NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System

Jian Xu*
University of California, San Diego
jix024@eng.ucsd.edu

Lu Zhang*
University of California, San Diego
luzh@eng.ucsd.edu

Amirsaman Memaripour
University of California, San Diego

Akshatha Gangadharaiah
University of California, San Diego

Amit Borase
University of California, San Diego

Tamires Brito Da Silva
University of California, San Diego

Steven Swanson
University of California, San Diego
swanson@cs.ucsd.edu

Andy Rudoff
Intel Corporation
andy.rudoff@intel.com

ABSTRACT

Emerging fast, persistent memories will enable systems that combine conventional DRAM with large amounts of non-volatile main memory (NVMM) and provide huge increases in storage performance. Fully realizing this potential requires fundamental changes in how system software manages, protects, and provides access to data that resides in NVMM. We address these needs by describing an NVMM-optimized file system called NOVA-Fortis that is both fast and resilient in the face of corruption due to media errors and software bugs. We identify and propose solutions for the unique challenges in adding fault tolerance to an NVMM file system, adapt state-of-the-art reliability techniques to an NVMM file system, and quantify the performance and storage overheads of these techniques. We find that NOVA-Fortis' reliability features consume 14.8% of the storage for redundancy and reduce application-level performance by between 2% and 38% compared to the same file system with the features removed. NOVA-Fortis outperforms DAX-aware file systems without reliability features by 1.5× on average. It outperforms reliable, block-based file systems running on NVMM by 3× on average.

*The first two authors contributed equally to this work.

SOSP '17, October 28, 2017, Shanghai, China

ACM ISBN 978-1-4503-5085-3/17/10.
https://doi.org/10.1145/3132747.3132761

CCS CONCEPTS

• **Information systems** → *Phase change memory*; *Mirroring*; *RAID*; *Point-in-time copies*;

KEYWORDS

Persistent Memory, Non-volatile Memory, Direct Access, DAX, File Systems, Reliability

ACM Reference Format:
Jian Xu, Lu Zhang, Amirsaman Memaripour, Akshatha Gangadharaiah, Amit Borase, Tamires Brito Da Silva, Steven Swanson, and Andy Rudoff. 2017. NOVA-Fortis: A Fault-Tolerant Non-Volatile Main Memory File System. In *Proceedings of ACM SIGOPS 26th Symposium on Operating Systems Principles (SOSP '17).* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3132747.3132761

1 INTRODUCTION

Fast, persistent memory technologies (e.g., battery-backed NVDIMMs [37] or Intel and Micron's 3D XPoint [36]) will enable computer systems with expansive memory systems that combine volatile and non-volatile memories. These hybrid memory systems offer the promise of dramatic increases in storage performance.

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 [12, 15, 64] designed specifically for NVMMs and several Windows and Linux file systems now include at least rudimentary support for them [10, 20, 63]. 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 [64] by incorporating snapshots, replication, checksums, and RAID-4 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.

We make the following contributions:

(1) We identify the unique challenges that the *caveat DAXor* principle presents to building a fault-tolerant NVMM file systems.
(2) We 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). Then, we describe NOVA-Fortis' snapshot and (meta)data protection mechanisms (Sections 3 and 4). Section 5 evaluates these mechanisms, and Section 6 presents our conclusions.

2 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 Non-volatile Memory Technologies

Modern server platforms have support NVMM in form of NVDIMMs [22, 37] 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) [32, 44], resistive RAM (ReRAM) [18, 56], and 3D XPoint memory technology [36] 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) [28] are faster, but less dense and may find other roles in future systems (e.g., as non-volatile caches [67]). These technologies are all under active development and knowledge about their reliability and performance is evolving rapidly.

Commercial availability of these technologies appears to be close at hand. The 3D XPoint memory technology has already appeared in SSDs [25], and is expected to appear on the processor memory bus shortly [26]. 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 [2, 6]. Without these guarantees it is impossible to build data structures in NVMM that are reliable in the face of power failures [11, 60, 61].

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 [12, 30, 41].

For example, under x86 a persist barrier comprises a `clflush` or `clwb` [24] 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.

2.2 NVMM File Systems and DAX

Several groups have designed NVMM file systems [12, 14, 15, 63, 64] 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 2.4.

NVMMs' low latencies make software efficiency much more important than in block-based storage devices [7, 9, 62, 65].

NVMM-aware CPUs provide a load/store interface with atomic 8-byte [1] 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 [20] and Linux [10, 63] 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 [11, 42, 61] 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 [11].

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

[1] NVMM-aware Intel CPUs provide 8-byte atomic operations. Other architectures may provide different atomicity semantics.

complex data structures, but they are more difficult to use correctly.

2.3 File System Consistency and Reliability

Apart from their core function of storing and retrieving data, file systems also provide facilities to protect the data they hold from corruption due to system failures, media errors, and software bugs (both in the file system and elsewhere).

File systems have devised a variety of different techniques to guarantee system-wide consistency of file system data structures, including journaling [15, 58], copy-on-write [8, 12, 45] and log-structuring [46, 47].

Highly-reliable file systems like ZFS [8] and Btrfs [45] 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.

2.3.1 Snapshots. Snapshots provide a consistent image of the file system at a moment in time. Their most important application is facilitating consistent backups without unmounting the file system, affording protection against catastrophic system failures and the accidental deletion or modification of files.

Many modern file systems have built-in support for snapshots [8, 29, 45]. In other systems the underlying storage stack (e.g., LVM in Linux) can take snapshots of the underlying block device.

Neither existing DAX-enabled NVMM file systems nor current low-level NVMM drivers support snapshots, making consistent online backups impossible.

2.3.2 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 [4, 49, 50], SSDs [34, 40, 51] and DRAM-based memories [52, 54, 55]. 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 4.1).

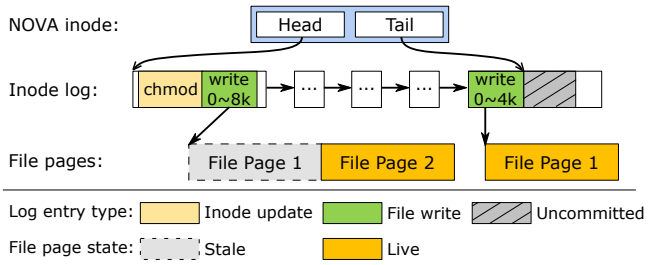

Figure 1: NOVA inode log structure – A NOVA inode log records changes to the inode (e.g., the mode change and two file write operations shown above). NOVA stores file data outside the log.

