System Software for Persistent Memory

Subramanya R Dulloor[1,3] Sanjay Kumar[1] Anil Keshavamurthy[2] Philip Lantz[1]
Dheeraj Reddy[1] Rajesh Sankaran[1] Jeff Jackson[1]

[1]Intel Labs, [2]Intel Corp, [3]Georgia Institute of Technology

Abstract

Emerging byte-addressable, non-volatile memory technologies offer performance within an order of magnitude of DRAM, prompting their inclusion in the processor memory subsystem. However, such load/store accessible Persistent Memory (*PM*) has implications on system design, both hardware and software. In this paper, we explore system software support to enable low-overhead PM access by new and legacy applications. To this end, we implement *PMFS*, a light-weight POSIX file system that exploits PM's byte-addressability to avoid overheads of block-oriented storage and enable direct PM access by applications (with memory-mapped I/O). PMFS exploits the processor's paging and memory ordering features for optimizations such as fine-grained logging (for consistency) and transparent large page support (for faster memory-mapped I/O). To provide strong consistency guarantees, PMFS requires only a simple hardware primitive that provides software enforceable guarantees of durability and ordering of stores to PM. Finally, PMFS uses the processor's existing features to protect PM from stray writes, thereby improving reliability.

Using a hardware emulator, we evaluate PMFS's performance with several workloads over a range of PM performance characteristics. PMFS shows significant (up to an order of magnitude) gains over traditional file systems (such as ext4) on a RAMDISK-like PM block device, demonstrating the benefits of optimizing system software for PM.

1. Introduction

In recent years, NAND flash has helped bring down the historically high performance gap between storage and memory [30]. As storage gets faster, the trend is to move it closer to the CPU. Non-Volatile DIMMs (*NVDIMMs*), for instance, attach storage directly to the scalable memory (DDR) interface [15, 38]. NVDIMMs are gaining popularity due to their ability to provide low-latency predictable performance at high rated IOPS [29, 38] But, despite being attached to the CPU, large capacity NAND-based NVDIMMs are still accessed as block devices [38] in a separate address space, due to the inherent properties of NAND [36].

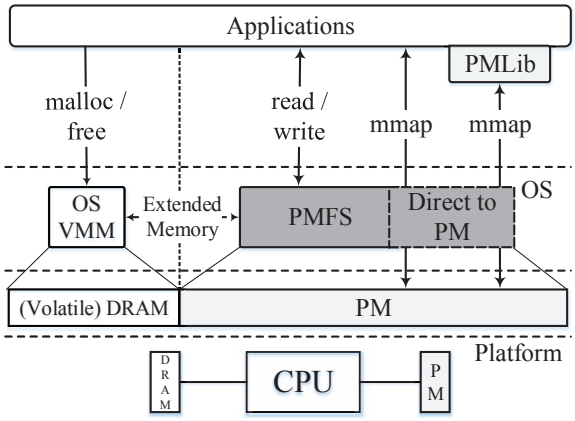


Figure 1: PM System Architecture

However, this appears likely to change in the near future due to emerging non-volatile memory technologies that are suited for use as large capacity, byte-addressable, storage class memory. Table 1 shows per-device characteristics of some of these technologies. We refer to such memory as *Persistent Memory (PM)*.

PM has implications on system architecture, system software, libraries, and applications [26, 27, 36, 40]. In this paper, we address the challenge of system software support to enable efficient access of PM by applications.

Traditionally, an OS separates the management of volatile memory (e.g., using a Virtual Memory Manager or VMM) and storage (e.g., using a file system and a block driver). Since PM is both byte-addressable (like volatile memory) and persistent (like storage), system software could manage PM in several ways, such as:

(1) extending VMM to manage PM;

(2) implementing a block device for PM for use with an existing file system (such as ext4);

EuroSys 2014, April 13-16, 2014, Amsterdam, ST, Netherlands.

http://dx.doi.org/10.1145/2592798.2592814

Parameter	DRAM	NAND Flash	RRAM	PCM
Density	1X	4X	2X-4X	2X-4X
Read Latency	60ns	25µs	200-300ns	200-300ns
Write Speed	~1GB/s	2.4MB/s	~140MB/s	~100MB/s
Endurance	10^{16}	10^4	10^6	10^6 to 10^8
Byte-Addressable	Yes	No	Yes	Yes

Table 1: Comparison of Memory Technologies [14, 36]

(3) implementing a file system optimized for PM without going through a block layer.

