NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories

Jian Xu Steven Swanson
University of California, San Diego

Abstract

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

We present NOVA, a file system designed to maximize performance on hybrid memory systems while providing strong consistency guarantees. NOVA adapts conventional log-structured file system techniques to exploit the fast random access that NVMs provide. In particular, it maintains separate logs for each inode to improve concurrency, and stores file data outside the log to minimize log size and reduce garbage collection costs. NOVA's logs provide metadata, data, and mmap atomicity and focus on simplicity and reliability, keeping complex metadata structures in DRAM to accelerate lookup operations. Experimental results show that in write-intensive workloads, NOVA provides 22% to 216× throughput improvement compared to state-of-the-art file systems, and 3.1× to 13.5× improvement compared to file systems that provide equally strong data consistency guarantees.

1. Introduction

Emerging non-volatile memory (NVM) technologies such as spin-torque transfer, phase change, resistive memories [2, 28, 52] and Intel and Micron's 3D XPoint [1] technology promise to revolutionize I/O performance. Researchers have proposed several approaches to integrating NVMs into computer systems [11, 13, 19, 31, 36, 41, 58, 67], and the most exciting proposals place NVMs on the processor's memory bus alongside conventional DRAM, leading to hybrid volatile/non-volatile main memory systems [4, 51, 72, 78]. Combining faster, volatile DRAM with slightly slower, denser non-volatile main memories (NVMMs) offers the possibility of storage systems that combine the best characteristics of both technologies.

Hybrid DRAM/NVMM storage systems present a host of opportunities and challenges for system designers. These systems need to minimize software overhead if they are to fully exploit NVMM's high performance and efficiently support more flexible access patterns, and at the same time they must provide the strong consistency guarantees that applications require and respect the limitations of emerging memories (e.g., limited program cycles).

Conventional file systems are not suitable for hybrid memory systems because they are built for the performance characteristics of disks (spinning or solid state) and rely on disks' consistency guarantees (e.g., that sector updates are atomic) for correctness [47]. Hybrid memory systems differ from conventional storage systems on both counts: NVMMs provide vastly improved performance over disks while DRAM provides even better performance, albeit without persistence. And memory provides different consistency guarantees (e.g., 64-bit atomic stores) from disks.

Providing strong consistency guarantees is particularly challenging for memory-based file systems because maintaining data consistency in NVMM can be costly. Modern CPU and memory systems may reorder stores to memory to improve performance, breaking consistency in case of system failure. To compensate, the file system needs to explicitly flush data from the CPU's caches to enforce orderings, adding significant overhead and squandering the improved performance that NVMM can provide [6, 76].

Overcoming these problems is critical since many applications rely on atomic file system operations to ensure their own correctness. Existing mainstream file systems use journaling, shadow paging, or log-structuring techniques to provide atomicity. However, journaling wastes bandwidth by doubling the number of writes to the storage device, and shadow paging file systems require a cascade of updates from the affected leaf nodes to the root. Implementing either technique imposes strict ordering requirements that reduce performance.

Log-structured file systems (LFSs) [55] group small random write requests into a larger sequential write that hard disks and NAND flash-based solid state drives (SSDs) can process efficiently. However, conventional LFSs rely on the availability of contiguous free regions, and maintaining those regions requires expensive garbage collection operations. As a result, recent research [59] shows that LFSs perform worse than journaling file systems on NVMM.

To overcome all these limitations, we present the *NOn-Volatile memory Accelerated (NOVA)* log-structured file system. NOVA adapts conventional log-structured file system techniques to exploit the fast random access provided by hybrid memory systems. This allows NOVA to support massive concurrency, reduce log size, and minimize garbage collection costs while providing strong consistency guarantees for conventional file operations and mmap-based load/store accesses.

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

NOVA does not log data, so the recovery process only needs to scan a small fraction of the NVMM. This also allows NOVA to immediately reclaim pages when they become stale, significantly reducing garbage collection overhead and allowing NOVA to sustain good performance even when the file system is nearly full.

In describing NOVA, this paper makes the following contributions:

- It extends existing log-structured file system techniques to exploit the characteristics of hybrid memory systems.
- It describes *atomic mmap*, a simplified interface for exposing NVMM directly to applications with a strong consistency guarantee.
- It demonstrates that NOVA outperforms existing journaling, shadow paging, and log-structured file systems running on hybrid memory systems.
- It shows that NOVA provides these benefits across a range of proposed NVMM technologies.

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

The remainder of the paper is organized as follows. Section 2 describes NVMMs, the challenges they present, and related work on NVMM file system design. Section 3 gives a overview of NOVA architecture and Section 4 describes the implementation in detail. Section 5 evaluates NOVA, and Section 6 concludes.

2. Background

NOVA targets memory systems that include emerging non-volatile memory technologies along with DRAM. This section first provides a brief survey of NVM technologies and the opportunities and challenges they present to system designers. Then, we discuss how other file systems have provided atomic operations and consistency guarantees. Finally, we discuss previous work on NVMM file systems.

2.1. Non-volatile memory technologies

Emerging non-volatile memory technologies, such as spin-torque transfer RAM (STT-RAM) [28, 42], phase change memory (PCM) [10, 18, 29, 52], resistive RAM (ReRAM) [22, 62], and 3D XPoint memory technology [1], promise to provide fast, non-volatile, byte-addressable memories. Suzuki *et al.* [63] provides a survey of these technologies and their evolution over time.

These memories have different strengths and weaknesses that make them useful in different parts of the memory hierarchy. STT-RAM can meet or surpass DRAM's latency and it may eventually appear in on-chip, last-level caches [77], but its large cell size limits capacity and its feasibility as a DRAM replacement. PCM and ReRAM are denser than DRAM, and may enable very large, non-volatile main memories. However, their relatively long latencies make it unlikely that they will fully replace DRAM as main memory. The 3D XPoint memory technology recently announced by Intel and Micron is rumored to be one of these and to offer performance up to 1,000 times faster than NAND flash [1]. It will appear in both SSDs and on the processor memory bus. As a result, we expect to see hybrid volatile/non-volatile memory hierarchies become common in large systems.