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" [31] 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 4.8.

2.4 The NOVA File System

NOVA-Fortis is based on the NOVA NVMM file system [64]. 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 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 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 [64].

3 SNAPSHOTS

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

Below, we described the three central challenges that NOVA-Fortis' snapshot mechanisms address: Taking efficient snapshots, managing storage for snapshots, and taking usable snapshots of DAX `mmap()`'d files.

3.1 Taking and Restoring Snapshots

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

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

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

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

3.2 Snapshot Management

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

NOVA-Fortis maintains a *snapshot manifest* for each snapshot. Entries in the manifest contain a pointer to a log entry, a *birth snapshot ID*, and a *death snapshot ID*. We denote a manifest entry with [*create snapshot ID, delete snapshot ID*). The manifest for a snapshot ID contains entries for all the log entries that were *born* (i.e., created) with that snapshot ID and have since died (i.e., become stale because another write replaced them) in a *different* snapshot. If a snapshot is deleted, the manifest entries for that snapshot become part of the following snapshot. This can result in a snapshot containing multiple manifest entries for the same file data. In this case, it is safe to remove the older entry (i.e., those with earlier birth snapshot IDs).

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

The write to location 0 in (b) kills the write to that location in (a). NOVA-Fortis checks whether that log entry was born and died in different snapshots. In the example, it was born in snapshot 0 and died in snapshot 1, so NOVA-Fortis adds a entry to the snapshot 0 manifest (Step 3) and preserves the log entry. In Figure 2 (c), the application issues two writes that overwrite entries in snapshots 0 and 1, so NOVA-Fortis adds entries to those manifests.

In Figure 2 (d), NOVA-Fortis deletes snapshot 0 by removing it from the list of live snapshots and adding the manifest entries for snapshot 0 to snapshot 1's manifest (Step 4). In effect, this moves the start of snapshot 1 to the beginning of snapshot 0.

Then, NOVA-Fortis starts a background thread that compacts the combined manifest by discarding all the manifest entries that cover one part of the file except the most recent

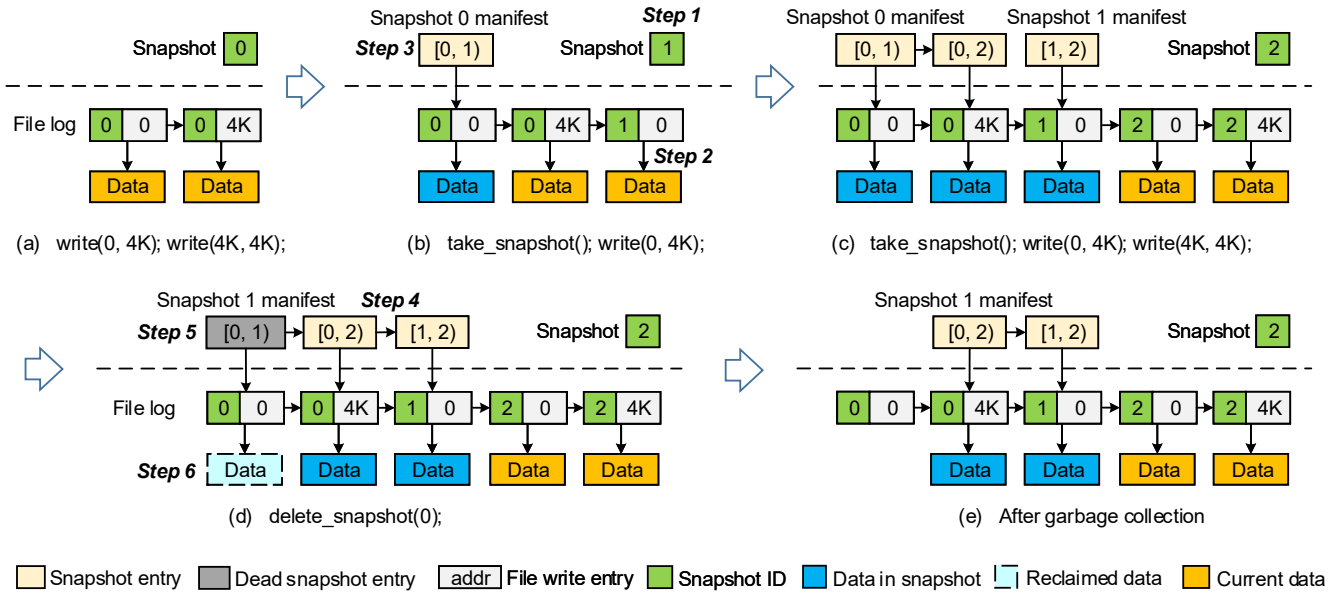

Figure 2: Snapshots in NOVA-Fortis – NOVA-Fortis stores the current snapshot ID in each log entry, and takes a snapshot by incrementing the snapshot ID. When a data block or log entry becomes dead in the current snapshot, NOVA-Fortis records its death in the snapshot manifest for most recent snapshot it survives in. To delete a snapshot, NOVA-Fortis adds the manifest for the deleted snapshot to the manifest for the next snapshot and removes redundant entries.

(Step 5). As it discards the manifest entries, it marks the corresponding log entries as dead and reclaims the associated file data (Step 6). Finally, (e) shows the file system state after manifest compaction is complete.

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

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

Taking consistent snapshots while applications are modifying files using DAX-style `mmap` requires NOVA-Fortis to reckon with the order in which stores to NVMM become persistent (i.e., reach physical NVMM so they will survive a system failure). These applications rely on the processor's "memory persistence model" [41] to make guarantees about when and in what order stores become persistent, allowing them to restore their data to a consistent state when the application restarts after a system failure.

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

A naive approach to checkpointing `mmap()`'d files in NOVA-Fortis would simply mark each of the read/write mapped pages as read-only and then do copy-on-write when a store occurs to preserve the old pages as part of the snapshot.