PMFS adopts the strategy of implementing a POSIX-compliant file system optimized for PM. Figure 1 shows a high-level overview of the proposed PM system architecture, with PMFS as the system software layer managing PM. PMFS has many advantages over the other two approaches:

(1) *Support for legacy applications.* Many storage-intensive applications rely on a traditional file system interface. PMFS implements a fully POSIX-compliant file system interface.

(2) *Support for a light-weight file system.* Given the anticipated performance characteristics of PM, the overheads from maintaining a separate storage address space (e.g., operating at block granularity and copying data between storage and DRAM) become dominant [21, 30, 44]. By optimizing for PM and avoiding the block layer, PMFS eliminates copy overheads and provides substantial benefits (up to 22×) to legacy applications. Figure 2 shows a high-level comparison of the two approaches. PMFS exploits PM's byte-addressability to optimize consistency using a combination of atomic in-place updates, logging at cacheline granularity (fine-grained journaling), and copy-on-write (CoW).

(3) *Optimized memory-mapped I/O.* Synchronously accessing fast storage with memory semantics (e.g., using memory-mapped I/O) has documented advantages [21, 44]. However, with traditional file system implementations, memory-mapped I/O would first copy accessed pages to DRAM, even when storage is load/store accessible and fast. PMFS avoids this overhead by mapping PM pages directly into an application's address space. PMFS also implements other features, such as transparent large page support [18], to further optimize memory-mapped I/O.

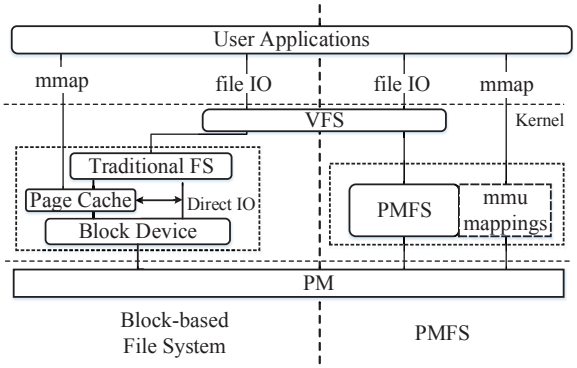


Figure 2: PMFS vs Traditional File Systems

PMFS presented us with several interesting challenges. For one, PMFS accesses PM as write-back (WB) cacheable memory for performance reasons [22], but still requires a way to enforce both ordering and durability of stores to PM. To further complicate this situation, memory writes in most modern architectures are posted, with memory controllers scheduling writes asynchronously for performance. This problem is common to all PM software and not just PMFS. To address this issue, we propose a hardware primitive, which we call PM write barrier or *pm_wbarrier*, that guarantees durability of stores to PM that have been flushed from CPU caches (§2).

For performance and simplicity, PMFS maps the entire PM into kernel virtual address space at the time of mounting. As a result, PM is exposed to permanent corruption from stray writes due to bugs in the OS or drivers. One solution is to map PM pages as read-only in the CPU page tables, and temporarily upgrade specific PM pages as writable in code sections that need to write to them. However, this requires expensive global TLB shootdowns [18]. To avoid these overheads, we utilize processor write protection control to implement uninterruptible, temporal, *write windows* (§3.3).

Another challenge in PMFS is validation and correctness testing of consistency. Though a well-known problem for storage software [34], consistency in PMFS is further complicated by the need for careful consideration of processor memory ordering and use of *pm_wbarrier* for enforcing durability. For PMFS validation, we use a hypervisor-based validation tool that uses record-replay to simulate and test for ordering and durability failures in PM software (§3.5) [31].

Finally, while memory-mapped I/O (mmap) does provide memory-like access to storage, the interface is too low-level for many applications. Recently researchers have proposed new programming models to simplify direct use of PM by applications [26, 39, 40]. We envision such programming models and libraries, referred to as *PMLib* in Figure 1, building on PMFS using mmap for direct access to PM. We intend to explore PMLib, including integration with PMFS, in the future.

Contributions of this paper are as follows:

- A high-level PM system architecture, including a simple new hardware primitive (*pm_wbarrier*) that provides software enforceable guarantees of durability and ordering of stores to PM.
- Design and implementation of PMFS, a light-weight POSIX file system with optimizations for PM and the processor architecture, such as fine-grained logging for consistency (§3.2), direct mapping of PM to applications with transparent large page support (§3.1), and a low-overhead scheme for protecting PM from stray writes by the kernel (§3.3).
- Detailed performance evaluation of PMFS with a PM hardware emulator, comparing PMFS with traditional file

systems on a RAMDISK-like Persistent Memory Block Device (*PMBD*).

