, it performs worse than Pmemobj-R by between 6% and 25%. This is because when enabling parity, every Pangolin transaction checks the pool freeze flag (an atomic variable), incurring synchronization overhead. This overhead is noticeable for short transactions with 64 B objects but becomes negligible for larger updates. Pangolin-MLPC only performs marginally worse than Pangolin-MLP.

Figure 3.5: Key-value store performance - Each transaction either inserts or removes one key-value pair from the data store.

Scaling degrades for all configuration as update size and thread count grow because the sustainable bandwidth of the persistent memory modules becomes saturated.

3.3.5 Impacts on NVMM Applications

To evaluate Pangolin in more complex applications, we use six data structures included in the PMDK toolkit: crit-bit tree (ctree), red-black tree (rbtree), btree, skiplist, radix tree (rtree), and hashmap. They have a wide range of object sizes and use a diverse set of algorithms to insert, remove, and lookup values. We rewrite these benchmarks with Pangolin's programming interface as described in Section 3.2.4.

Table 3.3 summarizes the object and transaction sizes for each workload. The tree structures and the skiplist have a single type of object, which is the tree or list node. Hashmap has two kinds of objects. One is the hash table that contains pointers to buckets. The hash table grows as the application inserts more key-value pairs. Each bucket is a linked list of fixed-sized

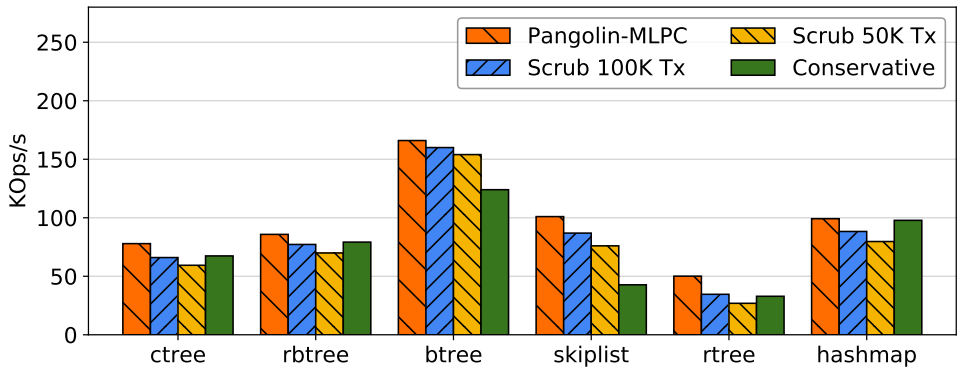

Figure 3.6: Checksum verification impact - Pangolin-MLPC bars are the same as those in Figure 3.5 for 1M Inserts. The cost of different policies depends strongly on data structures.

entry objects.

Each insertion or removal is a transaction processing a key-value pair. The workloads involve a mix of object allocations, overwrites, and deallocations. Table 3.3 shows, on average, the number of bytes and objects (in parentheses) involved in each data structure's transaction. Deallocated sizes are not shown because they marginally affect the performance differences (see Figure 3.3).

An average allocation size ("New" rows in the table) smaller than the object size means the data structure does not allocate a new object for every insert operation (e.g., btree). The average modified sizes ("Mod" rows) determine the logging size and affect the performance drop between Pangolin and Pangolin-ML. Note that a transaction does not necessarily modify (and log) the whole range of an involved object. The performance difference between Pangolin-ML and Pangolin-MLP is a consequence of both allocated and modified sizes.

For insert transactions, Pangolin is faster than Pmemobj for ctree and btree, but slower than Pmemobj for other data structures. This is because the slower applications have relatively small modified sizes compared to the object sizes, and Pangolin's data movement from NVMM to micro-buffers overshadows its advantage for whole-object updates, as shown in Figure 3.3. For remove transactions, Pangolin is marginally faster than Pmemobj except for the case of skiplist,

Table 3.4: Vulnerability evaluation - Each row shows object bytes (normalized to Pmemobj) accessed without checksum verification.

	ctree	rbtree	btree	sklist	rtree	hmap
Pmemobj	1.0	1.0	1.0	1.0	1.0	1.0
Pgl-MLPC	0.92	0.84	0.87	0.96	0.42	0.42
Scrub 100K	0.10	0.09	0.09	0.10	0.04	0.05
Scrub 50K	0.05	0.04	0.05	0.05	0.02	0.02
Conservative	0	0	0	0	0	0

which is also because of the data movement caused by micro-buffering.

Pangolin-MLP's performance is 95% of Pmemobj-R on average, and it saves orders of magnitude NVMM space by using parity data as redundancy. Pangolin-MLPC adds scribble detection and performance drops by between 1.5% to 15% relative to Pangolin-MLP. Adding object checksums impacts rtree's transactions the most because the allocated object size is large, which requires more checksum computing time.

Pangolin does not impact the lookup performance because it performs direct NVMM reads without constantly verifying object checksums. Pangolin ensures data integrity with its checksum verification policy, as discussed in Section 3.2.3.

Figure 3.6 illustrates the impact of different strategies for checksum verification. We compare Pangolin's default mode (Pangolin-MLPC) with two "Scrub" modes and a "Conservative" mode. The default mode only verifies checksums for micro-buffered objects. In "Scrub" mode, a scrubbing thread verifies data integrity of the whole object pool when a preset number (indicated by legends in Figure 3.6) of transactions have completed. The "Conservative" mode verifies the checksum for every object access (including those read by `pgl_get` without micro-buffering).

Table 3.4 quantifies the vulnerability using the amount of object data that is accessed without checksum verification. The data accumulates across all transactions for Pmemobj, Pangolin-MLPC, and "Conservative" modes. For "Scrub" modes, we count the vulnerable data between two scrubbing runs. Numbers in Table 3.4 are normalized to Pmemobj, which does not have any checksum protection for object data.

The cost of verifying checksums for every object access depends strongly on the data structure size and its insertion algorithm. For small objects, such as ctree, rbtree, and hashmap, the cost is negligible. But for btree, skiplist, and rtree, due to their large object sizes, the cost is significant. Thus, a scrubbing-based policy could be faster, with more data subject to corruption between two successive runs.

3.3.6 Error Detection and Correction

Pangolin provides error injection functions to emulate both hardware-uncorrectable NVMM media errors and hardware-undetectable scribbles.

We initially developed Pangolin using conventional DRAM machines that lack support for injecting NVMM errors at the hardware level. Therefore, we use `mprotect()` and `SIGSEGV` to emulate NVMM media errors and `SIGBUS`. When an NVMM file is DAX-mapped, the injector can randomly choose a page that contains user-allocated objects, erase it, and call `mprotect(PROT_NONE)` on the page. Later, when the application reads the corrupted page, Pangolin intercepts `SIGSEGV`, changes the page to read/write mode, and restores the page's data. The injector function can also randomly corrupt a metadata region or a victim object to emulate software-induced, scribble errors.

In both test cases, we observe Pangolin can successfully repair a victim page or an object and resume normal program execution. In our evaluation using a 100 GB pool and 1 GB parity, we measured 180 μs to repair a page of a page column.

We also intentionally introduce buffer overrun bugs in our applications and observe that Pangolin can successfully detect them using micro-buffer canaries. The transaction then aborts to prevent any NVMM corruption. We have also verified Pangolin is compatible with AddressSanitizer for detecting buffer overrun bugs (when updating a micro-buffered object exceeds its buffer's boundary), if both Pangolin and its application code are compiled with support.

3.4 Discussion

Persistent memory promises dramatic increases in storage performance. However, it also complicates data protection and fault tolerance with direct access from the user-space. In this section, we discuss our anticipated challenges and Pangolin's limitations.

Hardware vs. software protection. Advanced memory-protection techniques such as Chipkill and RAIM [65] can provide better reliability than basic ECC at the hardware level. But whether emerging NVMM modules will adopt these techniques is unclear, and because they do not understand software-layer semantics, they cannot protect against scribbles caused by software memory bugs. Implementing parity and checksum in Pangolin is necessary for providing software-layer protection, but it inevitably sacrifices performance, as shown in our evaluation, especially when updating parity. If the NVMM controller can perform such data-intensive tasks with control from the software layer, it can reduce the performance penalty for fault tolerance.

Vulnerability windows. Although Pangolin provides a viable solution for software-based NVMM fault-tolerance, it only enforces automatic checksum verification with the micro-buffered access, that is, when using `pgl_open()` or `pgl_tx_open()` functions. To preserve performance for read-only workloads, Pangolin's `pgl_get()` function gives direct read access to NVMM objects and does not perform checksum verification. Although Pangolin can still repair hardware-detected media errors, it cannot discover silent data corruption when using `pgl_get()`. A time-based NVMM pool scrubbing mechanism can reduce vulnerability windows with a performance hit.

Online recovery. Being able to repair media errors and corruptions online is critical to providing high availability for NVMM-based applications. The asynchronous, signal-based mechanism for notifying a user-space process about NVMM media errors raises great challenges for online error recovery. Currently, Pangolin only allows one thread to perform any online corruption recovery, and that thread must not be in the middle of updating any parity data.

Supporting multi-threaded online recovery, and allowing it to work when the thread has partially written to NVMM would require complex reasoning about how to restore the data and its parity to a consistent state, and may require some hardware- or kernel-level support for better error notification mechanisms.

3.5 Related Work

In this section, we place Pangolin in context relative to previous projects that have explored how to use NVMM effectively.

Transaction Support. All previous libraries for using NVMMs to build complex objects rely on transactions for crash consistency. Although we built Pangolin on `libpmemobj`, its techniques could be applied to another persistent object system. NV-Heaps [12], Atlas [10], DCT [52], and `libpmemobj` [82] provide undo logging for applications to snapshot persistent objects before making in-place updates. Mnemosyne [103], SoftWrAp [27], and DUDETM [59] use variations of redo logging. REWIND [11] implements both undo and redo logging for fine-grained, high-concurrent transactions. Log-structured NVMM [36] makes changes to objects via append-only logs, and it does not require extra logging for consistency. Romulus [14] uses a main-back mechanism to implement efficient redo log-based transactions.