However, this approach can leave the snapshot in an inconsistent state: Setting the page to read-only captures its contents for the snapshot, and the kernel requires NOVA-Fortis to set the pages as read-only one at a time. If the order in which NOVA-Fortis marks pages as read-only is incompatible with ordering that the application requires, the snapshot will contain an inconsistent version of the file.

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

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

Figure 3 illustrates how we resolve the problem: When NOVA-Fortis starts marking pages as read-only in (a), it

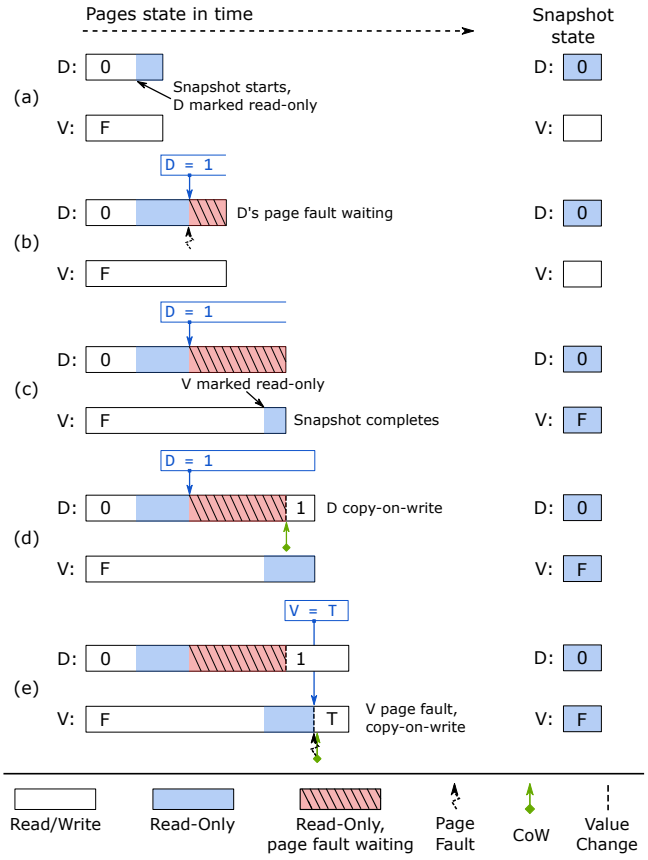

Figure 3: Snapshots of `mmap()`'d data – NOVA-Fortis marks `mmap()`'d pages as read-only to capture their contents in the snapshot. To preserve the application's constraints on when stores become persistent (here, that the assignment to V follow the assignment to D), it prevents pages faults to read-only pages from completing until it can mark all the pages read-only.

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

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

This approach guarantees that, for each thread, the snapshot will reflect a prefix of the program order-based sequence of NVMM store operations the thread has performed. This model is equivalent to strict persistence [41], the most restrictive model for how NVMM memory operations should behave (i.e., in what order updates can become persistent) in a multi-processor system. CPUs may implement more aggressive, relaxed models for memory persistence, but strict persistence is strictly more conservative than all proposed models, so NOVA-Fortis' approach is correct under those models as well [2].

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

Table 1: Snapshot manifest size – The size of snapshot manifest is proportional to the size of changes between snapshots.

3.4 Design Decisions and Evaluation

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

In practice, the size of the manifests is manageable. We ran the fileserver workloads described in Section 5 and the WHISPER [39] workloads while taking snapshots every 10 seconds. WHISPER is a suite of eight applications that use DAX `mmap()` for data access. Table 1 shows the results. The size of the manifests ranged from 0.027% to 0.05% of the space in use in the file system.

Taking snapshots has different effects on the performance of applications that use file-based access and those that use DAX-style `mmap()`. Figure 4 measures the impact on both

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

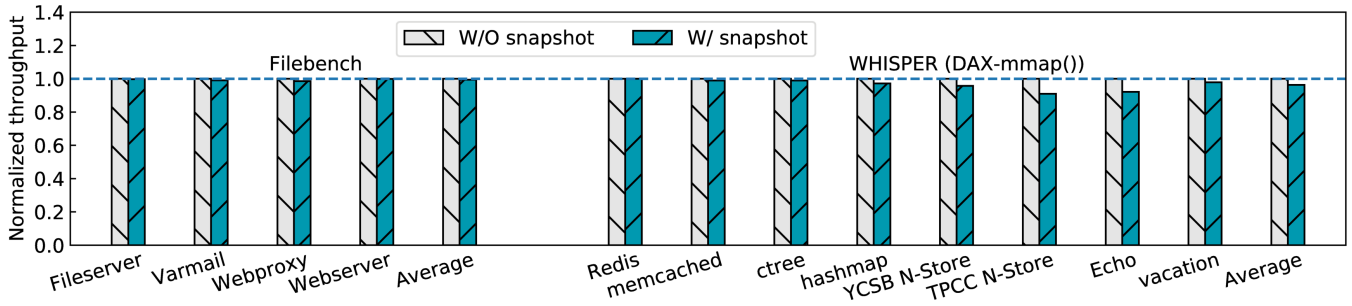

Figure 4: Snapshot performance impact – For file-based application ("Filebench"), taking a snapshot every 10 s reduces performance by just 0.6% on average. WHISPER applications that use DAX-`mmap()` see a larger drop: 3.7% on average.