In next section, we describe the proposed system architecture and *pm_wbarrier* primitive in detail (§2). We then present the design and implementation of PMFS (§3), followed by a detailed performance evaluation of PMFS (§4). Finally, we conclude the paper with a brief survey of related research and thoughts on future work.

2. System Architecture

Figure 1 shows the high-level system architecture assumed in the paper. For illustration purposes, we assume standard high-volume server platforms and processors based on Intel® 64-bit architecture, but the concepts are applicable to other architectures as well.

We assume a processor complex with one or more integrated memory controllers, capable of supporting both volatile DRAM and PM. The OS VMM continues to manage DRAM, while PMFS is responsible for managing PM.

A common flow for consistency in storage software such as PMFS requires a set of writes (*A*) to be durable before another set of writes (*B*) [4, 27]. Earlier research explored different approaches to managing ordering and durability of stores to PM, and implications on volatile CPU caches and store buffers. These include mapping PM as write-through (WT) [22], limiting PM writes to use non-temporal store instructions to bypass the CPU caches, or a new caching architecture for epoch based ordering [27].

In our evaluation, we encountered practical limitations with these approaches. While WT mapping offers the simplest solution to avoid caching related complications, it is not suited for use with PM due to both WT overheads [22] and limited PM write bandwidth (Table 1). Meanwhile, restricting PM stores to non-temporal instructions imposes programming challenges; for instance, in switching between cacheable loads and non-temporal stores. Non-temporal instructions also suffer from performance issues for partial cacheline writes, further discouraging general use. Finally, while an epoch-based caching architecture offers an elegant solution, it would require significant hardware modifications, such as tagged cachelines and complex write-back eviction policies. Such hardware mechanisms would involve non-trivial changes to cache and memory controllers, especially for micro-architectures with distributed cache hierarchies.

Based on our analysis, we found that using PM as WB cacheable memory and explicitly flushing modified data from volatile CPU caches (using *clflush* for instance) works well, even for complex usage. However, that alone is not sufficient for desired durability guarantees. Although *clflush* enables software to evict modified PM data and enforce its completion (using *sfence* for instance), it does not guarantee that modified data actually reached the durability point; i.e, to PM or some intermediate power-fail safe buffer.

In most memory controller designs, for performance and scheduling reasons, writes to memory are treated as posted transactions and considered complete once accepted and queued. Also, for all memory write requests accepted, the memory controller enforces processor memory ordering (e.g., read-after-write ordering) [19] by servicing reads of in-flight writes from internal posted buffers. For existing volatile memory usage, such behavior is micro-architectural and transparent to software. But PM usage has additional implications, particularly since the durability point is beyond the memory controller's posted buffers.

To provide PM software with durability guarantees in such an architecture, we propose a simple new hardware primitive (*pm_wbarrier*) that guarantees durability of PM stores already flushed from CPU caches. We envision two variants of this primitive: (1) an on-demand variant that allows software to synchronously enforce durability of stores to PM; and (2) a lazy variant that utilizes residual platform capacitance to asynchronously enforce durability of all in-flight accepted writes on detecting power failure. We assume an on-demand (synchronous) *pm_wbarrier* in this paper.

Consider the above mentioned software flow again. For the desired ordering (*A* before *B*) with the proposed primitive, PM software first writes back cachelines dirtied by stores to *A* (for instance, using *clflush*), issues an *sfence* for completion, and finally issues a single *pm_wbarrier*. At this point, *A* is guaranteed to be durable and software can proceed to the writes in *B*.

PMFS requires only *pm_wbarrier* for correct operation. The performance of cache write back is vital to the proposed PM architecture. However, as reported by previous work [22, 40], current implementations of *clflush* are strongly ordered (with implicit fences), and therefore suffer from serious performance problems when used for batch flush operations. For this paper, we assume and emulate an optimized *clflush* implementation that provides improved performance through weaker ordering on cache write back operations. Ordering is enforced in software with the use of memory fence operations (e.g., *sfence*). We used special write-combining stores to emulate the optimized *clflush* instruction, and observed up to 8× better performance compared to the strongly ordered *clflush* instruction (depending on the cacheline state).