None of these systems provide fault tolerance for NVMM errors. We believe they can adopt Pangolin's parity and checksum design to improve their resilience to NVMM errors at low storage overhead. In Section 3.2.5 we described how to apply the hybrid parity updating scheme to an undo logging-based system. Log-structured and copy-on-write systems can adopt the techniques in similar ways.

Fault Tolerance. Both Pangolin and `libpmemobj`'s replication mode protect against media errors, but Pangolin provides stronger protection and much lower space overhead. Furthermore, `libpmemobj` can only repair media errors offline, and it does not detect or repair software

corruption to user objects.

NVMalloc [72] uses checksums to protect metadata. It does not specify whether application data is also checksum-protected, and it does not provide any form of redundancy to repair the corruption. NVMalloc uses `mprotect()` to protect NVMM pages while they are not mapped for writing. Pangolin could adopt this technique to prevent an application from scribbling its own persistent data structures.

The NOVA file system [104, 105] uses parity-based protection for file data. However, it must disable these features for NVMM pages that are DAX-mapped for writing in user-space, since the page's contents can change without the file system's knowledge, making it impossible for NOVA to keep the parity information consistent if an application modifies DAX-mapped data. As a result, Pangolin's and NOVA's fault tolerance mechanisms are complementary.

3.6 Conclusion

This work presents Pangolin, a fault-tolerant, DAX-mapped NVMM programming library for applications to build complex data structures in NVMM. Pangolin uses a novel, space-efficient layout of data and parity to protect arbitrary-sized NVMM objects combined with per-object checksums to detect corruption. To maintain high performance, Pangolin uses micro-buffering, carefully-chosen parity and checksum updating algorithms. As a result, Pangolin provides stronger protection, better availability, and much lower storage overhead than existing NVMM programming libraries.

Acknowledgments

This chapter contains material from "Pangolin: A Fault-Tolerant Persistent Memory Programming Memory" by Lu Zhang and Steven Swanson, which has appeared in the proceedings

of the 2019 USENIX Annual Technical Conference. The dissertation author is the primary investigator and first author of this paper. The materials are copyright ©2019 by USENIX Association.

Chapter 4

PmemConjurer and PmemSanitizer

Although DAX offers fast access to persistent data, ensuring crash consistency of data stored in directly-mapped NVMM remains challenging. We define *recovery bugs* as NVMM-specific programming errors that may cause unrecoverable data inconsistency after a crash. Developing NVMM applications without recovery bugs requires programmers to carefully reason about when and in what order data becomes persistent during program execution, explicitly insert cache-line flushing and memory ordering instructions at proper locations in the source code, and implement recovery methods to restore an NVMM image to a consistent state after a crash. Moreover, it also demands special memory management mechanisms (e.g., atomic, crash-consistent memory allocation) and transaction algorithms that are unique to NVMM programming. Adding the required functionality introduces pervasive changes to existing programming practices, and the subtleties involved open the door to a wide range of recovery bugs.

To address these challenges, industry and academia have proposed libraries [12, 14, 18, 36, 48, 59, 66, 82, 103, 108] to facilitate NVMM programming. They typically encapsulate low-level cache-line operations in convenient library functions, provide NVMM allocators for memory management, and support transactions to manage arbitrary-sized NVMM updates. Despite hiding low-level details from programmers, NVMM libraries replace one set of challenges with another

– using the libraries correctly. Recovery bugs also manifest with inappropriate usage of library functions. The public commit history for Intel's Persistent Memory Development Kit (PMDK) example programs demonstrates how easily even the library designers can misuse them.

These challenges illustrate the need for effective NVMM recovery debugging tools. However, most of existing NVMM debugging tools, such as PMemCheck [45], PMReorder [81], and PMTest [60], share two major limitations: 1) lack of static analysis and 2) poor multi-threading support. Without static analysis, tools must rely on instrumenting an NVMM program and running dynamic analysis on a finite set of test cases. Therefore, the instrumentation effort and test case quality significantly limit the testing coverage. Moreover, they often require programmers to manually annotate the source code with testing constructs, steepening the learning curve, reducing code readability and maintainability, and reducing portability. Finally, existing NVMM debugging tools [45, 60, 81] provide little or no support for inter-thread bug analysis. PMemCheck and PMReorder extend Valgrind [76], which serializes all threads with an emulated CPU, and does not consider interactions between threads and transactions. PMTest also lacks inter-thread analyses.

To overcome the limitations of existing NVMM debugging tools, we propose PmemConjurer and PmemSanitizer, debugging tools that combine both static and dynamic program analysis for finding bugs in NVMM applications. PmemConjurer is a static analyzer using symbolic execution to explore a program's control flow graph and search for recovery bugs. To support inter-thread analysis, PmemSanitizer adds compiler instrumentation to inject dynamic diagnosis code into the program. An instrumented program will execute natively with threading, while PmemSanitizer's runtime library performs inter-thread analysis and store-reordering tests. This chapter makes the following contributions:

1. **Introduce static analysis for finding recovery bugs.** PmemConjurer is the first tool applying static analysis to debugging NVMM applications. We extend the Clang Static Analyzer's symbolic execution [61] to support recovery bug analysis.

2. **Detect recovery bugs in multi-threaded programs.** PmemSanitizer supports multi-threaded programs by injecting thread-safe analysis code into the target binary and performing runtime inspection.

3. **Support online store-reordering testing.** PmemSanitizer supports store-reordering tests to emulate crashes without terminating the running program.

4. **Identify new bugs in PMDK examples.** We discover eight unknown bugs in the example programs of PMDK. The maintainers have accepted our patches to fix them.

In the following sections, we first introduce the background on NVMM programming and program analysis in Section 4.1. Then, Section 4.2 presents the design overview. Section 4.3 and Section 4.4 describe implementation details for static and dynamic analysis, respectively. Section 4.5 evaluates both tools' bug-finding ability and runtime performance implications. Section 4.6 compares PmemConjurer and PmemSanitizer with related work, and finally, Section 4.7 concludes.

4.1 Background

In this section, we first provide a brief primer on NVMM programming and how recovery bugs occur. Then, we describe Clang and LLVM's program analysis frameworks that are relevant to PmemConjurer and PmemSanitizer.

4.1.1 NVMM Programming

NVMM programming faces challenges from two aspects: 1) a store to NVMM is not guaranteed persistent due to volatile caches and on-CPU buffers, and 2) the order in which stores actually update NVMM may not correspond to the program's store ordering.

```
struct node {           |  // PMEMoid is from libpmemobj
  int in_use;           |     typedef struct pmemoid {
  uint64_t key;         |       uint64_t pool_uuid_lo;
  char val[128];        |       uint64_t off;
  PMEMoid next;         |     } PMEMoid;
};                      |
-------------------------------------------------------
void insert(PMEMobjpool *pop, uint64_t key, char *val) {
  PMEMoid head = pmemobj_root(pop, sizeof(struct node));
  struct node *pnode = pmemobj_direct(head);

  while (pnode->in_use && !OID_IS_NULL(pnode->next))
    pnode = pmemobj_direct(pnode->next);

  if (pnode->in_use) {
    TX_BEGIN(pop) {
      PMEMoid new_node = pmemobj_tx_alloc(/* args */);
      TX_ADD_DIRECT(&pnode->next); // Undo logging
      pnode->next = new_node;
      pnode = pmemobj_direct(new_node);
      pnode->in_use = 0; // Initializing the new node
      pnode->next = OID_NULL;
    } TX_END
  }

  pnode->key = key;
  strncpy(pnode->val, val, 128);
  pmem_flush(pnode, sizeof(struct node)); // Flush
  pmem_drain(); // Memory barrier
  pnode->in_use = 1;
  pmem_persist(&pnode->in_use, 4); // Flush & Barrier
}
```

Figure 4.1: An NVMM programming example - The `insert` function finds the first unused node in an NVMM linked list to store a key-value pair.

Figure 4.1 demonstrates an NVMM programming example using functions from PMDK to tackle these challenges. The `insert` function finds the first unused node in an NVMM linked list to store a key-value pair. If all nodes are in use, it allocates a new node and inserts it to the tail with a transaction.

PMDK is a collection of multiple libraries. `Libpmem` provides low-level functions (named with the `pmem` prefix) wrapping machine-dependent instructions for cache-line operations. `Libpmemobj` functions (with the `pmemobj` prefix) are higher-level, object-oriented, and support memory management and transactions. This chapter focuses on `libpmem` and `libpmemobj` and we refer to them collectively as PMDK.

NVMM pointers and objects. `Libpmemobj`'s programming model uses `PMEMoid` instead of a native C-style pointer to refer to persistent objects. A `PMEMoid` consists of an 8-byte file UUID and an 8-byte offset that gives its location within a memory-mapped file. The function `pmemobj_direct` converts a `PMEMoid` into a `void *` pointer for direct memory access. An object type for NVMM (e.g., `struct node`) is mostly the same as its conventional form, except pointer fields are `PMEMoid` type.

Atomic NVMM updates. The insertion logic on lines 26 - 31 uses `libpmem` functions for crash consistency. It first ensures a node's key-value data is persistent by explicitly flushing updates (line 28) and enforcing memory ordering (line 29). It then atomically sets the `in_use` flag to validate the node's data (lines 30 and 31). Since x86 processors guarantee aligned, 8-byte stores atomically update NVMM [44], a crash between lines 26 - 31 either results in a fully populated node or an unused one ready for a new key-value pair.

Libpmemobj transactions. Allocating a new node and inserting it to the linked list involves object allocation and modifications more complex than a single 8-byte write. `Libpmemobj` provides the transactional interface illustrated in lines 16 - 23. `TX_BEGIN` and `TX_END` macros wrap library functions to control a transaction's progression. Inside the transaction, the program

does not have to invoke flushing or ordering functions explicitly. Instead, it must make calls to `TX_ADD_DIRECT` (or similar ones) to create undo-logs before modifying any existing NVMM data (line 19). The undo log-based transaction mechanism serves two purposes: 1) flushing logged (hence modified) ranges on transaction commit, or 2) restoring the logged data if a transaction aborts or crashes. A newly allocated object (line 17) does not require logging, but a transaction should initialize it (lines 21 and 22) before it commits. Transactions can nest, and the outer-most transaction commits all nested ones.