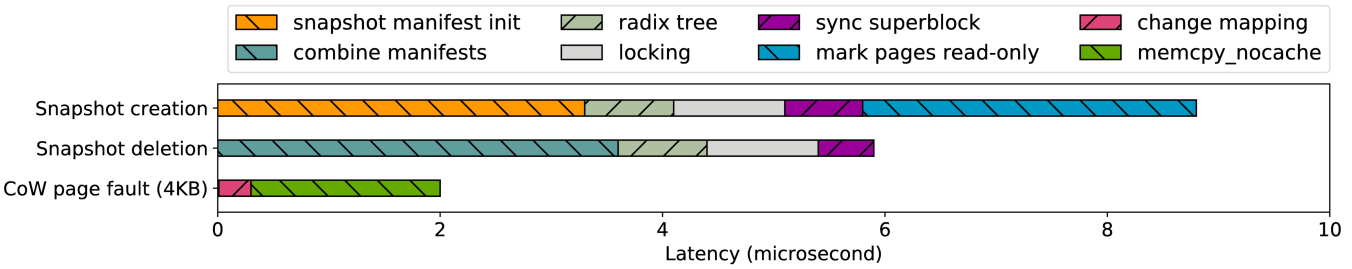

Figure 5: Snapshot operation latency – The time required to mark pages read-only during snapshot creation is proportional to the number of write-faulted pages (300 in this example) in the DAX `mmap()`'d regions.

groups. On the right, it shows results for the WHISPER applications. On the left are results for four Filebench [57] workloads.

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

Figure 5 shows the latency of snapshot operations. For file-based applications, snapshot creation takes constant time. For DAX `mmap()` applications, the time to mark `mmap()`'d regions as read-only is proportional to the number of write-faulted pages, and each page takes about 10 ns. Snapshot deletion always takes constant time, since it only needs to combine the deleted snapshot's manifest with the next snapshot's manifest and then perform stale data reclamation in the background. The background thread takes about 400 ns to reclaim 4 KB data. A copy-on-write page fault takes 2 μs: 1.8 μs for copying data to new page and 200 ns to update the page table.

4 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.

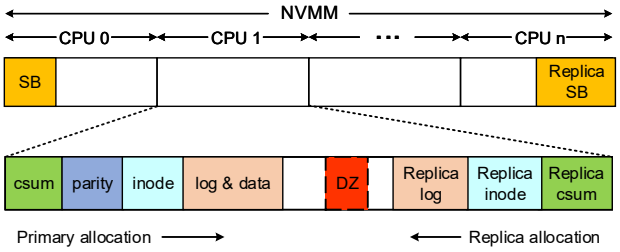

Figure 6: NOVA-Fortis space layout – NOVA-Fortis' per-core allocators satisfy requests for primary and replica storage from different directions. They also store data pages and their checksum and parity pages separately. The "dead zone" (DZ) for metadata allocation guarantees a gap between primary and replica pages.

4.1 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) [23] 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 [59].

4.2 Tick-Tock Metadata Protection

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

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

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

4.3 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 6 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) and scribbles. This is an interesting challenge but beyond the scope of this paper. 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 [43] take, but we store parity for each page rather than one parity page per file [43], 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 [33], Chipkill [21], 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.

4.4 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 [19, 31, 66]. 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 7 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

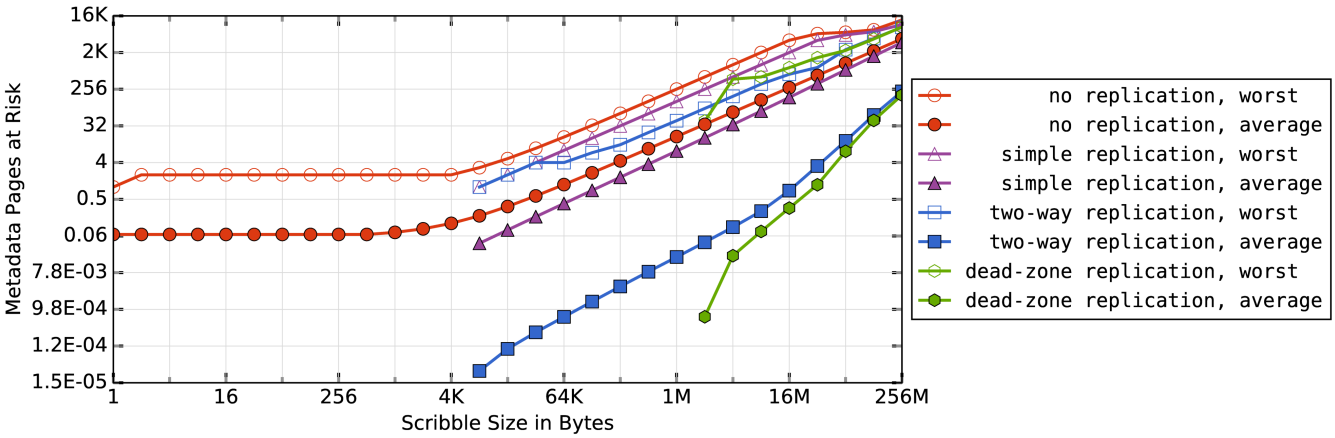

Figure 7: Scribble size and metadata bytes at risk – Replicating metadata pages and taking care to allocate the replicas separately improves resilience to scribbles. The most effective technique enforces a 1 MB "dead zone" between replicas and eliminates the threat of a single scribble smaller than 1 MB. The graph omits zero values due to the vertical log scale.

pages. Figure 6 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.

4.5 Preventing Scribbles

The mechanisms described above let NOVA-Fortis detect and recover from data corruption. NOVA-Fortis can borrow a technique from WAFL [31] and PMFS [15] 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 ~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 [5].

4.6 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 5.

4.7 Protecting DRAM Data Structures

Corruption of DRAM data structures can result in file system corruption [13, 66], and NOVA-Fortis protects most of its critical DRAM data structures with checksums. Most DRAM structures that NOVA-Fortis does not protect are short lived (e.g., the DRAM copies we create of metadata structures) or are not written back to NVMM. However, the snapshot logs and allocator state are exceptions and they are 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.

4.8 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.

4.8.1 Is NOVA-Fortis Ferrous? InteRnally cONistent (IRON) file systems [43] 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.

4.8.2 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 [3], and errors in NVMM are also likely to exhibit complex patterns of locality [35, 54, 55]. 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 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.

5 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.

5.1 Experimental Setup

We implemented NOVA-Fortis for Linux 4.10, and the source code is available at https://github.com/NVSL/linux-nova.