2.2. Challenges for NVMM software

NVMM technologies present several challenges to file system designers. The most critical of these focus on balancing the memories' performance against software overheads, enforcing ordering among updates to ensure consistency, and providing atomic updates.

Performance The low latencies of NVMMs alters the trade-offs between hardware and software latency. In conventional storage systems, the latency of slow storage devices (e.g., disks) dominates access latency, so software efficiency is not critical. Previous work has shown that with fast NVMM, software costs can quickly dominate memory latency, squandering the performance that NVMMs could provide [7, 12, 68, 74].

Since NVMM memories offer low latency and will be on the processor's memory bus, software should be able to access them directly via loads and stores. Recent NVMM-based file

systems [21, 71, 73] bypass the DRAM page cache and access NVMM directly using a technique called *Direct Access (DAX)* or *eXecute In Place (XIP)*, avoiding extra copies between NVMM and DRAM in the storage stack. NOVA is a DAX file system and we expect that all NVMM file systems will provide these (or similar) features. We describe currently available DAX file systems in Section 2.4.

Write reordering Modern processors and their caching hierarchies may reorder store operations to improve performance. The CPU's memory consistency protocol makes guarantees about the ordering of memory updates, but existing models (with the exception of research proposals [20, 46]) do not provide guarantees on when updates will reach NVMMs. As a result, a power failure may leave the data in an inconsistent state.

NVMM-aware software can avoid this by explicitly flushing caches and issuing memory barriers to enforce write ordering. The x86 architecture provides the `clflush` instruction to flush a CPU cacheline, but `clflush` is strictly ordered and needlessly invalidates the cacheline, incurring a significant performance penalty [6, 76]. Also, `clflush` only sends data to the memory controller; it does not guarantee the data will reach memory. Memory barriers such as Intel's `mfence` instruction enforce order on memory operations before and after the barrier, but `mfence` only guarantees all CPUs have the same view of the memory. It does not impose any constraints on the order of data writebacks to NVMM.

Intel has proposed new instructions that fix these problems, including `clflushopt` (a more efficient version of `clflush`), `clwb` (to explicitly write back a cache line without invalidating it) and `PCOMMIT` (to force stores out to NVMM) [26, 79]. NOVA is built with these instructions in mind. In our evaluation we use a hardware NVMM emulation system that approximates the performance impacts of these instructions.

Atomicity POSIX-style file system semantics require many operations to be atomic (i.e., to execute in an "all or nothing" fashion). For example, the POSIX `rename` requires that if the operation fails, neither the file with the old name nor the file with the new name shall be changed or created [53]. Renaming a file is a metadata-only operation, but some atomic updates apply to both file system metadata and data. For instance, appending to a file atomically updates the file data and changes the file's length and modification time. Many applications rely on atomic file system operations for their own correctness.

Storage devices typically provide only rudimentary guarantees about atomicity. Disks provide atomic sector writes and processors guarantee only that 8-byte (or smaller), aligned stores are atomic. To build the more complex atomic updates that file systems require, programmers must use more complex techniques.

2.3. Building complex atomic operations

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

Journaling Journaling (or write-ahead logging) is widely used in journaling file systems [24, 27, 32, 71] and databases [39, 43] to ensure atomicity. A journaling system records all updates to a journal before applying them and, in case of power failure, replays the journal to restore the system to a consistent state. Journaling requires writing data twice: once to the log and once to the target location, and to improve performance journaling file systems usually only journal metadata. Recent work has proposed back pointers [17] and decoupling ordering from durability [16] to reduce the overhead of journaling.

Shadow paging Several file systems use a copy-on-write mechanism called shadow paging [20, 8, 25, 54]. Shadow paging file systems rely heavily on their tree structure to provide atomicity. Rather than modifying data in-place during a write, shadow paging writes a new copy of the affected page(s) to an empty portion of the storage device. Then, it splices the new pages into the file system tree by updating the nodes between the pages and root. The resulting cascade of updates is potentially expensive.

Log-structuring Log-structured file systems (LFSs) [55, 60] were originally designed to exploit hard disk drives' high performance on sequential accesses. LFSs buffer random writes in memory and convert them into larger, sequential writes to the disk, making the best of hard disks' strengths.

Although LFS is an elegant idea, implementing it efficiently is complex, because LFSs rely on writing sequentially to contiguous free regions of the disk. To ensure a consistent supply of such regions, LFSs constantly clean and compact the log to reclaim space occupied by stale data.

Log cleaning adds overhead and degrades the performance of LFSs [3, 61]. To reduce cleaning overhead, some LFS designs separate hot and cold data and apply different cleaning policies to each [69, 70]. SSDs also perform best under sequential workloads [9, 14], so LFS techniques have been applied to SSD file systems as well. SFS [38] classifies file blocks based on their update likelihood, and writes blocks with similar "hotness" into the same log segment to reduce cleaning overhead. F2FS [30] uses multi-head logging, writes metadata and data to separate logs, and writes new data directly to free space in dirty segments at high disk utilization to avoid frequent garbage collection.

RAMCloud [44] is a DRAM-based storage system that keeps all its data in DRAM to service reads and maintains a persistent version on hard drives. RAMCloud applies log structure to both DRAM and disk: It allocates DRAM in a log-structured way, achieving higher DRAM utilization than other memory allocators [56], and stores the back up data in logs on disk.

2.4. File systems for NVMM

Several groups have designed NVMM-based file systems that address some of the issues described in Section 2.2 by applying one or more of the techniques discussed in Section 2.3, but none meet all the requirements that modern applications place on file systems.