Transactions and threading. Multi-threading interacts with `libpmemobj`'s transactions in two ways: 1) a transaction spawns and joins concurrent worker threads, or 2) concurrent worker threads create thread-local transactions. In the first case, a `libpmemobj` transaction commits NVMM updates from all threads as a group, and we refer to them as "grouped transactions." Transactions of the second form are mutually independent, so we call them "independent transactions."

Examples of recovery bugs. Considering Figure 4.1 without one or more of the lines with comments illustrates many potential bugs. Without undo logging (line 18), a crash before the transaction commit can render the old data of `pnode->next` unrecoverable. Dropping the flush or memory barrier on lines 28 and 29 may cause `pnode->in_use` to be true in NVMM before the key-value pair becomes persistent, mislabeling invalid data as valid. We define more types of recovery bugs in Section 4.2.

4.1.2 Program Analysis Frameworks in Clang and LLVM

PmemConjurer extends the symbolic execution of the Clang Static Analyzer (CSA) [61] to find recovery bugs and PmemSanitizer adds an LLVM [56] IR-level instrumentation pass to inject dynamic analysis code into the target binary. Below, we briefly introduce their operation. Although many CSA- and LLVM-based debugging tools exist, as far as we know, no prior works

have applied them detecting NVMM programming bugs.

Clang Static Analyzer. CSA works on a program's control flow graph (CFG) before generating LLVM's intermediate representation (IR). CSA's core engine uses symbolic execution to simulate a program's execution by traversing its CFG and progressively building a graph of reachable program states. Each node of the graph corresponds to a program point in the source code and contains information about the simulated program's state.

CSA implements a region-based memory model [106], and during the simulated execution, it interprets variables and memory objects as symbolic values or symbolic memory regions. In Figure 4.1, `head` represents a variable region of function `insert`'s stack frame, and it stores a symbolic value (`PMEMoid` type) returned by `pmemobj_root`. Each pointer symbol in CSA corresponds to a memory region of a program's address space. Taking the address of a stack-allocated variable (e.g., `&head`) generates a pointer for a call-stack region, and the return value of `malloc` or `new` represents a heap region. Generally, a pointer returned by a function unknown to CSA (the case of calling `pmemobj_direct`) represents an unknown region.

Symbolic values may have constraints depending on the execution path. For example, the value of `pnode->in_use` is unbounded (can be any 32-bit value) before the `if` statement. Symbolic execution bifurcates at conditional branches and associates path constraints with symbolic values. Therefore, the true-path of the `if` statement constraints `pnode->in_use` to be non-zero values and the false-path bounds it to zero.

CSA handles loops by iterating a constant number of times (by default 4). For each iteration, if CSA can deterministically resolve the termination condition, it either continues within the loop or breaks out accordingly. Otherwise, CSA assumes both paths are possible and associates path constraints with related symbolic values. For the `while` loop in Figure 4.1, CSA analyzes both paths for every iteration because values of `pnode->in_use` and `pnode->next` are not statically available.

On top of CSA's symbolic execution engine, it allows a checker plugin to attach to various

Table 4.1: Rules that PmemConjurer or PmemSanitizer automatically checks for correct NVMM programming

Rule	Description	Statically Checkable
1	A program must partially or fully flush any modified NVMM object.	Yes
1a	A flush operation's target must be an NVMM address.	Yes
1b	The flushed size must be no less than the target's size (if typed).	Yes
2	Code in a `libpmemobj` transaction must log an object before modifying it.	Yes
3	A transaction must initialize newly allocated NVMM objects before commit.	Yes
4	A program must not use effectively deallocated NVMM objects.	Yes
5	Independent transactions must not log the same object or NVMM range.	No
6	Reordering stores in-between memory barriers must not cause unrecoverable inconsistency.	No

program points in the source code by implementing callback functions. These program points can be before or after making a function call, reading or writing a memory location, and at the exit of a function, etc. A checker's callback function can implement debugging logic to analyze the program's current state. Multiple checkers exist for finding conventional programming errors, such as null pointer dereferencing and use-after-free.

LLVM instrumentation and dynamic analysis. Before the machine-code generation, LLVM can instrument a program's IR for runtime profiling or debugging. An LLVM's instrumentation pass injects probing functions before or after its interested program statements, and the linker links instrumented object files with a related runtime analysis library. A widely adopted set of tools based on this principle is the sanitizer series. For example, AddressSanitizer [94] and ThreadSanitizer [20] detect out-of-bound memory accesses and data races, respectively. A user can enable a particular sanitizer tool with compile-time and link-time switches for Clang.

4.2 Design Overview

PmemConjurer extends CSA's symbolic execution to perform NVMM-specific static analysis and PmemSanitizer adds a sanitizer-like dynamic debugging tool to complement the static analyzer with multi-threading support and store-reordering tests. Figure 4.2 illustrates their components in the compilation flow. They perform program analysis at two independent phases:

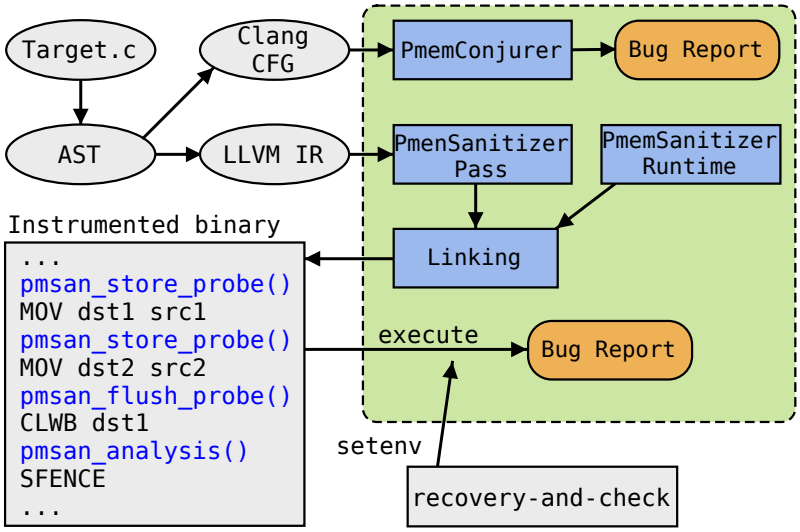

Figure 4.2: PmemConjurer and PmemSanitizer design overview

the static analyzer runs on each source file's control flow graph before LLVM IR generation, and the dynamic analysis runs when executing an instrumented binary. A user can selectively enable them either with compile-time switches or runtime environment variables.

In this section, we first describe the programming rules that PmemConjurer or PmemSanitizer checks for detecting recovery bugs and then introduce how to use them when developing NVMM applications.

4.2.1 Automatically-checked NVMM Programming Rules

The rules are based on the NVM programming model [95] and PMDK's library function semantics. PmemConjurer or PmemSanitizer detects rule violations and reports them as recovery bugs. Table 4.1 presents a summary of the rules. In the description below, we refer to lines in Figure 4.1 to show examples of recovery bugs.

Rule-1: A program must partially or fully flush any modified NVMM object. Outside a transaction, a program should explicitly flush modified NVMM ranges at some point (see below). Otherwise, the updates may still reside in CPU caches. Inside a transaction, we check

Rule-2 instead of Rule-1 because the transaction mechanism automatically flushes modified ranges (see Section 4.1.1).

PmemConjurer and PmemSanitizer both track written NVMM regions and check this rule at memory barriers (e.g., line 29 and 31). Because the static analyzer works on each source file independently, it also checks this rule when a top-level function (without a caller in the same source file) returns. PmemSanitizer also checks it when a running program terminates.

The size of the area that needs to be flushed can be challenging for a debugging tool to infer because we account for transient fields (i.e., non-persistent) in an otherwise-persistent object. We adopt different strategies for static and dynamic analysis. PmemConjurer assumes any modified NVMM object should be flushed at least once, despite the offset and size, at the next verification point. PmemSanitizer adopts a more aggressive strategy: the flushed ranges should fully cover modified NVMM data. To compensate for possible missed bugs in PmemConjurer, we define two additional statically checkable rules for flushing: Rule-1a and Rule-1b.

Rule-1a: A flush operation's target must be an NVMM address. Flushing non-NVMM address ranges, such as the call stack or `malloc`'ed memory, does not, in itself, threaten crash consistency, but it is pointelss and it may indicate a programming error (e.g., if the target of the call should have been NVMM data). For instance, writing line 28 to `pmem_flush(&pnode, ..)` violates this rule, because it flushes the variable `pnode`'s stack address rather than its pointed NVMM location.

Rule-1b: The flushed size must be no less than the target's size (if typed). When a flush function's target is a typed region (lines 28 and 31), we expect the flushed size to be no smaller than the data type's size. For instance, a violation occurs if line 28 changes to `pmem_flush(pnode, sizeof(pnode))`, because the data type that `pnode` points to is `struct node`, but the flushed size is the size of a single pointer.

Rule-2: Code in a `libpmemobj` transaction must log an object before modifying it.

Modifying existing NVMM objects without logging may cause unrecoverable data inconsistency if a crash happens before the transaction commits. Similar to flush checking, PmemConjurer expects at least logging some field of a modified object, and PmemSanitizer checks if all modified data has been logged.

Rule-3: A transaction must initialize newly allocated NVMM objects before commit. Some of `libpmemobj`'s allocation functions (line 17) do not zero-out the memory block so it may contain garbage data. When the program allocates an object in a `libpmemobj` transaction, PmemConjurer checks for two types of actions before the transaction commits: 1) the transaction allocates and initializes it using library functions like `pmemobj_tx_zalloc`, or 2) the transaction writes to some field of the object. If neither of the two presents (e.g., removing lines 21 - 22), it reports a bug.

Rule-4: A program must not use effectively deallocated NVMM objects. Use-after-free bugs are a common problem, but NVMM complicates their detection within transactions because deallocation does not take effect until a transaction commits. PmemConjurer models this deferred behavior and detects accesses to effectively freed objects.