We use the Intel Persistent Memory Emulation Platform (PMEP) [15] 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 [48], 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

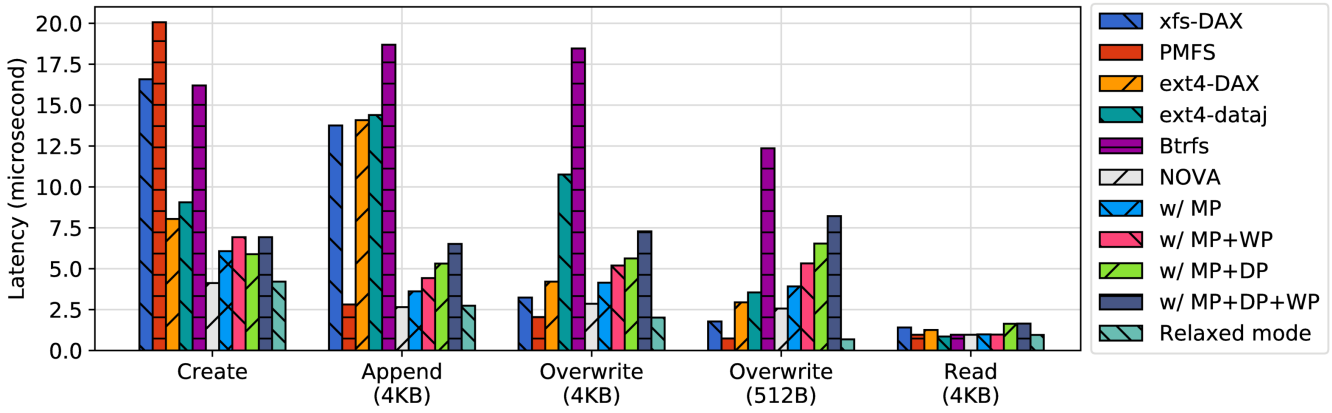

Figure 8: File operation latency – NOVA-Fortis' basic file operations are faster than competing file systems except in cases where the other file system provides weaker consistency guarantees and/or data protection (e.g., writes and overwrites vs. Ext4-DAX and xfs-DAX). Sacrificing these protections in NOVA-Fortis ("Relaxed mode") improves performance.

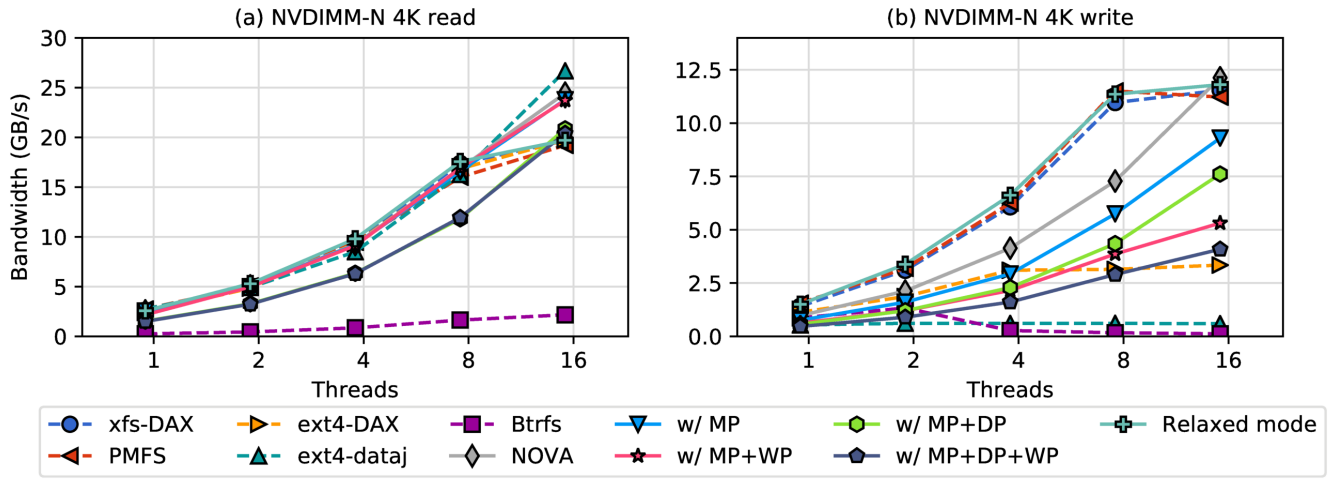

Figure 9: NOVA-Fortis random read/write bandwidth on NVDIMM-N – Read bandwidth is similar across all the file systems except Btrfs, and NOVA-Fortis' reliability mechanisms reduces its throughput by between 14% and 19%. For writes the cost of reliability is higher – up to 66%, but still outperforms Btrfs by 33× with 16 threads.

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.

5.2 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 4.6).

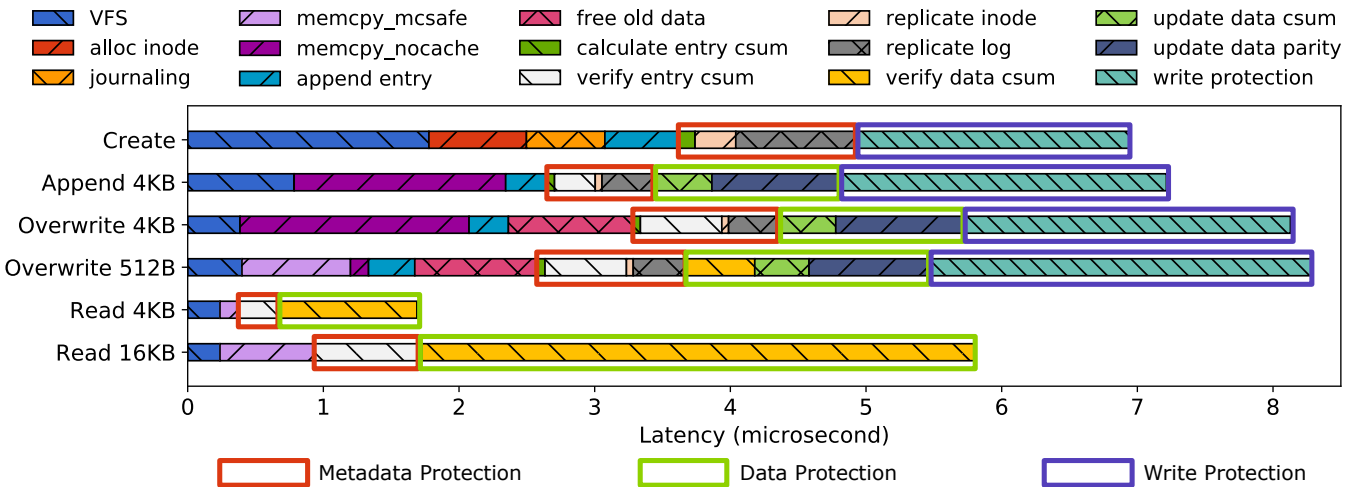

Figure 10: NOVA-Fortis latencies for NVDIMM-N – Protecting file data is usually more expensive than protecting metadata because the cost of computing checksums and parity for data scales with access size.