BPFS [20] is a shadow paging file system that provides metadata and data consistency. BPFS proposes a hardware mechanism to enforce store durability and ordering. BPFS uses short-circuit shadow paging to reduce shadow paging overheads in common cases, but certain operations that span a large portion of the file system tree (e.g., a move between directories) can still incur large overheads.

PMFS [21, 49] is a lightweight DAX file system that bypasses the block layer and file system page cache to improve performance. PMFS uses journaling for metadata updates. It performs writes in-place, so they are not atomic.

Ext4-DAX [71] extends Ext4 with DAX capabilities to directly access NVMM, and uses journaling to guarantee metadata update atomicity. The normal (non-DAX) Ext4 file system has a data-journal mode to provide data atomicity. Ext4-DAX does not support this mode, so data updates are not atomic.

SCMFS [73] utilizes the operating system's virtual memory management module and maps files to large contiguous virtual address regions, making file accesses simple and lightweight. SCMFS does not provide any consistency guarantee of metadata or data.

Aerie [66] implements the file system interface and functionality in user space to provide low-latency access to data in NVMM. It has an optimization that improves performance by relaxing POSIX semantics. Aerie journals metadata but does not support data atomicity or mmap operation.

3. NOVA Design Overview

NOVA is a log-structured, POSIX file system that builds on the strengths of LFS and adapts them to take advantage of hybrid memory systems. Because it targets a different storage technology, NOVA looks very different from conventional log-structured file systems that are built to maximize disk bandwidth.

We designed NOVA based on three observations. First, logs that support atomic updates are easy to implement correctly in NVMM, but they are not efficient for search operations (e.g., directory lookup and random-access within a file). Conversely, data structures that support fast search (e.g., tree structures) are more difficult to implement correctly and efficiently in NVMM [15, 40, 65, 75]. Second, the complexity of cleaning logs stems primarily from the need to supply contiguous free regions of storage, but this is not necessary in NVMM, because random access is cheap. Third, using a single log makes sense for disks (where there is a single disk head and improving spatial locality is paramount), but it limits concurrency. Since NVMMs support fast, highly concurrent random accesses, using multiple logs does not negatively impact performance.

Based on these observations, we made the following design decisions in NOVA.

Keep logs in NVMM and indexes in DRAM. NOVA keeps log and file data in NVMM and builds radix trees [35] in DRAM to quickly perform search operations, making the in-NVMM data structures simple and efficient. We use a radix tree because there is a mature, well-tested, widely-used implementation in the Linux kernel. The leaves of the radix tree point to entries in the log which in turn point to file data.

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

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

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

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

each log page.

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

Do not log file data. The inode logs in NOVA do not contain file data. Instead, NOVA uses copy-on-write for modified pages and appends metadata about the write to the log. The metadata describe the update and point to the data pages. Section 4.4 describes file write operation in more detail.

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

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

4. Implementing NOVA

We have implemented NOVA in the Linux kernel version 4.0. NOVA uses the existing NVMM hooks in the kernel and has passed the Linux POSIX file system test suite [50]. The source code is available on GitHub: https://github.com/NVSL/NOVA. In this section we first describe the overall file system layout and its atomicity and write ordering mechanisms. Then, we describe how NOVA performs atomic directory, file, and mmap operations. Finally we discuss garbage collection, recovery, and memory protection in NOVA.

4.1. NVMM data structures and space management

Figure 1 shows the high-level layout of NOVA data structures in a region of NVMM it manages. NOVA divides the NVMM into four parts: the superblock and recovery inode, the inode tables, the journals, and log/data pages. The superblock contains global file system information, the recovery inode stores recovery information that accelerates NOVA remount after a clean shutdown (see Section 4.7), the inode tables contain inodes, the journals provide atomicity to directory operations, and the remaining area contains NVMM log and data pages. We designed NOVA with scalability in mind: NOVA maintains an inode table, journal, and NVMM free page list at each CPU to avoid global locking and scalability bottlenecks.

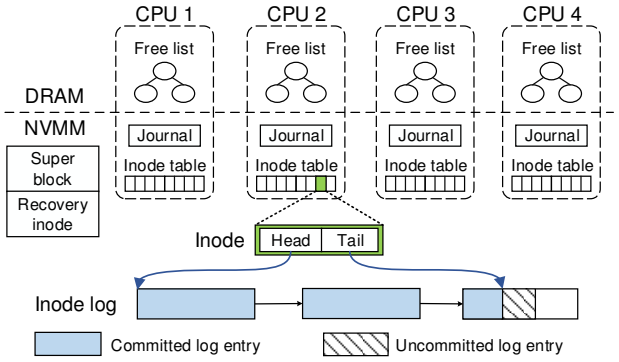

Figure 1: NOVA data structure layout. NOVA has per-CPU free lists, journals and inode tables to ensure good scalability. Each inode has a separate log consisting of a singly linked list of 4 KB log pages; the tail pointer in the inode points to the latest committed entry in the log.

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

A NOVA inode contains pointers to the head and tail of its log. The log is a linked list of 4 KB pages, and the tail always points to the latest committed log entry. NOVA scans the log from head to tail to rebuild the DRAM data structures when the system accesses the inode for the first time.

Journal A NOVA journal is a 4 KB circular buffer and NOVA manages each journal with a <enqueue, dequeue> pointer pair. To coordinate updates that across multiple inodes, NOVA first appends log entries to each log, and then starts a transaction by appending all the affected log tails to the current CPU's journal enqueue, and updates the enqueue pointer. After propagating the updates to the target log tails, NOVA updates the dequeue equal to enqueue to commit the transaction. For a `create` operation, NOVA journals the directory's log tail pointer and new inode's valid bit. During power failure recovery, NOVA checks each journal and rolls back any updates between the journal's dequeue and enqueue. NOVA only allows one open transaction at a time on each core and per-CPU journals allow for concurrent transactions. For each directory operation, the kernel's virtual file system

(VFS) layer locks all the affected inodes, so concurrent transactions never modify the same inode.