Rule-5: Independent transactions must not log the same object or NVMM range. Independent transactions are thread-local, and each worker function has its own undo logs. Allowing two threads to log the same data may compromise data consistency if one transaction commits its update, while the other fails and reverts the same object to a logged version. This recovery bug does not always imply a data race, so conventional data racing checkers are not sufficient.

Rule-6: Reordering stores in-between memory barriers must not cause unrecoverable inconsistency. This rule detects missing memory barriers, (e.g., omitting line 29). Because stores between two memory barriers do not have ordering constraints, a crash may result in any subset of the stores becoming persistent. Thus, if any subset of the stores causes

```
# PmemConjurer - Static analysis
clang --analyze -Xanalyzer -analyzer-checker=pmem src1.c
clang --analyze -Xanalyzer -analyzer-checker=pmem src2.c

# PmemSanitizer - Dynamic analysis
clang src1.c src2.c -O3 -ggdb -fsanitize=pmem -o target
PMSAN_OPTIONS="rcprog='./recovery-and-check nvmmfile' reorder=
    RevertSingle" ./target nvmmfile
```

Figure 4.3: Examples using PmemConjurer and PmemSanitizer

unrecoverable inconsistency, a program's implementation is not crash-consistent. Checking this rule requires the user to provide a consistency-checking program for PmemSanitizer's dynamic analyzer to run for each ordering test case. We explain the details of this testing method in Section 4.4.

4.2.2 Using PmemConjurer and PmemSanitizer

We design PmemConjurer and PmemSanitizer to be easily adoptable. The target users are developers using `libpmem`, `libpmemobj`, or compiler intrinsics for cache operations. It automatically checks the programming rules above without need of annotating the source code.

A user can enable the PmemConjurer static analyzer by specifying `pmem` as the checker name when invoking Clang Static Analyzer. CSA supports inter-procedural analysis by inlining callee functions, but currently, it does not support cross-file analysis. Thus, PmemConjurer also has to analyze each source file independently. We add `-fsanitize=pmem` to Clang and Clang++ as a compile-time switch for enabling PmemSanitizer's IR-level instrumentation. The same switch also works for linking PmemSanitizer's runtime analysis library.

An PmemSanitizer-instrumented binary checks the `PMSAN_OPTIONS` environment when it starts running (before entering `main`), Its value configures PmemSanitizer's runtime library, including the recovery-and-check program, its arguments, and reordering strategy. Figure 4.3 illustrates how to use PmemSanitizer at compile time and run the instrumented binary.

4.3 PmemConjurer

PmemConjurer modifies the core of CSA and implements an NVMM-specific checker with the following capabilities.

1. Identify NVMM regions via pointer symbols.
2. Track NVMM objects via `PMEMoid` and derived symbols.
3. Emulate NVMM-specific functions and memory accesses.
4. Analyze recovery bugs at various program points.

In this section, we explain some implementation details of PmemConjurer's static analyzer design.

4.3.1 NVMM Regions and Region Symbols

CSA does not distinguish stores to NVMM and other memory regions. To resolve it, we modify CSA's core to add a new type of memory region called *NVMMSpaceRegion* and designate certain functions as returning pointers to NVMM regions, including `pmem_map_file`, `pmemobj_open`, and `pmemobj_direct`. PmemSanitizer's static checker attaches to these function call sites during symbolic execution and creates a pointer-type symbol (*RegSym*) representing an NVMMSpaceRegion for the returned address. PmemConjurer assumes that pointers from other sources (e.g., taking the address of a stack variable or calling `malloc`), point to volatile memory.

In Figure 4.1, `pmemobj_direct` on line 10 returns a RegSym. The assignment on line 10 stores the symbol to the stack variable `pnode`. CSA can extract this symbol stored in `pnode` until reassigning it on line 13 or 20. If another variable copies `pnode`'s value, it acquires the same RegSym. A derived pointer (e.g., `&pnode->next`) inherits its origin's NVMMSpaceRegion property, and PmemConjurer can extract the RegSym for the underlying pointer.

4.3.2 NVMM Objects and Object Symbols

We track NVMM objects using `PMEMoid` symbols and their derived symbols. A `PMEMoid` symbol can originate from multiple sources, for instance, function's return value, function's arguments, and struct fields or array elements of `PMEMoid` type (e.g., `pnode->next`). PmemConjurer creates new NVMM object symbols (*ObjSym*) for function-returned `PMEMoid` values (e.g., `pmemobj_root` and `pmemobj_tx_alloc`). Reading a `PMEMoid`-typed memory location does not always generate a new ObjSym. For example, reading `new_node` on line 19 and 20 retrieves the ObjSym created for `pmemobj_tx_alloc`. In contrast, the first read from a memory location with unknown value (e.g., `pnode->next`) creates a new ObjSym, and the next reading from the same location gets the same ObjSym before reassigning `pnode`.

Derived ObjSyms mainly come from casting a `PMEMoid` to a C union termed `TOID` [80] (short for typed `PMEMoid`) by `libpmemobj` for encoding NVMM object types. CSA currently does not handle casting-to-union statements, and PmemConjurer lifts this restriction by creating a new ObjSym for the casting result and mapping it to the same RegSym of the `PMEMoid` symbol (see below).

4.3.3 NVMMRegionState and Symbol Mappings

One challenge for symbolic execution to analyze NVMM-specific code is that a program can access the same NVMM object either via a `PMEMoid` variable or its raw pointer, as shown in Listing 4.1. By default, CSA would create a new symbol as the return value for each `pmemobj_direct` call, ignoring the correlation between `ptr1` and `ptr2`. PmemConjurer avoids this problem by creating an *NVMMRegionState* data structure and accessing them via a two-level mapping mechanism.

Each RegSym corresponds to an NVMMRegionState data structure (Figure 4.4) describing the region. NVMMRegionState tracks the type of its NVMM region. *MapRegion* represents

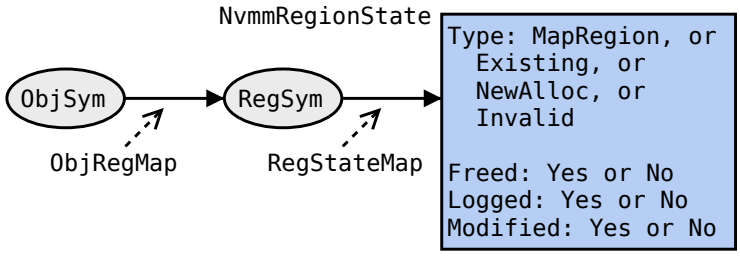

Figure 4.4: NVMMRegionState and two-level symbol mapping. This mechanism helps PmemConjurer recognize changes to the same NVMM region via either `PMEMoid` or pointer variables.

the region of a mapping function call (e.g., `pmem_map_file` or `pmemobj_open`) that is not related to a particular object. The *Existing* and *NewAlloc* types correspond to objects from different sources (see Section 4.3.4). A region's type becomes *Invalid* after unmapping or deallocation. NVMMRegionState also records various properties that change according to program statements.

```
1 struct node *ptr1 = pmemobj_direct(head);
2 struct node *ptr2 = pmemobj_direct(head);
3 TX_ADD(head);  // Undo-log: ObjSym for the head node
4 ptr1->in_use = 0; // Store: RegSym + offset_of(in_use)
5 ptr2->next = OID_NULL; // Store: RegSym + offset_of(next)
```

Listing 4.1: PmemConjurer recognizes the two pointers referencing the same memory location, and the last three lines operate on the same object.

Two-level symbol mapping uses one hashmap (*ObjRegMap*) to map an ObjSym to a RegSym, and another hashmap (*RegStateMap*) redirects a RegSym to an NVMMRegionState instance. We model `pmemobj_direct` so that it queries the ObjRegMap to retrieve and return an existing RegSym as its result pointer. If the mapping does not exist, we know that an untracked, existing object appears in the program and create a new set of ObjSym, RegSym, and NVMMRegionState mappings for it. When a program statement reads or writes memory locations, PmemConjurer extracts the RegSym from the address variable and uses the RegStateMap to retrieve its NVMMRegionState. Functions operating on `PMEMoid` query the ObjRegMap followed

Table 4.2: Program statements, NVMMRegionState transitions, and conditions for rule violations

Program statement	NVMMRegionState transitions	Rules to check and conditions for violations
Memory read	N/A	Rule-4: Destination region's type is Invalid.
Memory write	Modified: No → Yes	Rule-4: Destination region's type is Invalid.
		Rule-2: (TxLevel > 0) Object is Existing type but not logged.
Any flush function	Modified: Yes → No	Rule-1a: Destination's address is not from NVMMSpaceRegion.
Flush function with size	Modified: Yes → No	Rule-1b: Flushed size is less than the destination's type size.
Memory barrier	N/A	Rule-1: Any tracked NVMMRegionState is modified.
Top-level function return	N/A	Rule-1: Any tracked NVMMRegionState is modified.
Unmapping function	Type: MapRegion → Invalid	N/A
`TX_BEGIN`	TxLevel → TxLevel + 1	N/A
`TX_ADD` (or similar)	Logged: No → Yes	N/A
`TX_FREE`	Freed: No → Yes	N/A
`TX_END` (TxLevel ¿ 1)	TxLevel → TxLevel - 1	N/A
`TX_END` (TxLevel == 1)	TxLevel → 0	Rule-3: Any tracked object is NewAlloc type but not modified.
	Type: NewAlloc → Existing	Rule-3: Any tracked object is NewAlloc type but not modified.
	Type: Any (Freed) → Invalid	Rule-3: Any tracked object is NewAlloc type but not modified.
	Logged: Yes → No	Rule-3: Any tracked object is NewAlloc type but not modified.
	Modified (Logged): Yes → No	Rule-3: Any tracked object is NewAlloc type but not modified.

by the RegStateMap. This mechanism ensures the last three lines of Listing 4.1 update and analyze information of the same object.