For the remainder of the paper, we refer to optimized *clflush* simply as *clflush*. Also, unless specified otherwise, by making *A* durable, we mean the successful completion of the sequence of flushing dirty data in *A* from CPU caches (using *clflush*), completing the operation with an *sfence* or *mfence*, and enforcing durability with a single *pm_wbarrier*.

One drawback of the proposed scheme is that software is required to keep track of dirty cachelines in PM. Our evaluation shows that the resulting performance gains justify the additional programming complexity. Moreover, for normal applications, most of the nuances of PM program-

ming can be hidden behind programming models and libraries [26, 39, 40]. PMFS itself uses the proposed hardware primitives and the above mentioned software flow for most usage (for instance, in consistency and logging), and uses non-temporal stores sparingly for specific streaming write operations (e.g., in write system call).

Another important architectural decision is that of wear-leveling. As mentioned before, memory-mapped I/O in PMFS is optimized to grant direct PM access to applications. But, in doing so, one has to consider the issue of wear-leveling. We assume that wear-leveling is done in the hardware (e.g., in the PM modules), which simplifies our decision to map PM directly into the application's address space. We believe that software-based wear-leveling would be overly complicated, particularly when dealing with a large number of PM modules behind multiple memory controllers. We plan to explore this issue further in the future.

3. PMFS Design and Implementation

Figure 2 shows a high-level software architecture of a system using PMFS, including a comparison with a traditional file system. PMFS design goals are:

(1) *Optimize for byte-addressable storage.* PMFS exploits PM's byte addressability to avoid the overheads of the block-based design of traditional file systems, such as copies to DRAM during file I/O and read-modify-write at block granularity (as opposed to cacheline granularity) for consistency. PMFS design, including layout (§3.1) and consistency (§3.2), is optimized for PM and the processor architecture.

(2) *Enable efficient access to PM by applications.* Because PM performance is comparable to DRAM, it is important to eliminate software overheads in accessing PM [44]. For this reason, PMFS optimizes file read, write, and mmap by avoiding unnecessary copies and software overheads (§3.1). File read/write in PMFS, for instance, requires only a single copy between PM and user buffers, while mmap avoids copying altogether.

(3) *Protect PM from stray writes.* PMFS maps the entire PM into kernel virtual address space for performance, which exposes PM to permanent corruption from stray writes due to bugs in the OS or drivers. To prevent this, PMFS implements a prototype low-overhead write protection scheme using a write protect control feature in the processor (§3.3).

3.1 Optimizations for memory-mapped I/O

PMFS Layout: PMFS data layout is shown in Figure 3. The superblock and its redundant copy are followed by a journal (*PMFS-Log*) and dynamically allocated pages. As in many other file systems, metadata in PMFS is organized using a B-tree, one of the best options for indexing large amounts of possibly sparse data. The B-tree is used to represent both the inode table and the data in inodes.

Allocator: Most modern file systems use *extent-based* allocations (e.g., ext4, btrfs), while some older ones are *indirect block-based* (e.g., ext2). Allocations in PMFS are *page-based*, with support for all processor page sizes (4KB, 2MB, 1GB), to enable transparent large page support [18]. By default, the allocator uses 4KB pages for metadata (internal) nodes of a data file's B-tree, but data (leaf) nodes can be 4KB, 2MB, or 1GB.

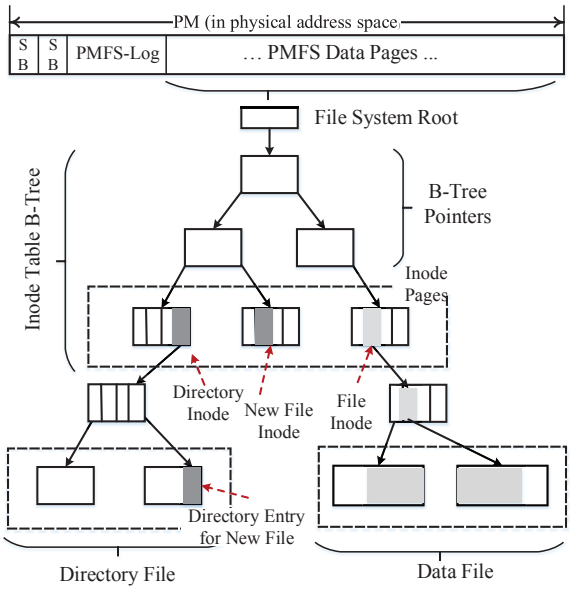


Figure 3: PMFS data layout

In many aspects, PMFS allocator is similar to the OS virtual memory allocator, except for consistency and durability guarantees (§3.2). Therefore, well-studied memory management issues, such as fragmentation due to support for multiple allocation sizes, apply to PMFS. In the current implementation, the PMFS allocator only coalesces adjacent pages to avoid major fragmentation. We plan to explore more strategies in the future.