NVMM space management To make NVMM allocation and deallocation fast, NOVA divides NVMM into pools, one per CPU, and keeps lists of free NVMM pages in DRAM. If no pages are available in the current CPU's pool, NOVA allocates pages from the largest pool, and uses per-pool locks to provide protection. This allocation scheme is similar to scalable memory allocators like Hoard [5]. To reduce the allocator size, NOVA uses a red-black tree to keep the free list sorted by address, allowing for efficient merging and providing $O(\log n)$ deallocation. To improve performance, NOVA does not store the allocator state in NVMM during operation. On a normal shutdown, it records the allocator state to the recovery inode's log and restores the allocator state by scanning the all the inodes' logs in case of a power failure.

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

4.2. Atomicity and enforcing write ordering

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

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

Logging NOVA uses the inode's log to record operations that modify a single inode. These include operations such as `write`, `msync` and `chmod`. The logs are independent of one another.

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

Enforcing write ordering NOVA relies on three write ordering rules to ensure consistency. First, it commits data and log entries to NVMM before updating the log tail. Second, it commits journal data to NVMM before propagating the updates. Third, it commits new versions of data pages to NVMM before recycling the stale versions. If NOVA is running on a system that supports `clflushopt`, `clwb` and `PCOMMIT` instructions, it uses the code in Figure 2 to enforce the write ordering.

```
new_tail = append_to_log(inode->tail, entry);
// writes back the log entry cachelines
clwb(inode->tail, entry->length);
sfence();       // orders subsequent PCOMMIT
PCOMMIT();      // commits entry to NVMM
sfence();       // orders subsequent store
inode->tail = new_tail;
```

Figure 2: Pseudocode for enforcing write ordering. NOVA commits the log entry to NVMM strictly before updating the log tail pointer. The persistency of the tail update is not shown in the figure.

First, the code appends the entry to the log. Then it flushes the affected cache lines with `clwb`. Next, it issues a `sfence` and a `PCOMMIT` instruction to force all previous updates to the NVMM controller. A second `sfence` prevents the tail update from occurring before the `PCOMMIT`. The write-back and commit of the tail update are not shown in the figure.

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

4.3. Directory operations

NOVA pays close attention to directory operations because they have a large impact on application performance [37, 33, 64]. NOVA includes optimizations for all the major directory operations, including `link`, `symlink` and `rename`.

NOVA directories comprise two parts: the log of the directory's inode in NVMM and a radix tree in DRAM. Figure 3 shows the relationship between these components. The directory's log holds two kinds of entries: directory entries (dentry) and inode update entries. Dentries include the name of the child file/directory, its inode number, and timestamp. NOVA uses the timestamp to atomically update the directory inode's *mtime* and *ctime* with the operation. NOVA appends a dentry to the log when it creates, deletes, or renames a file or subdirectory under that directory. A dentry for a `delete` operation has its inode number set to zero to distinguish it from a create dentry.

NOVA adds inode update entries to the directory's log to record updates to the directory's inode (e.g., for `chmod` and `chown`). These operations modify multiple fields of the inode, and the inode update entry provides atomicity.

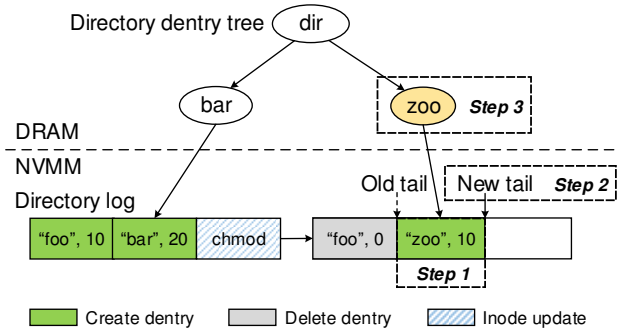

Figure 3: NOVA directory structure. Dentry is shown in <name, inode_number> format. To create a file, NOVA first appends the dentry to the directory's log (step 1), updates the log tail as part of a transaction (step 2), and updates the radix tree (step 3).

To speed up dentry lookups, NOVA keeps a radix tree in DRAM for each directory inode. The key is the hash value of the dentry name, and each leaf node points to the corresponding dentry in the log. The radix tree makes search efficient even for large directories. Below, we use file creation and deletion to illustrate these principles.

Creating a file Figure 3 illustrates the creation of file *zoo* in a directory that already contains file *bar*. The directory has recently undergone a `chmod` operation and used to contain another file, *foo*. The log entries for those operations are visible in the figure. NOVA first selects and initializes an unused inode in the inode table for *zoo*, and appends a create dentry of *zoo* to the directory's log. Then, NOVA uses the current CPU's journal to atomically update the directory's log tail and set the valid bit of the new inode. Finally NOVA adds the file to the directory's radix tree in DRAM.

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

4.4. Atomic file operations

The NOVA file structure uses logging to provide metadata and data atomicity with low overhead, and it uses copy-on-write for file data to reduce the log size and make garbage collection simple and efficient. Figure 4 shows the structure of a NOVA file. The file inode's log records metadata changes, and each file has a radix tree in DRAM to locate data in the file by the file offset.