4.3.4 Emulating Function Calls and Memory Accesses

PmemConjurer recognizes NVMM-specific functions and models their behaviors according to their semantics. Most `libpmemobj` functions that return `PMEMoid` retrieve existing NVMM objects (Existing type). The exceptions are allocator functions which make new ones (NewAlloc type). `TX_ADD` and `TX_FREE` functions[1] update the *Modified* and *Freed* properties of an NVMMRegionState, respectively. Memory writes and allocator functions that zero-out a new object set the *Modified* property, and a flush function or a transaction commit resets it.

In addition to tracking region and object symbols, PmemConjurer also tracks the transaction context during symbolic execution using a variable called TxLevel. It increments with a `TX_BEGIN` function and decrements at `TX_END`. The `TX_END` for the outermost transaction (TxLevel == 1) commits all nested transactions. At commit, PmemConjurer iterates over its tracked NVMMRegionState instances and update them if necessary. Table 4.2 presents details of NVMM-

[1]We use short macro names during the discussion but CSA actually analyzes the underlying functions.

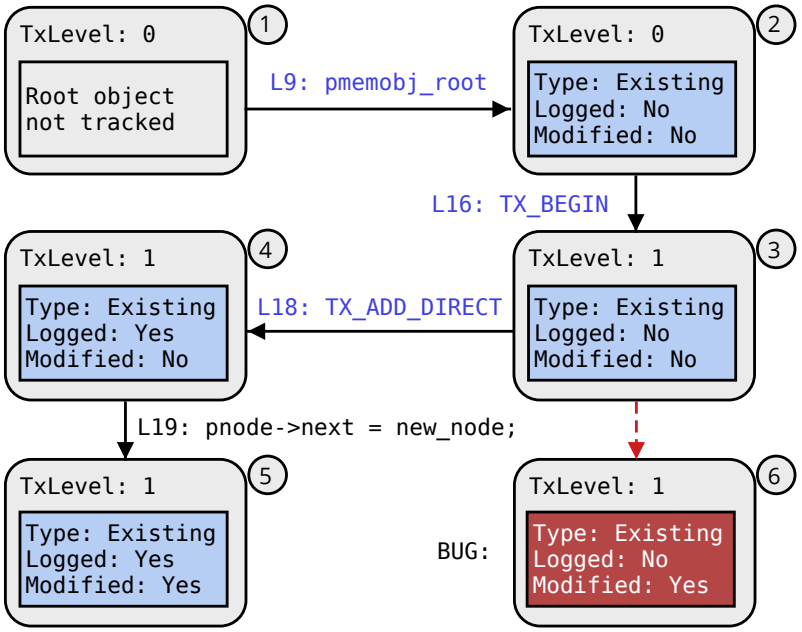

Figure 4.5: PmemConjurer's analysis flow for the root object in Figure 4.1.

RegionState transitions.

4.3.5 Rule-checking with NVMMRegionState

The state that PmemConjurer tracks during symbolic execution allows it to enforce most of the rules listed in Section 4.2.1. Table 4.2 describes the points at which PmemConjurer checks for rule violations. Figure 4.5 illustrates CSA's symbolic execution and PmemConjurer's analysis flow for the "root" object (the list's head node) in Figure 4.1. Each block in Figure 4.5 shows how PmemConjurer tracks program state, including the transaction-nesting level and the root object's NVMMRegionState. The state transition corresponds to the execution path following lines 9 - 12, without going into the `while` loop and then taking the `if` statement's true-branch (lines 15 - 19).

PmemConjurer starts tracking the root object when `pmemobj_root` returns and treats it as an existing object. Variables `head` and `pnode` get the root object's ObjSym and RegSym, respectively. PmemConjurer checks Rule-2 when memory writes happen, and TxLevel is non-zero. If a memory writer's target address derives from a tracked RegSym, and its NVMMRegionState

is not logged, PmemConjurer reports a violation. State 6 shows an error state caused by leaving out line 18 of Figure 4.1.

4.3.6 Limitations of PmemConjurer's Static Analysis

Although PmemConjurer's static analysis can find many kinds of bugs, it does have limitations in reasoning about thread interactions and store ordering constraints.

PmemConjurer's symbolic execution does not consider interactions between threads. Thus, it does not detect a violation of Rule-5 since it cannot decide if two object symbols from two separate threads reference the same NVMM object.

The static analysis cannot infer store ordering constraints. Some NVMM programs require explicit memory barriers to enforce the ordering about when stores become persistent, such as using `pmem_drain` in Figure 4.1. Store ordering constraints are difficult for static analysis to infer because the code lacks description about its logic to maintain crash consistency. For example, it cannot decide if there should be a memory barrier between stores to `pnode->key` and `pnode->val`, because that depends on what actions the recovery code takes after a crash.

PmemSanitizer's dynamic analysis complements PmemConjurer to overcome these limitations by providing multi-threading support and store-reordering testing for detecting ordering-related bugs.

4.4 PmemSanitizer

PmemSanitizer aims to complement PmemConjurer by providing multi-threading and store ordering analysis. It also performs more precise checks on flushing and logging sizes. PmemSanitizer uses an LLVM IR instrumentation pass to inject probe or analysis functions before program statements that access or manage NVMM. Figure 4.2 shows some of the inserted functions as they appear in the target program's assembly. The probe functions are thread-safe so

that PmemSanitizer can analyze concurrent programs.

4.4.1 Instrumentation and Runtime Analysis

PmemSanitizer probes PMDK functions that memory-map NVMM to get the actual address and size of each mapped NVMM region. It also probes all memory writes, cache flushes, and transaction logging functions to collect the range of addresses they affect. PmemSanitizer's probe function collects thread-local information and assigns each operation a timestamp to track ordering among them. For a store operation, such as a variable assignment or calling a memory-altering function (e.g., `memcpy` or `memset`), PmemSanitizer records its destination's old contents for reordering tests (see Section 4.4.3).

PmemSanitizer performs rule-checking at two points: 1) when a transaction commits, or 2) at a memory barrier. When a transaction is about to commit (e.g., just before calling `pmemobj_tx_commit`), PmemSanitizer gathers modified NVMM ranges from all threads and checks if they have overlaps (violating Rule-5). It also verifies Rule-2 with more accuracy than the static analysis. At a memory barrier, PmemSanitizer verifies Rule-1 with concrete store/flush sizes and performs reordering tests to check Rule-6 (see below).

4.4.2 Supporting Threaded Programs

PmemSanitizer's dynamic analysis supports multi-threaded programs by collecting thread-local information in each probe function and retrospectively analyzing their interactions in an analysis function (any thread).

For coordinating probe and analysis functions, PmemSanitizer uses a global reader/writer in an unconventional way: probe functions take the "reader" access to collect thread-local information, while the analysis function takes "writer" access before starting analyzing, so other threads cannot perform stores to NVMM or any instrumented NVMM-specific operations.

This technique allows the concurrent execution of multiple program threads. It does have the limitation of not catching a bug if a particular runtime schedule serializes all threads' execution, but this rarely happens during our evaluation on modern multi-core machines. In contrast, PMemCheck or PMReorder runs a target program using Valgrind, which serializes threads [76] and does not consider interactions between threads.

4.4.3 Online Reordering Tests

PmemSanitizer performs store-reordering tests at memory barriers to test the program's ability to recover after a crash. Figure 4.6 is the implementation of PmemSanitizer's store-probe and ordering-testing functions. Function `pmsan_probe_store` runs before each store and assigns a timestamp to each store operation and records the store destination's old contents. In `pmsan_analysis`, it gathers stores from all threads and selectively (see below) reverts some of them to emulate a crash before this memory barrier takes effect (when all stores become persistent).

```
RevertSingle: (), (A, B), (A, C), (B, C), (A, B, C)
RevertAccumulative: (A, B, C), (B, C), (C), ()
```

Listing 4.2: Reordering test cases for three stores: A, B, and C.

Reordering strategy. At runtime, the "reorder" value of `PMSAN_OPTIONS` dictates PmemSanitizer how to revert stores. PmemSanitizer supports two strategies: "RevertSingle" reverts each individual store, and "RevertAccumulative" reverts stores accumulatively in the order specified by their timestamps. Supposing `A`, `B`, and `C` represent the three stores on line 26, 27, and 30 of Figure 4.1 and that the memory barrier on line 29 is missing, Listing 4.2 demonstrates test cases generated for them using different strategies. Although the program executes the stores in sequence, they do not necessary become persistent in the same order. Each tuple in Listing 4.2 indicates a test case where only shown stores are persistent when a crash happens. Function

```
pmsan_store_probe(uint64_t dst, size_t size) {
  if (!is_nvmm_range(dst, size))
    return;
  std::shared_lock probe(pmsan_lock);
  record_dst_contents(dst, size);
}
```

(a) Probe function for stores

```
pmsan_analysis() {
  std::unique_lock analysis(pmsan_lock);
  gather_thread_local_stores();
  while (next_reorder()) {
    pit_t child = fork();
    if (child == 0)
      execv("recovery-and-check", args);
    waitpid(child, &status, 0);
    if (!exited_normally(status))
      report_ordering_bug();
  }
}
```

(b) Analysis function (only showing the reordering logic)

Figure 4.6: Reordering tests implementation (gists only).

`next_reorder` (Figure 4.6-b) generates a sequence of all test cases according to the selected strategy.

Online testing. For each test case, the parent process forks to run the recovery-and-check program (specified via `PMSAN_OPTIONS`) in the child process. The recovery-and-check program is also PmemSanitizer-instrumented, and it communicates with the parent via shared memory. For each store to revert of a test case, the child restores the contents of its affected range before the store happened, effectively emulating a store that did not become persistent because of a crash. Then, the child process executes the recovery code and the consistency-check routine. The parent process reaps the child and checks its exit status to determine whether it recovered successfully.

Recovery-and-check program. The recovery-and-check program should perform crash recovery and then apply a consistency check on the resulting data. For Figure 4.1, the pro-

gram would verify the following: when `pnode->in_use` is true, the string length of `pnode->val` should be non-zero. Those checks would detect inconsistent results produced by test cases `(C)` or `(A, C)`.