5.3 Microbenchmarks

We evaluate basic file system operations: `create`, 4 KB append, 4 KB `write`, 512 B `write`, and 4 KB `read`. Figure 8 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 verifies the data checksum before returning to the user.

Figure 10 breaks down the latency for NOVA-Fortis and its reliability mechanisms. For `create`, inode allocation and appending to the log combine to consume 48% of latency, 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 9 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.

5.4 Macrobenchmarks

We use nine application-level workloads to evaluate NOVA-Fortis: Four Filebench [57] workloads (fileserver, varmail, webproxy, and webserver), two key-value stores (RocksDB [17] and MongoDB [38]), the Exim email server [16], SQLite [53], and TPC-C running on Shore-MT [27]. Fileserver, varmail, webproxy and Exim are metadata-intensive workloads, while other workloads are data-intensive. Table 2 summarizes the workloads.

Figure 11 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

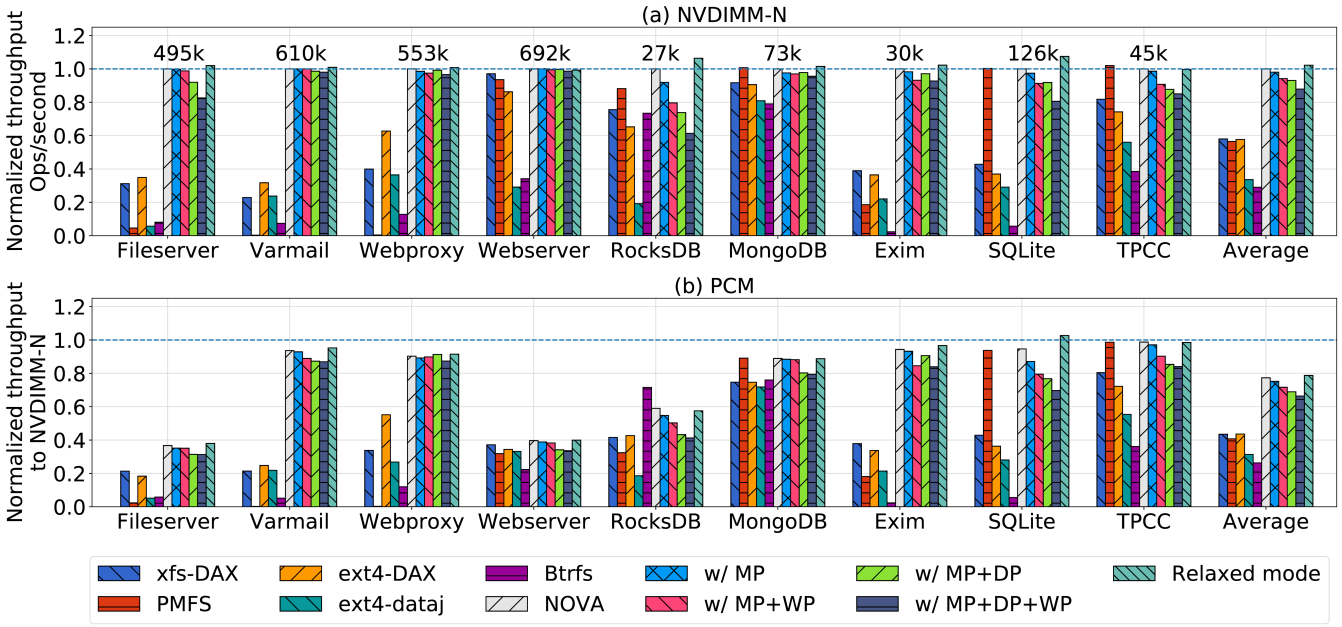

Figure 11: Application performance on NOVA-Fortis – Reliability overheads and the benefits of relaxed mode have less impact on applications than microbenchmarks (Figures 8 and 9). The change is especially small for read-intensive workloads (e.g., web-proxy), while databases and key-value stores see greater differences. The numbers above the bars measure NOVA throughput in operations per second.

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

Table 2: Application benchmarks

enabled outperforms it by between 26% and 42×. NOVA-Fortis achieves larger improvement on metadata-intensive workloads, such as varmail and Exim.

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

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

For the PCM configuration, fileserver, webserver and RocksDB show the largest performance drop compared to NVDIMM-N. Fileserver and RocksDB are write-intensive and saturate PCM's write bandwidth. Webserver is read-intensive and PCM's read latency limits performance. Btrfs outperforms other DAX file systems on 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.

5.5 NVMM Storage Utilization

Protecting data integrity introduces storage overheads. Figure 12 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

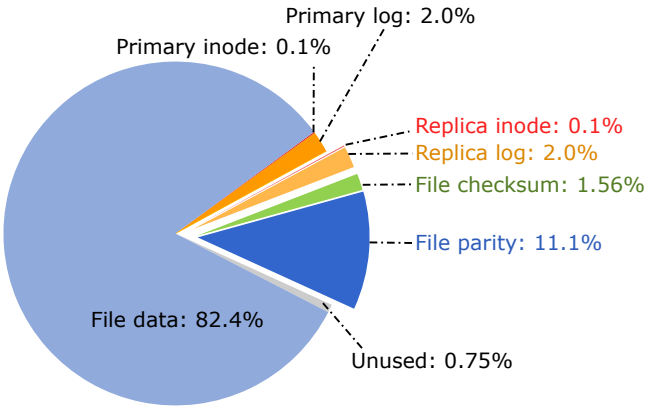

Figure 12: NVMM storage utilization – Extra storage required for reliability is highlighted to the right. Protecting data is more expensive that protecting metadata, consuming 12.7% of storage compared to just 2.1% for metadata.

reliability. Of this, metadata redundancy accounts for 2.1% and data redundancy occupies 12.7%.