A file inode's log contains two kinds of log entries: inode update entries and file write entries that describe file write

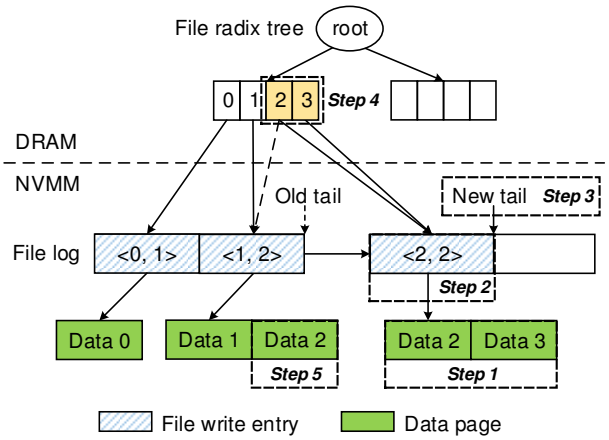

Figure 4: NOVA file structure. An 8 KB (i.e., 2-page) write to page two (<2, 2>) of a file requires five steps. NOVA first writes a copy of the data to new pages (step 1) and appends the file write entry (step 2). Then it updates the log tail (step 3) and the radix tree (step 4). Finally, NOVA returns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

operations and point to data pages the `write` modified. File write entries also include timestamp and file size, so that `write` operations atomically update the file's metadata. The DRAM radix tree maps file offsets to file write entries.

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

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

Figure 4 illustrates a `write` operation. The notation <*file pgoff*, *num pages*> denotes the page offset and number of pages a `write` affects. The first two entries in the log describe two `write`s, <0, 1> and <1, 2>, of 4 KB and 8 KB (i.e., 1 and 2 pages), respectively. A third, 8 KB `write`, <2, 2>, is in flight.

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

4.5. Atomic mmap

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

While DAX-mmap bypasses the file system page cache and avoids paging overheads, it presents challenges for programmers. DAX-mmap provides raw NVMM so the only atomicity mechanisms available to the programmer are the 64-bit writes, fences, and cache flush instructions that the processor provides. Using these primitives to build robust non-volatile data structures is very difficult [19, 67, 34], and expecting programmers to do so will likely limit the usefulness of direct-mapped NVMM.

To address this problem, NOVA proposes a direct NVMM access model with stronger consistency called `atomic-mmap`. When an application uses `atomic-mmap` to map a file into its address space, NOVA allocates *replica pages* from NVMM, copies the file data to the replica pages, and then maps the replicas into the address space. When the application calls `msync` on the replica pages, NOVA handles it as a `write` request described in the previous section, uses `movntq` operation to copy the data from replica pages to data pages directly, and commits the changes atomically.

Since NOVA uses copy-on-write for file data and reclaims stale data pages immediately, it does not support DAX-mmap. `Atomic-mmap` has higher overhead than DAX-mmap but provides stronger consistency guarantee. The normal DRAM mmap is not atomic because the operating system might eagerly write back a subset of dirty pages to the file system, leaving the file data inconsistent in event of a system failure [45]. NOVA could support atomic mmap in DRAM by preventing the operating system from flushing dirty pages, but we leave this feature as future work.

4.6. Garbage collection

NOVA's logs are linked lists and contain only metadata, making garbage collection simple and efficient. This structure also frees NOVA from the need to constantly move data to maintain a supply of contiguous free regions.

NOVA handles garbage collection for stale data pages and stale log entries separately. NOVA collects stale data pages immediately during `write` operations (see Section 4.4).

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

- A file write entry is dead, if it does not refer to valid data pages.

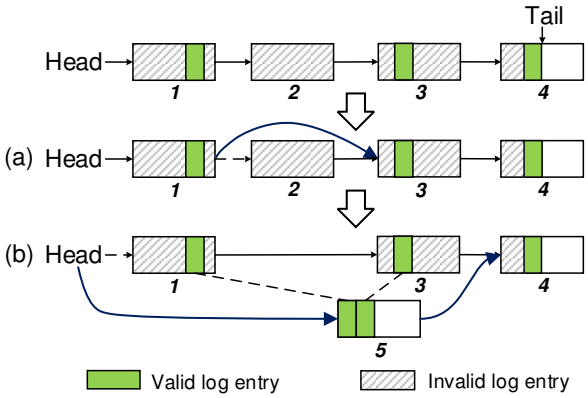

Figure 5: NOVA log cleaning. The linked list structure of log provides simple and efficient garbage collection. Fast GC reclaims invalid log pages by deleting them from the linked list (a), while thorough GC copies live log entries to a new version of the log (b).

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

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

These rules determine which log entries are alive and dead, and NOVA uses two different garbage collection (GC) techniques to reclaim dead entries.

Fast GC Fast GC emphasizes speed over thoroughness and it does not require any copying. NOVA uses it to quickly reclaim space when it extends an inode's log. If all the entries in a log page are dead, fast GC reclaims it by deleting the page from the log's linked list. Figure 5(a) shows an example of fast log garbage collection. Originally the log has four pages and page 2 contains only dead log entries. NOVA atomically updates the next page pointer of page 1 to point to page 3 and frees page 2.

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

Figure 5(b) illustrates thorough GC after fast GC is complete. NOVA allocates a new log page 5, and copies valid log entries in page 1 and 3 into it. Then, NOVA links page 5 to page 4 to create a new log and replace the old one. NOVA

does not copy the live entries in page 4 to avoid updating the log tail, so that NOVA can atomically replace the old log by updating the log head pointer.

4.7. Shutdown and Recovery

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

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

Recovery after a failure In case of a unclean dismount (e.g., system crash), NOVA must rebuild the NVMM allocator information by scanning the inode logs. NOVA log scanning is fast because of two design decisions. First, per-CPU inode tables and per-inode logs allow for vast parallelism in log recovery. Second, since the logs do not contain data pages, they tend to be short. The number of live log entries in an inode log is roughly the number of extents in the file. As a result, NOVA only needs to scan a small fraction of the NVMM during recovery. The NOVA failure recovery consists of two steps:

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

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

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

4.8. NVMM Protection

Since the kernel maps NVMM into its address space during NOVA mount, the NVMM is susceptible to corruption by errant stores from the kernel. To protect the file system and prevent permanent corruption of the NVMM from stray writes, NOVA must make sure it is the only system software that accesses the NVMM.