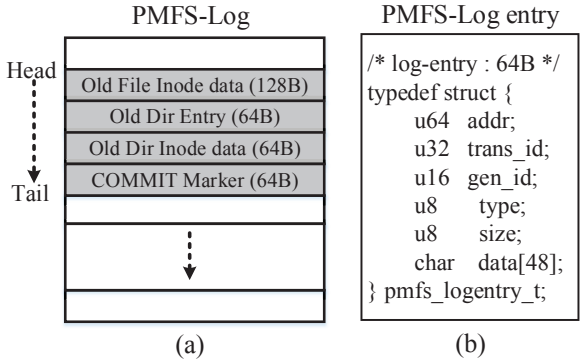


Figure 4: PMFS journaling data structures

Memory-mapped I/O (mmap): *Mmap* in PMFS maps file data directly into the application's virtual address space, so users can access PM directly. PMFS's *mmap* transparently chooses the largest hardware page table mappings, depending on *mmap* and file data node size. Using large page mappings has several benefits, such as efficient use of TLBs, fewer page faults, and shorter page table walks. Another

benefit of using large pages is smaller page table structures, which is even more important at large PM capacities since page table pages are allocated from limited DRAM.

However, using large pages without application hints could cause internal fragmentation. Therefore, by default, file data nodes in PMFS are 4KB. This default behavior can be overridden using one of the following strategies:
(1) Changing the default page size at mount time. This optimization works well for applications that use large files of fixed or similar sizes [7].
(2) Using existing storage interfaces to provide file size hints. If an application expects a file to grow to 10GB, for example, it can communicate this to PMFS using either *fallocate* (to allocate immediately) or *ftruncate* (to allocate lazily). These hints cause PMFS to use 1GB pages instead of 4KB pages for the file's data nodes. In our experience, modifying the applications to provide such file size hints is usually trivial. For instance, we were able to add basic *fallocate* support to Linux GNU coreutils [3] with just two lines of code, enabling large page allocations for file utilities such as *cp* and *mv*.

Finally, PMFS uses large page mappings only if the file data is guaranteed not to be copy-on-write (i.e., for files that are opened read-only or are *mmap*'ed as MAP_SHARED). Otherwise, we use 4KB mappings. We plan to provide switching between large page and 4KB mappings in the future, so we can use large pages in more cases.

3.2 Consistency

A file system must be able to survive arbitrary crashes or power failures and still remain consistent. To provide this guarantee, file systems must implement some form of consistent and recoverable update mechanism for metadata and (optionally) data writes. From the file system's view, an operation is *atomic* if and only if updates made by the operation are committed in all or none fashion. In PMFS, every system call to the file system is an atomic operation. Note that applications directly accessing PM via *mmap* have to manage consistency of the file contents on their own, either within the application or using language, library, and runtime support [26, 40].

Modern file systems and databases use one of the following three techniques to support consistency: copy-on-write (CoW) [2, 27], journaling (or logging) [4], and log-structured updates [30, 37].

CoW file systems and log-structured file systems perform CoW or logging at block or segment granularity, respectively. The block or segment size is typically 4KB or larger. These CoW and log-structured file systems are often accompanied by large write amplification, especially for metadata consistency which typically requires only small metadata updates. Journaling, particularly with PM, can log the metadata updates at much finer granularity.

We performed an analysis of the above mentioned techniques for PM, assessing the cost (number of bytes written, number of *pm_wbarrier* operations, etc.) of metadata consistency for various system calls. Based on this study, we found that logging at cacheline or 64-byte granularity (called *fine-grained logging*) incurs the least overhead for metadata updates, compared to both CoW (with or without atomic in-place updates) and log-structured file systems.

Journaling, however, has the drawback that all the updates have to be written twice; once to the journal, and then to the file system. For updates larger than a CoW file system's block size or a log-structured file system's segment size, journaling becomes less desirable due to this double copy overhead and the associated write amplification. Therefore, in PMFS, we use atomic in-place updates and fine-grained logging for the (usually small) metadata updates, and CoW for file data updates. We show (§4.2.3) that PMFS incurs much lower overhead for metadata consistency compared to BPFS, a PM-optimized file system that uses CoW and atomic in-place updates for metadata and data consistency [27].