6 CONCLUSION

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

The extra storage required to implement these changes is modest, but their performance impact is significant for some applications. In particular, the cost of checking and maintaining checksums and parity for file data incurs a steep cost for both reads and writes, 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 work was supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA and by Intel Corporation, Toshiba Corporation, and NSF Award 1629395. We would like to thank our shepherd, Michael Swift, and the anonymous SOSP reviewers for their insightful comments and suggestions. We are also thankful to Subramanya R. Dulloor from Intel for his support and help with accessing PMEP.

REFERENCES

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

[2] Katelin Bailey, Luis Ceze, Steven D. Gribble, and Henry M. Levy. 2011. Operating System Implications of Fast, Cheap, Non-volatile Memory. In *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems (HotOS'13)*. USENIX Association, Berkeley, CA, USA, 2–2. http://dl.acm.org/citation.cfm?id=1991596.1991599

[3] Lakshmi N. Bairavasundaram, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, Garth R. Goodson, and Bianca Schroeder. 2008. An Analysis of Data Corruption in the Storage Stack. *Trans. Storage* 4, 3, Article 8 (Nov. 2008), 28 pages. https://doi.org/10.1145/1416944.1416947

[4] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. 2007. 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)*. ACM, New York, NY, USA, 289–300. https://doi.org/10.1145/1254882.1254917

[5] L. N. Bairavasundaram, M. Rungta, N. Agrawa, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. M. Swift. 2008. Analyzing the Effects of Disk-Pointer Corruption. In *2008 IEEE International Conference on Dependable Systems and Networks With FTCS and DCC (DSN)*. 502–511. https://doi.org/10.1109/DSN.2008.4630121

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

[7] Meenakshi Sundaram Bhaskaran, Jian Xu, and Steven Swanson. 2013. 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). ACM, New York, NY, USA, Article 1, 9 pages. https://doi.org/10.1145/2527792.2527793

[8] Jeff Bonwick and Bill Moore. 2007. 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. 2012. 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)*. ACM, New York, NY, USA, 387–400. https://doi.org/10.1145/2150976.2151017

[10] Dave Chinner. 2015. xfs: updates for 4.2-rc1. (2015). http://oss.sgi.com/archives/xfs/2015-06/msg00478.html.

[11] Joel Coburn, Adrian M. Caulfield, Ameen Akel, Laura M. Grupp, Rajesh K. Gupta, Ranjit Jhala, and Steven Swanson. 2011. 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)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/1950365.1950380

[12] Jeremy Condit, Edmund B. Nightingale, Christopher Frost, Engin Ipek, Benjamin Lee, Doug Burger, and Derrick Coetzee. 2009. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*. ACM, New York, NY, USA, 133–146. https://doi.org/10.1145/1629575.1629589

[13] Thanh Do, Tyler Harter, Yingchao Liu, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2013. HARDFS: Hardening HDFS with Selective and Lightweight Versioning. In *Presented as part of the 11th USENIX Conference on File and Storage Technologies (FAST 13)*. USENIX, San Jose, CA, 105–118. https://www.usenix.org/conference/fast13/technical-sessions/presentation/do

[14] Mingkai Dong and Haibo Chen. 2017. Soft Updates Made Simple and Fast on Non-volatile Memory. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 719–731. https://www.usenix.org/conference/atc17/technical-sessions/presentation/dong

[15] Subramanya R. Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System Software for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 15, 15 pages. https://doi.org/10.1145/2592798.2592814

[16] Exim 2017. Exim Internet Mailer. (2017). http://www.exim.org.

[17] Facebook. 2017. RocksDB. (2017). http://rocksdb.org.

[18] R. Fackenthal, M. Kitagawa, W. Otsuka, K. Prall, D. Mills, K. Tsutsui, J. Javanifard, K. Tedrow, T. Tsushima, Y. Shibahara, and G. Hush. 2014. 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*. 338–339. https://doi.org/10.1109/ISSCC.2014.6757460

[19] Aishwarya Ganesan, Ramnatthan Alagappan, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2017. Redundancy Does Not Imply Fault Tolerance: Analysis of Distributed Storage Reactions to Single Errors and Corruptions. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 149–166.

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

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

[22] Intel. 2015. NVDIMM Namespace Specification. (2015). http://pmem.io/documents/NVDIMM_Namespace_Spec.pdf.

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

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

[25] Intel. 2017. Intel Optane Memory. (2017). http://www.intel.com/content/www/us/en/architecture-and-technology/optane-memory.html.

[26] Intel. 2017. Intel ships first Optane memory modules for testing. (2017). http://www.pcworld.com/article/3162177/storage/intel-ships-first-optane-memory-modules-for-testing.html.

[27] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. 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)*. ACM, New York, NY, USA, 24–35. https://doi.org/10.1145/1516360.1516365

[28] Takayuki Kawahara. 2011. Scalable Spin-Transfer Torque RAM Technology for Normally-Off Computing. *Design & Test of Computers, IEEE* 28, 1 (Jan 2011), 52–63. https://doi.org/10.1109/MDT.2010.97

[29] Ram Kesavan, Rohit Singh, Travis Grusecki, and Yuvraj Patel. 2017. Algorithms and Data Structures for Efficient Free Space Reclamation in WAFL. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association.