NOVA uses the same protection mechanism that PMFS does. Upon mount, the whole NVMM region is mapped as read-only. Whenever NOVA needs to write to the NVMM pages, it opens a write window by disabling the processor's write protect control (CR0.WP). When CR0.WP is clear, kernel software running on ring 0 can write to pages marked read-only in the kernel address space. After the NVMM write completes, NOVA resets CR0.WP to close the write window. CR0.WP is not saved across interrupts so NOVA disables local interrupts during the write window. Opening and closing the write window does not require modifying the page tables or the TLB, so it is inexpensive.

5. Evaluation

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

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

We first describe the experimental setup and then evaluate NOVA with micro- and macro-benchmarks.

5.1. Experimental setup

To emulate different types of NVMM and study their effects on NVMM file systems, we use the Intel Persistent Memory Emulation Platform (PMEP) [21]. PMEP is a dual-socket Intel Xeon processor-based platform with special CPU microcode and firmware. The processors on PMEP run at 2.6 GHz with 8 cores and 4 DDR3 channels. The BIOS marks the DRAM memory on channels 2 and 3 as emulated NVMM. PMEP supports configurable latencies and bandwidth for the emulated NVMM, allowing us to explore NOVA's performance on a variety of future memory technologies. PMEP emulates `clflushopt`, `clwb`, and `PCOMMIT` instructions with processor microcode.

In our tests we configure the PMEP with 32 GB of DRAM and 64 GB of NVMM. To emulate different NVMM technologies, we choose two configurations for PMEP's mem-

NVMM	Read latency	Write bandwidth	clwb latency	PCOMMIT latency
STT-RAM	100 ns	Full DRAM	40 ns	200 ns
PCM	300 ns	1/8 DRAM	40 ns	500 ns

Table 1: NVMM emulation characteristics. STT-RAM emulates fast NVMs that have access latency and bandwidth close to DRAM, and PCM emulates NVMs that are slower than DRAM.

ory emulation system (Table 1): For STT-RAM we use the same read latency and bandwidth as DRAM, and configure PCOMMIT to take 200 ns; For PCM we use 300 ns for the read latency and reduce the write bandwidth to 1/8th of DRAM, and PCOMMIT takes 500 ns.

We evaluate NOVA on Linux kernel 4.0 against seven file systems: Two of these, PMFS and Ext4-DAX are the only available open source NVMM file systems that we know of. Both of them journal metadata and perform in-place updates for file data. Two others, NILFS2 and F2FS are log-structured file systems designed for HDD and flash-based storage, respectively. We also compare to Ext4 in default mode (Ext4) and in data journal mode (Ext4-data) which provides data atomicity. Finally, we compare to Btrfs [54], a state-of-the-art copy-on-write Linux file system. Except for Ext4-DAX and Ext4-data, all the file systems are mounted with default options. Btrfs and Ext4-data are the only two file systems in the group that provide the same, strong consistency guarantees as NOVA.

PMFS and NOVA manage NVMM directly and do not require a block device interface. For the other file systems, we use the Intel persistent memory driver [48] to emulate NVMM-based ramdisk-like device. The driver does not provide any protection from stray kernel stores, so we disable the CR0.WP protection in PMFS and NOVA in the tests to make the comparison fair. We add clwb and PCOMMIT instructions to flush data where necessary in each file system.

5.2. Microbenchmarks

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

Figures 6(a) and 6(b) show the results on STT-RAM and PCM, respectively. The latency of fsync is amortized across the append operations. NOVA provides the lowest latency for each operation, outperforms other file systems by between 35% and 17×, and improves the append performance by 7.3× and 6.7× compared to Ext4-data and Btrfs respectively. PMFS is closest to NOVA in terms of append and delete performance. NILFS2 performs poorly on create operations, suggesting that naively using log-structured, disk-oriented file systems on NVMM is unwise.

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

Table 2: Filebench workload characteristics. The selected four workloads have different read/write ratios and access patterns.

NOVA is more sensitive to NVMM performance than the other file systems because NOVA's software overheads are lower, and so overall performance more directly reflects the underlying memory performance. Figure 6(c) shows the latency breakdown of NOVA file operations on STT-RAM and PCM. For create and append operations, NOVA only accounts for 21%–28% of the total latency. On PCM the NOVA delete latency increases by 76% because NOVA reads the inode log to free data and log blocks and PCM has higher read latency. For the create operation, the VFS layer accounts for 49% of the latency on average. The memory copy from the user buffer to NVMM consumes 51% of the append execution time on STT-RAM, suggesting that the POSIX interface may be the performance bottleneck on high speed memory devices.

5.3. Macrobenchmarks

We select four Filebench [23] workloads—fileserver, webproxy, webserver and varmail—to evaluate the application-level performance of NOVA. Table 2 summarizes the characteristics of the workloads. For each workload we test two dataset sizes by changing the number of files. The *small* dataset will fit entirely in DRAM, allowing file systems that use the DRAM page cache to cache the entire dataset. The *large* dataset is too large to fit in DRAM, so the page cache is less useful. We run each test five times and report the average. Figure 7 shows the Filebench throughput with different NVMM technologies and data set sizes.

In the fileserver workload, NOVA outperforms other file systems by between 1.8× and 16.6× on STT-RAM, and between 22% and 9.1× on PCM for the large dataset. NOVA outperforms Ext4-data by 11.4× and Btrfs by 13.5× on STT-RAM, while providing the same consistency guarantees. NOVA on STT-RAM delivers twice the throughput compared to PCM, because of PCM's lower write bandwidth. PMFS performance drops by 80% between the small and large datasets, indicating its poor scalability.