PmemSanitizer has two additional requirements for the recovery-and-check program: 1) it must memory-map the same NVMM file with flag `MAP_PRIVATE`, and 2) it must return zero from `main` for a successful recovery or non-zero for failure. The first requirement prevents the recovery procedure from modifying the shared NVMM data. The second one lets PmemSanitizer recognize a successful recovery by checking the child process' exit status.

After each test case, PmemSanitizer reports a bug if the test case is not recoverable. In any case, it continues producing any remaining test cases.

4.5 Results

We evaluate PmemConjurer and PmemSanitizer from the following perspectives:

1. Can they detect a wide range of recovery bugs?
2. What is the impact on compilation and execution time?
3. How frequent are the false-negatives and false-positives?

4.5.1 Detecting Recovery Bugs

We evaluate PmemConjurer' and PmemSanitizer's bug detection abilities using both fabricated bugs and known ones from the PMDK repository. To generate test cases, we use `libpmem`, `libpmemobj`, and compiler intrinsics to build test programs (e.g., Figure 4.1), and then manually introduce bugs for testing. They detect all of the fabricated bugs except those crafted specifically to elicit a false-negative bug detection. To test them on real-world bugs, we searched PMDK's commit messages (more than 8000) for keywords like "fix" or "bug" to find

Table 4.3: Detecting previously fixed bugs in PMDK examples - We browsed PMDK's commit history and found 19 recovery bugs, including ones described by PMTest [60]. PmemConjurer and PmemSanitizer can identify all of them.

Violation	Short description	Bugs
Rule-1	Missing flush	1
Rule-2	Missing logging	5
Rule-3	No initialization	13
Rule-4	Use-after-free	0
Rule-5	Logging race	0
Rule-6	Missing barrier	0
Not checkable by our tools		0
Total		19

Table 4.4: New bugs detected by PmemConjurer and PmemSanitizer in PMDK examples

Violation	Short description	Source files	Bugs
Rule-1	Missing flush	array.c [34]	1
Rule-1a	Flushing stack	obj_cpp_ptr.cpp [28]	1
Rule-1b	Incomplete flush	pi.c [29], hashmap_rp.c [33]	2
Rule-2	Missing logging	rtree_map.c [30, 32]	3
Rule-4	Use-after-free	hashmap_tx.c [31]	1

previously fixed recovery bugs. We also tested PmemConjurer and PmemSanitizer against the bugs described in the evalutation of PMTest [60]. Table 4.3 summarizes the 19 bugs we collected from PMDK commit logs (excluding new ones found by us). We validate that PmemConjurer and PmemSanitizer can correctly identify all of them. Its static analyzer overlooks one of them (see Section 4.5.3), but the dynamic analyzer catches it.

To search for new bugs, we ran PmemConjurer's static analyzer on 42 C and 24 C++ source files containing example code from the `pmem/pmdk` and `pmem/pmdk-examples` GitHub repositoires. We found eight new bugs. Table 4.4 summarizes them. PmemSanitizer's dynamic analysis also finds the Rule-1 and Rule-2 violations in Table 4.4. We have submitted patches to fix the bugs to the PMDK maintainers and they have accepted all of them.

We have not found bugs violating Rule-5 or Rule-6, partially due to practical programs using independent transactions or just relying on store ordering for crash consistency are rare. We

Figure 4.7: Reletive slowdown using different dynamic analysis tools - PmemSanitizer's dynamic analysis causes 4.5× slowdown on average.

have created our own test programs with these bugs and PmemSanitizer can successfully detect them.

Bugs appear less often in PMDK's C++ programs and our investigation shows C++ bindings for `libpmemobj` have made it less error-prone. For example, in a transaction implemented in C++, the programmer does not have to explicitly invoke logging functions before modifying NVMM objects. The C++ library provides smart pointers with operator overloading to perform object-level logging automatically.

4.5.2 Performance

PmemConjurer's static analyzer runs on programs' control flow graphs before generating the LLVM IR. Its analysis time varies between less-than-one to around ten seconds depending on each source file's length and complexity of its CFG. It works offline and does not impact the program's runtime performance.

We evaluate the performance impact of PmemSanitizer's dynamic analysis on a machine with one 8-core Xeon E3-1270v6 and 32 GB main memory. The CPU provides the `CLFLUSHOPT` instruction for flushing modified cache lines to NVMM's persistence domain, and the `SFENCE` instruction to ensure memory ordering. We configure 16 GB of the memory to emulated NVMM.

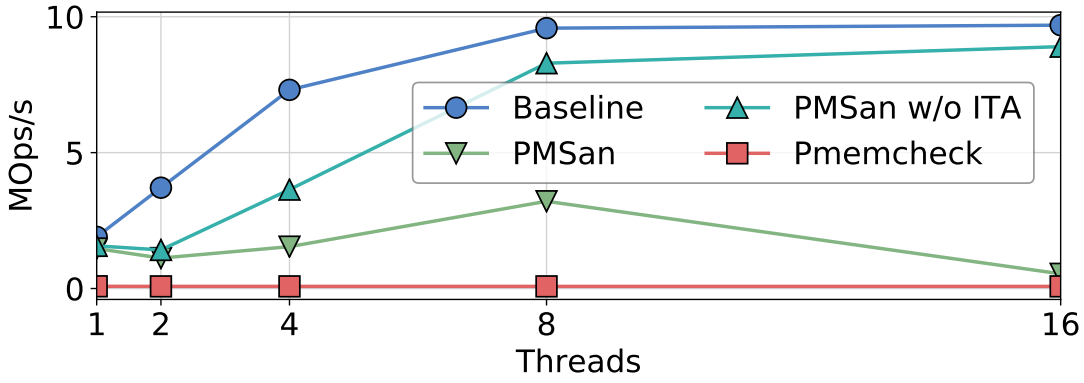

Figure 4.8: Scalability with independent transactions modifying 64-byte objects - Baseline indicates the throughput of the benchmark without any debugging instrumentation.

We run NOVA [104] as the DAX-enabling file system, and applications use `mmap()` to access NVMM-resident files.

We use seven mapping data structures (Figure 4.7) and their benchmark programs included in the latest PMDK repository to measure PmemSanitizer's performance and compare it to the baseline, PMTest, and PMemCheck.

The baseline contains normally-compiled benchmarks without any instrumentation. We evaluate PmemSanitizer by instrumenting the benchmarks at compile-time and running them on the test machine. PMemCheck is an extension to Valgrind, and we run the baseline programs on top of Valgrind with the tool enabled. PMTest requires manually instrumenting both `libpmemobj` and the benchmark programs, and it only supports an old version of PMDK. Therefore, we report relative slowdown factors for each tool, compared to their corresponding baseline.

Figure 4.7 shows PmemSanitizer's dynamic analysis causes 4.5× slowdown on average. That is 3.3× slower than PMTest since PMTest only instruments NVMM writes but PmemSanitizer also has to filter DRAM stores at runtime. PMTest does not have results for rtree and skiplist since they do not exist in its supported PMDK version. PMemCheck is more than 10× worse than the other two due to Valgrind's emulated execution.

We also create a threaded benchmark to evaluate PmemSanitizer's impact. In Figure 4.8,

each thread executes one transaction that is independent of other threads and overwrites an existing 64-byte NVMM object, and we vary the number of threads. The instrumented program does not scale well because PmemSanitizer acquires a global mutex (see Section 4.4.2) to perform inter-thread analysis when any thread commits a transaction.

4.5.3 False-Negatives

```
1 static PMEMoid alloc_int(size_t size) {
2   for (int i = 0; i < size; i++)
3     D_RW(array)[i] = i;
4   pmemobj_persist(pop, D_RW(array), sizeof(*D_RW(array)));
5   return array.oid;
6 }
```

Listing 4.3: False-negative: PmemConjurer does not report a bug because it assumes just flushing the array's first element is fine. But this function should flush a range of length `size * sizeof(*D_RW(array))`.

In our experiments, PmemSanitizer's false-negatives (i.e., a overlooked bug) are rare in real-world code. Listing 4.3 shows the only false-negative our static analyzer produced among the bugs in Table 4.3. The `pmemobj_persist` function should flush all modified elements of the array but its size parameter is only for the first element. PmemConjurer does not report partially-flushed objects because it assumes NVMM objects can contain transient fields that do not require persistence. Whether the function should flush all modified elements or just the first one is application-specific, so program analysis cannot make the correct inference without additional information.

PmemSanitizer reports any modified but unflushed range, so it does not overlook this bug as long as the program executes the function.

In general, for any dynamic analysis tool (not just PmemSanitizer), a major cause for

false-negatives is that test cases do not execise problematic code paths. PmemSanitizer's inter-thread may also miss some violations of Rule-5 if the dynamic execution schedule for two threads happens to prevent the error from occurring.

4.5.4 False-Positives

Our static analyzer reports false alarms mainly because some information is unavailable in one single source file. One example shown in Listing 4.4 shows using a function pointer to initialize a newly allocated object. Because the function pointer's implementation is in a separate source file, PmemConjurer cannot determine if it does the initialization and reports a violation of Rule-3.

```
1 int btree_map_insert_new(.., void (*constructor)(..)) {
2   TX_BEGIN(pop) {
3     PMEMoid n = pmemobj_tx_alloc(..);
4     constructor(pop, pmemobj_direct(n), ..);
5     ..
6   } TX_END
7 }
```

Listing 4.4: False-positive 1: The static analyzer cannot analyze functions that do not belong to the same source file under analysis. In this case, `constructor` points to an external function.

Listing 4.5 illustrates another false-positive case that reports unflushed stores when function `hm_rp_rebuild` returns. The function rebuilds the hashmap to extend its capacity, and it only flushes NVMM stores if the whole rebuild process succeeds. If function `entries_cache` fails by returning `-1`, the program continues from `rebuild_error` and does not flush. Having unflushed stores is fine in this case because the progam cannot access the modified NVMM data if the rebuild action fails.