Undo vs. Redo: Journaling comes in two flavors: 1) redo and 2) undo. In redo journaling, the new data to be written is logged and made durable before writing the data to the file system. This new data is written to the file system only when the transaction commits successfully. In undo journaling, the old data (in the file system) is first logged and made durable. The new data is written directly (in-place) to the file system during the transaction. In the event the transaction fails, any modifications to the file system are rolled back using the old data in the undo journal. Both redo and undo have pros and cons. Undo journaling in PMFS requires a *pm_wbarrier* for every log entry within a transaction while redo journaling requires only two *pm_wbarrier* operations per transaction, irrespective of the number of log entries. On the other hand, redo journaling is more complex to implement. Since the new data is written only to the redo journal during a transaction, all reads done as part of the transaction have to first search the redo journal for the latest copy. As a result, redo journaling incurs an additional overhead for all the read operations in a transaction, therefore placing practical restrictions on the granularity of logging in redo – the finer the logging granularity, the larger the overhead of searching the redo journal.

Undo journaling is simpler to implement and allows fine-grained logging of shared data structures (e.g., inode table). In PMFS, we use undo journaling for its above mentioned advantages and simplicity. However, we realize that undo is not always better than redo. For instance, redo could be expected to perform better than undo if the transaction creates a large number log entries but modifies only a small number of data structures. We plan to analyze the respective benefits of redo and undo journaling in the future.

As noted above, in PMFS, the original data is written and committed to the PMFS-Log (Figure 4) before the file system metadata is modified, thereby following undo seman-

tics. If a failure occurs in the middle of a transaction, PMFS recovers on the next mount by reading the PMFS-Log and undoing changes to the file system from uncommitted transactions. To minimize journaling overhead, PMFS leverages processor features to use atomic in-place updates whenever possible, sometimes avoiding logging altogether. For operations where in-place updates to the metadata are not sufficient, PMFS falls back to use fine-grained logging.

To summarize, PMFS uses a hybrid approach for consistency, switching between atomic in-place updates, fine-grained logging, and CoW. We now describe atomic in-place updates and fine-grained logging in more detail.

Atomic in-place updates: As suggested by previous work, PM provides a unique opportunity to use atomic in-place updates to avoid much more expensive journaling or CoW [27]. However, compared to prior work, PMFS leverages additional processor features for 16-byte and 64-byte atomic updates, avoiding logging in more cases. PMFS uses the various atomic update options in the following ways:

- 8-byte atomic updates: The processor natively supports 8-byte atomic writes. PMFS uses 8-byte atomic writes to update an inode's access time on a file read.
- 16-byte atomic updates: The processor also supports 16-byte atomic writes using *cmpxchg16b* instruction (with LOCK prefix) [17]. PMFS uses 16-byte in-place atomic updates in several places, such as for atomic update of an inode's size and modification time when appending to a file.
- 64-byte (cacheline) atomic updates: The processor also supports atomic cacheline (64-byte) writes if *Restricted Transactional Memory (RTM)* is available [20]. To atomically write to a single cacheline, PMFS starts a RTM transaction with XBEGIN, modifies the cacheline, and ends the RTM transaction with XEND, at which point the cacheline is atomically visible to rest of the system. On a subsequent *clflush*, the modified cacheline is written back to PM atomically, since the processor caching and memory hardware move data at least at cacheline granularity. PMFS uses cacheline atomicity in system calls that modify a number of inode fields (e.g., in deleting an inode). Note that if RTM is not present, PMFS simply falls back to use fine-grained logging.

Journaling for Metadata Consistency: PMFS uses undo journaling and fine-grained logging. The main logging data structure is a fixed-size circular buffer called PMFS-Log (Figure 4(a)). The head and tail pointers mark the beginning and end, respectively, of the sliding window of *active* logged data. For every atomic file system operation that needs logging, PMFS initiates a new transaction with a unique id (*trans_id*).

PMFS-Log consists of an array of 64-byte *log entries*, where each log entry describes an update to the file system metadata. A log entry consists of a header and data portion, as shown in Figure 4(b). The 2-byte *gen_id* is a special field. For a log entry in PMFS-Log to be considered valid by recovery code, the gen_id field in the log entry must match a similar gen_id field in PMFS-Log metadata. PMFS-Log's gen_id field is incremented after every log wrap-around and after every PMFS recovery, thereby automatically invalidating all of the stale log entries.