Webproxy is a read-intensive workload. For the small dataset, NOVA performs similarly to Ext4 and Ext4-DAX, and 2.1× faster than Ext4-data. For the large workload, NOVA performs between 36% and 53% better than F2FS and Ext4-DAX. PMFS performs directory lookup by linearly searching the directory entries, and NILFS2's directory lock

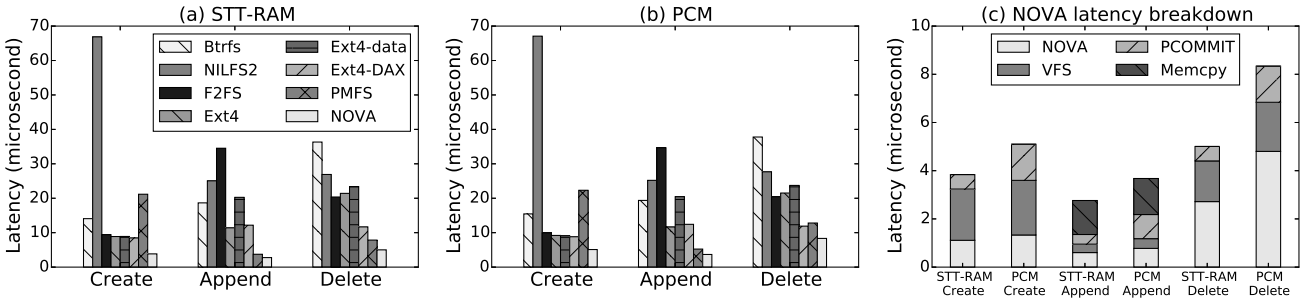

Figure 6: File system operation latency on different NVMM configurations. The single-thread benchmark performs create, append and delete operations on a large number of files.

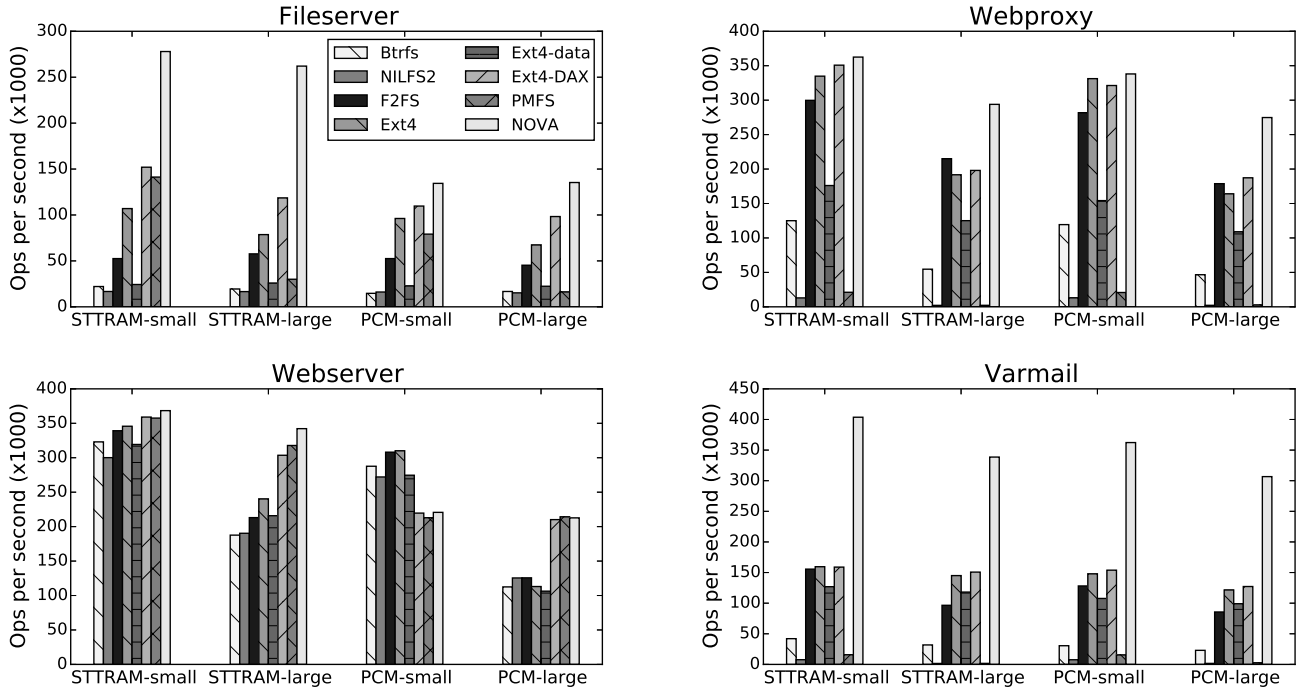

Figure 7: Filebench throughput with different file system patterns and dataset sizes on STT-RAM and PCM. Each workload has two dataset sizes so that the small one can fit in DRAM entirely while the large one cannot. The standard deviation is less than 5% of the value.

design is not scalable [57], so their performance suffers since webproxy puts all the test files in one large directory.

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

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

Duration	10s	30s	120s	600s	3600s
NILFS2	Fail	Fail	Fail	Fail	Fail
F2FS	37,979	23,193	18,240	Fail	Fail
NOVA	222,337	222,229	220,158	209,454	205,347
# GC pages					
Fast	0	255	17,385	159,406	1,170,611
Thorough	102	2,120	9,633	27,292	72,727

Table 3: Performance of a full file system. The test runs a 30 GB fileserver workload under 95% NVMM utilization with different durations, and reports the results in operations per second. The bottom three rows show the number of pages that NOVA garbage collector reclaimed in the test.

Overall, NOVA achieves the best performance in almost all cases and provides data consistency guarantees that are as strong or stronger than the other file systems. The performance advantages of NOVA are largest on write-intensive workloads with large number of files.