We find 12 source files result in false-positives of the static analyzer from 66 totally

analyzed ones. To suppress these false alarms, we plan to provide Clang command-line options to prune execution paths that involve external functions or when a callee returns failure code.

```
1 static int hm_rp_rebuild(/* args */) {
2   if (entries_cache(pop, &hashmap_rebuld, ..) == -1)
3     goto rebuild_err;
4   pmemobj_persist(/* args */);
5   ...
6 rebuild_error:
7   // no flush action
8 }
```

Listing 4.5: False-positive 2: PmemConjurer reports unflushed stores when `entries_cache` fails and the program continues from `rebuild_error`. But the consistency of these stores does not affect correctness.

The dynamic analyzer did not produce any false-positives, although the code in Listing 4.5 could lead one.

4.6 Related Work

In this section, we discuss related approaches for detecting NVMM programming errors and compare them with PmemConjurer and PmemSanitizer.

PMemCheck [45] is a Valgrind [76] extension aiming to find recovery bugs for NVMM programs using PMDK. It employs Valgrind's binary instrumentation to trace all NVMM accesses, cache flushes, logging routines and analyze their relations. It does not require annotating source files or instrumenting the target during compilation. But it relies on Valgrind's emulation layer to trap these operations from a running binary, incuring large performance penalty. Moreover, it also executes all threads in serial and omits interactions between threads. PMReorder [81] uses store traces collected by PMemCheck to perform store ordering analysis as PmemSanitizer does.

In contrast to PmemSanitizer, it requires the user annotating source files about what stores to reorder, and it can only start testing after a target program terminates.

Yat [55] is a validation framework designed for evaluating the PMFS [22] file system. It runs PMFS using a hypervisor called Hypersim and records memory traces between memory barrier instructions. Then, it reorders stores to NVMM and replays them to emulate crashes. Yat exhaustively tests all permutations of all subsets of stores between two memory barriers. The fact of using a virtual machine environment and the exhaustive testing strategy makes it very slow, limiting its applicability to other NVMM applications.

PMTest [60] provides two low-level checkers for verifying store persistence and ordering. It also instruments `libpmemobj`'s `TX_BEGIN` and `TX_END` functions to check if stores of a transaction have been logged. Currently, PMTest requires the user manually instrumenting NVMM stores in the source code, thus demanding more effort to delpoy than PMemCheck, PmemConjurer or PmemSanitizer. PMTest can adopt PmemSanitizer's compiler-instrumentation for better automation. On the other hand, PmemSanitizer can provide PMTest's low-level checkers for the programmer to better describe persistence requirements.

4.7 Conclusion

We presented PmemConjurer and PmemSanitizer, tools for detecting recovery bugs that challenge NVMM programming. To our knowledge, PmemConjurer provides the first static analyzer for recovery bugs and proves symbolic execution is a viable solution to tackle this challenge. PmemSanitizer's dynamic analysis also introduces methods for analyzing multi-threaded NVMM programs and performing store-reordering tests in a more convenient way than existing approaches.

Our evaluation shows the effectiveness of our tool by detecting various real-life recovery bugs in the PMDK repository. Although we have been targeting the `libpmem` and `libpmemobj`

libraries to define NVMM-specific programming rules, we expect to derive similar rules for other NVMM programming libraries, and adapt our program analysis techniques for them.

Acknowledgments

This chapter contains material from "Using Static Analysis to Find Non-Volatile Main Memory Programming Bugs" by Lu Zhang, Haolan Liu, Jishen Zhao, and Steven Swanson, which is submitted to the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'20). The dissertation author is the primary investigator and first author of this paper.

Chapter 5

Conclusion

This thesis presented NOVA-Fortis, Pangolin, PmemConjurer, and PmemSanitizer to explore the unique challenges that improving PMEM software's reliability presents. We expect their design and implementation to benefit the development of reliable software for future computing platforms with persistent memory.

NOVA-Fortis provides strong protection for its metadata using a full replication. It also protects file data using space-efficient parity when user-space applications access files using the `read()` and `write()` system calls. NOVA-Fortis can benefit many applications that already use file system interfaces for accessing storage. When applications memory-map a PMEM file, NOVA-Fortis has to disable its file data protection because DAX-style `mmap()` bypasses the file system. DAX-`mmap()` provides fast access to durable data from the user-space, but it also exposes user-space software to explicitly handling PMEM errors.

To facilitate the protection of memory-mapped persistent data, the Pangolin library provides high-performance, crash-consistent, and fault-tolerant library functions to access PMEM. Pangolin uses a novel, space-efficient layout of data and parity to protect arbitrary-sized PMEM objects combined with per-object checksums to detect corruption.

NOVA-Fortis and Pangolin demonstrate that PMEM file systems and programming li-

braries can make reliability and availability guarantees while providing high performance and supporting DAX-style `mmap()`. They also make a clear case for developing unique reliability mechanisms for PMEM rather than blithely adopting schemes used by conventional disk-based storage systems.

Applications that do not use NOVA-Fortis or Pangolin may implement explicit management of crash-consistency and fault-tolerance. They can adapt the techniques we described in Chapter 2 and Chapter 3 for their specific fault-tolerance needs. To catch recovery bugs, PmemConjurer helps programmers detect them early in the development stage, without compiling the program to binary or executing it. As a complementary tool, PmemSanitizer provides compiler instrumentation and run-time reordering tests for checking a PMEM application's reliability in the testing phase. Our evaluation showed the effectiveness of our debugging tools by detecting various real-life recovery bugs in the PMDK repository. We expect PMEM application developers to adopt these tools in their development flow or use the program analysis techniques for developing alternative debugging tools.

Bibliography

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

[2] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. Operating system implications of fast, cheap, non-volatile memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, HotOS'13, pages 2–2, Berkeley, CA, USA, 2011. USENIX Association.

[3] 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.

[4] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. An Analysis of Data Corruption in the Storage Stack. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, FAST'08, pages 8:1–8:28. ACM, 2008.

[5] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri 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.

[6] Kumud Bhandari, Dhruva R Chakrabarti, and Hans-J Boehm. Implications of CPU Caching on Byte-addressable Non-volatile Memory Programming. Technical report, HP Technical Report HPL-2012-236, 2012.

[7] Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson. Bankshot: Caching Slow Storage in Fast Non-volatile Memory. In *Proceedings of the 1st Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '13, pages 1:1–1:9, New York, NY, USA, 2013. ACM.

[8] Jeff Bonwick and Bill Moore. ZFS: The Last Word in File Systems, 2007.

[9] Adrian M. Caulfield, Todor I. Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing Safe, User-Space Access to Fast, Solid State Disks. In *Proceedings*

of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

[10] Dhruva R. Chakrabarti, Hans-J. Boehm, and Kumud Bhandari. Atlas: Leveraging Locks for Non-volatile Memory Consistency. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA'14, pages 433–452. ACM, 2014.

[11] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. REWIND: Recovery Write-ahead System for In-Memory Non-volatile Data-Structures. *Proceedings of the VLDB Endowment*, 8:497–508, 2015.

[12] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven 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. ACM, 2011.

[13] Jeremy Condit, Edmund B Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. Better I/O through Byte-addressable, Persistent Memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating systems principles*, SOSP'09, pages 133–146. ACM, 2009.

[14] Andreia Correia, Pascal Felber, and Pedro Ramalhete. Romulus: Efficient Algorithms for Persistent Transactional Memory. In *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures*, SPAA'18, pages 271–282. ACM, 2018.

[15] Dan Williams. libnvdimm for 4.12, 2017. https://lkml.org/lkml/2017/5/5/620.

[16] Dan Williams. libnvdimm for 4.13, 2017. https://lkml.org/lkml/2017/7/6/843.

[17] Dan Williams. use memcpy_mcsafe() for copy_to_iter(), 2018. https://lkml.org/lkml/2018/5/1/708.

[18] Joel E. Denny, Seyong Lee, and Jeffrey S. Vetter. NVL-C: Static Analysis Techniques for Efficient, Correct Programming of Non-Volatile Main Memory Systems. In *Proceedings of the 25th ACM International Symposium on High-Performance Parallel and Distributed Computing*, HPDC'16, pages 125–136. ACM, 2016.

[19] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi 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.

[20] Clang documentation. LLVM-ThreadSanitizer, 2020. https://clang.llvm.org/docs/ThreadSanitizer.html.

[21] Mingkai Dong and Haibo Chen. Soft Updates Made Simple and Fast on Non-volatile Memory. In *Proceedings of the USENIX Annual Technical Conference*, ATC'17, pages 719–731, Santa Clara, CA, 2017. USENIX Association.

[22] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff 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.

[23] Exim Internet Mailer, 2019. http://www.exim.org.

[24] Facebook. RocksDB, 2019. http://rocksdb.org.

[25] 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.

[26] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *Proceedings of the 15th Usenix Conference on File and Storage Technologies*, FAST'17, pages 149–165. USENIX Association, 2017.

[27] Ellis R Giles, Kshitij Doshi, and Peter Varman. SoftWrAP: A Lightweight Framework for Transactional Support of Storage Class Memory. In *Mass Storage Systems and Technologies (MSST), 2015 31st Symposium on*, MSST'16, pages 1–14. IEEE, 2015.

[28] Github. cpp: fix pop.persist function in obj_cpp_ptr, 2017. https://github.com/pmem/pmdk/commit/3d9e7c7cfb2a17e536bf1ff86c17d5505923f1a8.

[29] Github. examples: pi: fix persist range, 2017. https://github.com/pmem/pmdk/commit/e00c034634146ede9b7dd62070b4409cdb44be8f.

[30] Github. examples: fix missing undo logging in rtree, 2018. https://github.com/pmem/pmdk/commit/2cfc70b39588cd5c224ae973a5b50ddcd90e8f0a.

[31] Github. examples: fix use-after-free in hashmap_tx, 2018. https://github.com/pmem/pmdk/commit/9a1a164af821f618af82e355a6f696b9c0027c66.

[32] Github. examples: split a range snapshot in rtree, 2018. https://github.com/pmem/pmdk/commit/78c53e1e70dac9cb3282a4af10c0ceeecc2b7181.

[33] Github. examples: fix persist range in hashmap_rp, 2019. https://github.com/pmem/pmdk/commit/9075946e0d23a2cf8fd110dad4280409eed8bfdf.