[30] A. Kolli, J. Rosen, S. Diestelhorst, A. Saidi, S. Pelley, S. Liu, P. M. Chen, and T. F. Wenisch. 2016. Delegated Persist Ordering. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–13. https://doi.org/10.1109/MICRO.2016.7783761

[31] Harendra Kumar, Yuvraj Patel, Ram Kesavan, and Sumith Makam. 2017. High Performance Metadata Integrity Protection in the WAFL Copy-on-Write File System. In *15th USENIX Conference on File and Storage Technologies (FAST 17)*. USENIX Association, Santa Clara, CA, 197–212. https://www.usenix.org/conference/fast17/technical-sessions/presentation/kumar

[32] Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*. ACM, New York, NY, USA, 2–13. https://doi.org/10.1145/1555754.1555758

[33] P. J. Meaney, L. A. Lastras-Montanõ, V. K. Papazova, E. Stephens, J. S. Johnson, L. C. Alves, J. A. O'Connor, and W. J. Clarke. 2012. IBM zEnterprise Redundant Array of Independent Memory Subsystem. *IBM J. Res. Dev.* 56, 1 (Jan. 2012),

43–53. https://doi.org/10.1147/JRD.2011.2177106

[34] Justin Meza, Qiang Wu, Sanjev Kumar, and Onur Mutlu. 2015. 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)*. ACM, New York, NY, USA, 177–190. https://doi.org/10.1145/2745844.2745848

[35] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. 2015. 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*. 415–426. https://doi.org/10.1109/DSN.2015.57

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

[37] Micron. 2017. Hybrid Memory: Bridging the Gap Between DRAM Speed and NAND Nonvolatility. (2017). http://www.micron.com/products/dram-modules/nvdimm.

[38] MongoDB, Inc. 2017. MongoDB. (2017). https://www.mongodb.com.

[39] Sanketh Nalli, Swapnil Haria, Mark D. Hill, Michael M. Swift, Haris Volos, and Kimberly Keeton. 2017. 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)*. ACM, New York, NY, USA, 135–148. https://doi.org/10.1145/3037697.3037730

[40] Iyswarya Narayanan, Di Wang, Myeongjae Jeon, Bikash Sharma, Laura Caulfield, Anand Sivasubramaniam, Ben Cutler, Jie Liu, Badriddine Khessib, and Kushagra Vaid. 2016. SSD Failures in Datacenters: What? When? And Why?. In *Proceedings of the 9th ACM International on Systems and Storage Conference (SYSTOR '16)*. ACM, New York, NY, USA, Article 7, 11 pages. https://doi.org/10.1145/2928275.2928278

[41] Steven Pelley, Peter M. Chen, and Thomas F. Wenisch. 2014. Memory Persistency. In *Proceeding of the 41st Annual International Symposium on Computer Architecture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 265–276. http://dl.acm.org/citation.cfm?id=2665671.2665712

[42] pmem.io. 2017. NVM Library. (2017). http://pmem.io/nvml.

[43] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2005. IRON File Systems. In *The ACM Symposium on Operating Systems Principles (SOSP)*. ACM.

[44] 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. 2008. Phase-change Random Access Memory: A Scalable Technology. *IBM Journal of Research and Development* 52, 4.5 (July 2008), 465–479. https://doi.org/10.1147/rd.524.0465

[45] Ohad Rodeh, Josef Bacik, and Chris Mason. 2013. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage* 9, 3, Article 9 (Aug. 2013), 32 pages. https://doi.org/10.1145/2501620.2501623

[46] Mendel Rosenblum and John K. Ousterhout. 1991. The Design and Implementation of a Log-structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles (SOSP '91)*. ACM, New York, NY, USA, 1–15. https://doi.org/10.1145/121132.121137

[47] Stephen M. Rumble, Ankita Kejriwal, and John Ousterhout. 2014. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14)*. USENIX, Santa Clara, CA, 1–16. https://www.usenix.org/conference/fast14/technical-sessions/presentation/rumble

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

[49] Bianca Schroeder, Sotirios Damouras, and Phillipa Gill. 2010. Understanding Latent Sector Errors and How to Protect Against Them. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. USENIX Association, Berkeley, CA, USA, 6–6. http://dl.acm.org/citation.cfm?id=1855511.1855517

[50] Bianca Schroeder and Garth A Gibson. 2007. 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)*.

[51] Bianca Schroeder, Raghav Lagisetty, and Arif Merchant. 2016. Flash Reliability in Production: The Expected and the Unexpected. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 67–80. http://usenix.org/conference/fast16/technical-sessions/presentation/schroeder

[52] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. 2009. DRAM Errors in the Wild: A Large-scale Field Study. In *ACM SIGMETRICS*.

[53] SQLite. 2017. SQLite. (2017). https://www.sqlite.org.

[54] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. 2015. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM.

[55] V. Sridharan and D. Liberty. 2012. A study of DRAM failures in the field. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*. 1–11. https://doi.org/10.1109/SC.2012.13

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

[57] Vasily Tarasov, Erez Zadok, and Spencer Shepler. 2016. Filebench: A Flexible Framework for File System Benchmarking. *USENIX; login* 41 (2016).

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

[59] UEFI Forum. 2017. Advanced Configuration and Power Interface Specification. (2017). http://www.uefi.org/sites/default/files/resources/ACPI_6_2.pdf.

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

USA, 5–5.

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

[62] Dejan Vučinić, Qingbo Wang, Cyril Guyot, Robert Mateescu, Filip Blagojević, Luiz Franca-Neto, Damien Le Moal, Trevor Bunker, Jian Xu, Steven Swanson, and Zvonimir Bandić. 2014. DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST '14)*. USENIX, Santa Clara, CA, 309–315. https://www.usenix.org/conference/fast14/technical-sessions/presentation/vucinic

[63] Matthew Wilcox. 2014. Add Support for NV-DIMMs to Ext4. (2014). https://lwn.net/Articles/613384/.

[64] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *14th USENIX Conference on File and Storage Technologies (FAST 16)*. USENIX Association, Santa Clara, CA, 323–338. https://www.usenix.org/conference/fast16/technical-sessions/presentation/xu

[65] Jisoo Yang, Dave B. Minturn, and Frank Hady. 2012. When Poll Is Better than Interrupt. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12)*. USENIX, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=2208461.2208464

[66] Yupu Zhang, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2010. 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, Berkeley, CA, USA, 3–3. http://dl.acm.org/citation.cfm?id=1855511.1855514

[67] Jishen Zhao, Sheng Li, Doe Hyun Yoon, Yuan Xie, and Norman P. Jouppi. 2013. Kiln: Closing the Performance Gap Between Systems With and Without Persistence Support. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 421–432. https://doi.org/10.1145/2540708.2540744