5.4. Garbage collection efficiency

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

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

Table 3 shows the result. NILFS2 could not finish the 10-second test due to garbage collection inefficiencies. F2FS fails after running for 158 seconds, and the throughput drops by 52% between the 10s and 120s tests due to log cleaning overhead. In contrast, NOVA outperforms F2FS by 5.8× and successfully runs for the full hour. NOVA's throughput also remains stable, dropping by less than 8% between the 10s and one-hour tests.

The bottom half of Table 3 shows the number of pages that NOVA garbage collector reclaimed. On the 30s test fast GC reclaims 11% of the stale log pages. With running time rises, fast GC becomes more efficient and is responsible for 94% of reclaimed pages in the one-hour test. The result shows that in long-term running, the simple and low-overhead fast GC is efficient enough to reclaim the majority of stale log pages.

5.5. Recovery overhead

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

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

Table 4: Recovery workload characteristics. The number of files and typical I/O size both affect NOVA's recovery performance.

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

Table 5: NOVA recovery time on different scenarios. NOVA is able to recover 50 GB data in 116ms in case of power failure.

To measure the recovery overhead, we use the three workloads in Table 4. Each workload represents a different use case for the file systems: Videoserver contains a few large files accessed with large-size requests, mailserver includes a large number of small files and the request size is small, fileserver is in between. For each workload, we measure the cost of mounting after a normal shutdown and after a power failure.

Table 5 summarizes the results. With a normal shutdown, NOVA recovers the file system in 1.2 ms, as NOVA does not need to scan the inode logs. After a power failure, NOVA recovery time increases with the number of inodes (because the number of logs increases) and as the I/O operations that created the files become smaller (because file logs become longer as files become fragmented). Recovery runs faster on STT-RAM than on PCM because NOVA reads the logs to reconstruct the NVMM free page lists, and PCM has higher read latency than STT-RAM. On both PCM and STT-RAM, NOVA is able to recover 50 GB data in 116ms, achieving failure recovery bandwidth higher than 400 GB/s.

6. Conclusion

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

Acknowledgments

This work was supported by STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA. We would like to thank John Ousterhout, Niraj Tolia, Isabella Furth, and the anonymous reviewers for their insightful comments and suggestions. We are also thankful to Subramanya R. Dulloor from Intel for his support and hardware access.

References

[1] Intel and Micron produce breakthrough memory technology. http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology.

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

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

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

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

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

[7] M. S. Bhaskaran, J. Xu, and S. 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] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems, 2007.

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

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

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

[12] A. M. Caulfield, T. I. Mollov, L. A. Eisner, A. De, J. Coburn, and S. Swanson. Providing safe, user space access to fast, solid state disks. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 387–400, New York, NY, USA, 2012. ACM.

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

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

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

[16] V. Chidambaram, T. S. Pillai, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Optimistic crash consistency. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 228–243, New York, NY, USA, 2013. ACM.

[17] V. Chidambaram, T. Sharma, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Consistency without ordering. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

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

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

[20] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 133–146, New York, NY, USA, 2009. ACM.

[21] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System Software

for Persistent Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 15:1–15:15, New York, NY, USA, 2014. ACM.

[22] 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.

[23] Filebench file system benchmark. http://sourceforge.net/projects/filebench.

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

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

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

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

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

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

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

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

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

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

[34] Persistent Memory Programming. http://pmem.io.

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

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

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

[38] C. Min, K. Kim, H. Cho, S.-W. Lee, and Y. I. Eom. SFS: Random Write Considered Harmful in Solid State Drives. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST '12, page 12, 2012.

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

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

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

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

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

[44] J. Ousterhout, A. Gopalan, A. Gupta, A. Kejriwal, C. Lee, B. Montazeri, D. Ongaro, S. J. Park, H. Qin, M. Rosenblum, S. Rumble, R. Stutsman, and S. Yang. The RAMCloud Storage System. *ACM Trans. Comput. Syst.*, 33(3):7:1–7:55, Aug. 2015.

[45] S. Park, T. Kelly, and K. Shen. Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 225–238, New York, NY, USA, 2013. ACM.

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

[47] T. S. Pillai, V. Chidambaram, R. Alagappan, S. Al-Kiswany, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. All File Systems Are Not Created Equal: On the Complexity of Craft-

ing Crash-Consistent Applications. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 433–448, Broomfield, CO, Oct. 2014. USENIX Association.

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

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

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

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

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

[53] POSIX 1003.1 - man page for rename. http://www.unix.com/man-page/POSIX/3posix/rename/.

[54] O. Rodeh, J. Bacik, and C. Mason. BTRFS: The Linux B-Tree Filesystem. *Trans. Storage*, 9(3):9:1–9:32, Aug. 2013.

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

[56] S. M. Rumble, A. Kejriwal, and J. 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.

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

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

[59] P. Sehgal, S. Basu, K. Srinivasan, and K. Voruganti. An Empirical Study of File Systems on NVM. In *Proceedings of the 2015 IEEE Symposium on Mass Storage Systems and Technologies (MSST'15)*, 2015.

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

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

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

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

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

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

[66] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift. Aerie: Flexible File-system Interfaces to Storage-class Memory. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 14:1–14:14, New York, NY, USA, 2014. ACM.

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

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

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

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

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

[72] M. Wu and W. Zwaenepoel. eNVy: A Non-volatile, Main

Memory Storage System. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-VI, pages 86–97, New York, NY, USA, 1994. ACM.

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

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

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

[76] Y. Zhang and S. Swanson. A Study of Application Performance with Non-Volatile Main Memory. In *Proceedings of the 2015 IEEE Symposium on Mass Storage Systems and Technologies (MSST'15)*, 2015.

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

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

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