To be able to identify valid entries in PMFS-Log, one of two requirements must hold: either PMFS must atomically append entries to PMFS-Log or the recovery code must be able to detect partially written log entries. One possible solution is to use two *pm_wbarrier* operations: append the log entry to PMFS-Log and make it durable, then atomically set a valid bit in the log entry before making the valid bit durable. Other approaches include using a checksum in the log entry header [35] or *tornbit RAWL* [40], which converts the logging data in to a stream of 64-bit words, with a reserved torn (valid) bit in each word. However, all these approaches have high overhead from either additional serializing operations (double barrier) or compute (checksum, tornbit RAWL). In PMFS, we fix the size of log entries to a single (aligned) cacheline (64 bytes) and exploit the architectural guarantee in the processor caching hierarchy that writes to the same cacheline are never reordered. For example, if A and B are two separate 8-byte writes to the same cacheline and in that order, then A will always complete no later than B. PMFS uses the gen_id field in log entry header as a valid field. When writing a log entry to PMFS-Log, gen_id is written last, before the log entry is made durable. To ensure this scheme works, we instruct the compiler not to reorder writes to a log entry.

At the start of an atomic operation (or transaction), PMFS allocates the maximum number of required log entries for the operation by atomically incrementing the tail. When a transaction is about to modify any metadata, it first saves the old values by appending one or more log entries to PMFS-Log and making them durable, before writing the new values in-place. This process is repeated for all the metadata updates in the transaction. After all the metadata updates are done, the transaction is committed by first flushing all of the dirty metadata cachelines in PM and then using a single *pm_wbarrier* to make them durable. Finally, we append a special *commit* log entry to the transaction and make that durable, to indicate that the transaction has been completed.

As an optimization, we (optionally) skip the *pm_wbarrier* after the commit log entry. Some subsequent *pm_wbarrier*, either from another transaction or the asynchronous *log cleaner* thread will ensure that the commit log entry is made durable. This optimization avoids one *pm_wbarrier* per transaction, but the last committed transaction may be rolled back if PMFS has to recover from a failure. A log cleaner thread periodically frees up log entries corresponding to the committed transactions by first issuing a single

pm_wbarrier to make the commit log entries durable, and then atomically updating the head pointer in PMFS-Log.

PMFS recovery: If PMFS is not cleanly unmounted, due to a power failure or system crash, recovery is performed on the next mount. During recovery, PMFS scans through the PMFS-Log from head to tail and constructs a list of committed and uncommitted transactions, by looking for commit records with a valid gen_id. Committed transactions are discarded and uncommitted transactions are rolled back.

Consistency of Allocator: Using journaling to maintain the consistency of PMFS allocator data structures would incur high logging and ordering overhead, because of the frequency of alloc/free operations in the file system. Therefore, we maintain allocator structures in volatile memory, using freelists. PMFS saves the allocator structures in a reserved (internal) inode on a clean unmount. In case of a failure, PMFS rebuilds the allocator structures by walking the file system B-tree during recovery.

PMFS Data Consistency: As mentioned before, the journaling overhead for large file data updates could be prohibitive due to double copy. Hence, PMFS uses a hybrid approach, switching between fine-grained logging for metadata updates and CoW for data updates. For instance, on a multi-block file data update with the *write* system call, PMFS uses CoW to prepare pages with the new data and then updates the metadata using journaling.

In the current implementation, PMFS only guarantees that the data becomes durable before the associated metadata does. This guarantee is the same as the guarantee provided by ext3/ext4 in *ordered data* mode. We plan to explore stronger consistency guarantees in the future.

One open issue with CoW, however, is its use with large pages; for instance, CoW of a 1GB file data node, even if it is to write a few hundred megabytes, would cause significant write amplification. As this problem also occurs in OS virtual memory management, we plan to explore the best known practices (e.g., breaking down large pages to regular 4KB pages on CoW) in the future.

3.3 Write Protection

Since software can access PM as regular memory, we must be concerned about permanent corruption to PM from inadvertent writes due to bugs in unrelated software.