[34] Github. fix unflushed in relloc_int, 2019. https://github.com/pmem/pmdk/pull/3860.

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

[36] Qingda Hu, Jinglei Ren, Anirudh Badam, Jiwu Shu, and Thomas Moscibroda. Log-Structured Non-Volatile Main Memory. In *Proceedings of the USENIX Annual Technical Conference*, ATC'17, pages 703–717. USENIX Association, 2017.

[37] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'12, pages 111–122. ACM, 2012.

[38] IBM. Chipkill Memory, 1999. http://www-05.ibm.com/hu/termekismertetok/xseries/dn/chipkill.pdf.

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

[40] Intel. Intel 64 and IA-32 Architectures Software Developer's Manuals, 2017. https://software.intel.com/en-us/articles/intel-sdm.

[41] Intel. Intel Architecture Instruction Set Extensions Programming Reference, 2017. https://software.intel.com/en-us/isa-extensions.

[42] Intel. Introduction to Programming with Persistent Memory from Intel, 2017. https://software.intel.com/en-us/articles/introduction-to-programming-with-persistent-memory-from-intel.

[43] Intel. Optane Memory, 2017. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html.

[44] Intel. Persistent Memory Programming - Frequently Asked Questions, 2017. https://software.intel.com/en-us/articles/persistent-memory-programming-frequently-asked-questions.

[45] Intel. Discover Persistent Memory Programming Errors with Pmemcheck, 2018. https://software.intel.com/en-us/articles/discover-persistent-memory-programming-errors-with-pmemcheck.

[46] Intel. Intelligent Storage Acceleration Library, 2018. https://software.intel.com/en-us/storage/isa-l.

[47] Intel. Intel® Optane™ DC persistent memory, 2019. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[48] Joseph Izraelevitz, Terence Kelly, and Aasheesh Kolli. Failure-Atomic Persistent Memory Updates via JUSTDO Logging. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 427–442. ACM, 2016.

[49] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module, 2019. https://arxiv.org/abs/1903.05714.

[50] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. Shore-MT: A Scalable Storage Manager for the Multicore Era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT'09, pages 24–35, New York, NY, USA, 2009. ACM.

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

[52] Aasheesh Kolli, Steven Pelley, Ali Saidi, Peter M. Chen, and Thomas F. Wenisch. High-Performance Transactions for Persistent Memories. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'16, pages 399–411. ACM, 2016.

[53] Aasheesh Kolli, Jeff Rosen, Stephan Diestelhorst, Ali Saidi, Steven Pelley, Sihang Liu, Peter M Chen, and Thomas F Wenisch. Delegated Persist Ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[54] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith 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. USENIX Association, 2017.

[55] Philip Lantz, Dulloor Subramanya Rao, Sanjay Kumar, Rajesh Sankaran, and Jeff Jackson. Yat: A Validation Framework for Persistent Memory Software. In *Proceedings of the USENIX Annual Technical Conference*, ATC'14, pages 433–438. USENIX Association, 2014.

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

[57] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug 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.

[58] Xiaozhou Li, David G. Andersen, Michael Kaminsky, and Michael J. Freedman. Algorithmic Improvements for Fast Concurrent Cuckoo Hashing. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys'14, pages 27:1–27:14. ACM, 2014.

[59] Mengxing Liu, Mingxing Zhang, Kang Chen, Xuehai Qian, Yongwei Wu, Weimin Zheng, and Jinglei Ren. DudeTM: Building Durable Transactions with Decoupling for Persistent Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'17, pages 329–343. ACM, 2017.

[60] Sihang Liu, Yizhou Wei, Jishen Zhao, Aasheesh Kolli, and Samira Khan. PMTest: A Fast and Flexible Testing Framework for Persistent Memory Programs. In *Proceedings of the Twenty-Forth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'19. ACM, 2019.

[61] llvm.org. Clang Static Analyzer, 2019. https://clang-analyzer.llvm.org.

[62] Tony Luck. Patchwork mm/hwpoison: Clear PRESENT Bit for Kernel 1:1 Mappings of Poison Pages, 2017. https://patchwork.kernel.org/patch/9793701.

[63] LWN. Add Support for NV-DIMMs to Ext4, 2014. https://lwn.net/Articles/613384.

[64] LWN. xfs: DAX support, 2015. https://lwn.net/Articles/635514.

[65] Patrick J Meaney, Luis Alfonso Lastras-Montaño, Vesselina K Papazova, Eldee Stephens, JS Johnson, Luiz C Alves, James A O'Connor, and William J Clarke. IBM zEnterprise Redundant Array of Independent Memory Subsystem. *IBM Journal of Research and Development*, 56(1):43–53, 2012.

[66] Amirsaman Memaripour, Anirudh Badam, Amar Phanishayee, Yanqi Zhou, Ramnatthan Alagappan, Karin Strauss, and Steven 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. ACM, 2017.

[67] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field. In *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 415–426, June 2015.

[68] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur 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. ACM, 2015.

[69] Micron. 3D XPoint Technology, 2017. http://www.micron.com/products/advanced-solutions/3d-xpoint-technology.

[70] Micron. Hybrid Memory: Bridging the Gap Between DRAM Speed and NAND Non-volatility, 2017. http://www.micron.com/products/dram-modules/nvdimm.

[71] MongoDB, Inc. MongoDB, 2019. https://www.mongodb.com.

[72] Iulian Moraru, David G Andersen, Michael Kaminsky, Niraj Tolia, Parthasarathy Ranganathan, and Nathan 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. ACM, 2013.

[73] Sanketh Nalli, Swapnil Haria, Mark D Hill, Michael M Swift, Haris Volos, and Kimberly 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. ACM, 2017.

[74] Dushyanth Narayanan and Orion Hodson. Whole-system Persistence. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'12, pages 401–410. ACM, 2012.

[75] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra 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. ACM, 2016.

[76] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'07, pages 89–100. ACM, 2007.

[77] Matheus Almeida Ogleari, Ethan L Miller, and Jishen Zhao. Steal but No Force: Efficient Hardware Undo+Redo Logging for Persistent Memory Systems. In *High Performance Computer Architecture (HPCA), 2018 IEEE International Symposium on*, HPCA'18, pages 336–349. IEEE, 2018.

[78] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA'14, pages 265–276. IEEE Press, 2014.

[79] Christian Perone and David Murray. pynvm: Non-volatile memory for Python, 2017. https://github.com/pmem/pynvm.

[80] pmem.io. Type safety macros in libpmemobj, 2015. https://pmem.io/2015/06/11/type-safety-macros.html.

[81] pmem.io. PMReorder - Performs a persistent consistency check using a store reordering mechanism, 2018. http://pmem.io/pmdk/manpages/linux/master/pmreorder/pmreorder.1.html.

[82] pmem.io. Persistent Memory Development Kit, 2019. http://pmem.io/pmdk.

[83] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP'05, pages 206–220. ACM, 2005.

[84] S. Raoux, G.W. Burr, M.J. Breitwisch, C.T. Rettner, Y.C. Chen, R.M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H. L Lung, and C.H. Lam. Phase-change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development*, 52(4.5):465–479, July 2008.

[85] Jinglei Ren, Jishen Zhao, Samira Khan, Jongmoo Choi, Yongwei Wu, and Onur Mutiu. ThyNVM: Enabling software-transparent crash consistency in persistent memory systems. In *Proceedings of the 48th International Symposium on Microarchitecture*, MICRO'15, pages 672–685. IEEE, 2015.

[86] Ohad Rodeh, Josef Bacik, and Chris Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, August 2013.

[87] Mendel Rosenblum and John K. Ousterhout. The Design and Implementation of a Log-structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 1–15, New York, NY, USA, 1991. ACM.

[88] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, pages 1–16, Santa Clara, CA, 2014. USENIX.

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

[90] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies*, volume 6, pages 9:1–9:23, New York, NY, USA, September 2010. ACM.

[91] Bianca Schroeder and Garth 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.

[92] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-scale Field Study. In *Proceedings of the Eleventh International Joint Conference*

on Measurement and Modeling of Computer Systems, SIGMETRICS'09, pages 193–204. ACM, 2009.

[93] Jihye Seo, Wook-Hee Kim, Woongki Baek, Beomseok Nam, and Sam 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. ACM, 2017.

[94] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitry Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference*, ATC'12. USENIX Association, 2012.

[95] SNIA. NVM Programming Model v1.2, 2017. https://www.snia.org/tech_activities/standards/curr_standards/npm.

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

[97] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'15, pages 297–310. ACM, 2015.

[98] Dmitri B Strukov, Gregory S Snider, Duncan R Stewart, and R Stanley Williams. The Missing Memristor Found. *Nature*, 453(7191):80–83, 2008.

[99] Vasily Tarasov, Erez Zadok, and Spencer Shepler. Filebench: A flexible framework for file system benchmarking. *USENIX; login*, 41, 2016.

[100] Stephen C. Tweedie. Journaling the Linux ext2fs Filesystem. In *LinuxExpo'98: Proceedings of The 4th Annual Linux Expo*, 1998.

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

[102] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11. USENIX Association, 2011.

[103] Haris Volos, Andres Jaan Tack, and Michael M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS'11, pages 91–104. ACM, 2011.

[104] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the 14th USENIX Conference on File and Storage Technologies*, FAST'16, pages 323–338. USENIX Association, 2016.

[105] Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy 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. ACM, 2017.

[106] Zhongxing Xu, Ted Kremenek, and Jian Zhang. A Memory Model for Static Analysis of C Programs. In *Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation - Volume Part I*, ISoLA'10, pages 535–548, Berlin, Heidelberg, 2010. Springer-Verlag.

[107] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. NV-Tree: Reducing Consistency Cost for NVM-based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies*, FAST'15, pages 167–181. USENIX Association, 2015.

[108] Lu Zhang and Steven Swanson. Pangolin: A Fault-Tolerant Persistent Memory Programming Library. In *Proceedings of the USENIX Annual Technical Conference*, ATC'19, pages 897–912, Renton, WA. USENIX Association.

[109] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi 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. USENIX Association, 2010.

[110] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman 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'13, pages 421–432. ACM, 2013.