	User	Kernel
User	Process Isolation	SMAP
Kernel	Privilege Levels	Write windows

Table 2: Overview of PM Write Protection

Table 2 shows a brief overview of how PM write protection (from stray writes) works in the assumed processor architecture. The row name refers to the address space in which PM is mapped and the column name refers to the privilege level at which stray writes occur. Protection within a process (e.g., between multiple threads) is not covered here, but will be explored in the future. Protecting "kernel from user", and "user from user" follow from existing isolation mechanisms based on privilege levels and paging, respectively. Also, "user from kernel" follows from use of Supervisor Mode Access Prevention (SMAP) feature in the processor [13]. When SMAP is enabled, supervisor-mode (i.e., ring 0 or kernel) accesses to the user address space are not allowed. This feature is important since memory-mapped I/O in PMFS provides the user-level applications with direct access to PM. Due to SMAP, the *mmap*'ed memory is protected from stray writes in kernel. Protecting "kernel from kernel", however, could be challenging, particularly if multiple OS components sharing the same kernel address space are allowed to write to PM. Since PMFS is the only system software that manages PM, the solution space is much simpler in our case.

P: Read-only PM page in kernel virtual address
write(P): Write to page P in ring 0 (kernel)
GP: General protection fault

```
// CR0.WP in x86

if (ring0 && CR0.WP == 0)
   write(P) is allowed;
else
   write(P) causes GP;
```

```
// Using CR0.WP in PMFS
disable_write_protection() {
   CR0.WP = 0;
   disable_interrupts();
}
enable_write_protection() {
   enable_interrupts();
   CR0.WP = 1;
}
```

```
// Writes to PM in PMFS
disable_write_protection();
write(P);
enable_write_protection();
```

Figure 5: Write Protection in PMFS

One solution for "kernel to kernel" write protection is to not map PM into the kernel virtual address space at all, and to use only temporary mappings private to a physical core (similar to *kmap* in Linux for 32-bit x86 platforms). However, this approach adds complexity to PMFS software due to not being able to use fixed virtual addresses to access PM. The benefits of shared page table pages and large page mappings are also lost with temporary mappings.

Therefore, PMFS implements an alternate protection scheme, wherein it maps the entire PM as read-only during mount and upgrades it to writeable only for the sections of code that write to PM, called *write windows*. We could implement write windows by toggling write permission in the page table [18], but that would require an expensive global TLB shoot-down per write window. Instead, PMFS leverages the processor's write protect control (CR0.WP) [18] to implement the protection scheme described in Figure 5. PMFS exploits the following key attribute of CR0.WP: ring 0 or kernel writes to pages marked read-only in the kernel virtual address are allowed only if CR0.WP is not set. Therefore, by toggling CR0.WP, PMFS is able to open small

temporal write windows without modifying the page tables, thereby avoiding expensive TLB shootdowns.

One issue with CR0.WP, however, is that it is not saved/restored across interrupts or context-switches. To ensure PMFS doesn't lose control while PM pages are writeable, we disable interrupts for the duration of the write window. To ensure that interrupts aren't disabled for long, *disable_write_protection()* is invoked only when PMFS is about to write to PM. PMFS restricts the number of writes done in each write window to a few cachelines during fine-grained logging and to a small tunable limit (default 4KB) in the *write* system call. Following these writes, PMFS immediately invokes *enable_write_protection()*, thereby enabling the interrupts. We found this compromise to be reasonable in the prototype implementation for the given processor architecture.

Architectures with more generic *protection key* mechanisms (e.g., *Itanium*® and *PowerPC*®) would have fewer such limitations with write protection in PMFS [28]. For instance, in Itanium, it is possible to group sets of pages, each assigned a protection key. The TLB entry for a page has a tag that associates the page with the assigned protection key that now resides in the processor's *protection key registers* (PKRs). For a given process, the access right for a page is determined by combining the access bits on the corresponding page table entry and the PKR. Since modifying a PKR does not necessitate a global TLB shootdown, protection keys can be used to implement write windows with very low overhead, as we do with CR0.WP. However, with protection keys, PMFS can control access to PM at a page granularity and without disabling interrupts.

3.4 Implementation

The protoype PMFS implementation for the Linux 3.9 kernel, available open source [11], is written as a Linux kernel module and has about 10K lines of code. PMFS uses and extends the eXecute In Place (XIP) interface in the Linux kernel. When the underlying storage is byte-addressable, XIP provides a set of VFS callback routines that offer a clean and efficient way to avoid the page cache and block device layer. In PMFS, the *read*, *write*, and *mmap* callbacks registered with VFS are modeled after *xip_file_read*, *xip_file_write*, and *xip_file_mmap*, respectively. PMFS also implements the callback routine (*get_xip_mem*) that is used by XIP to translate a virtual address to a physical address in PM.

To support direct mapping of PM to the application, PMFS registers a page fault handler for the ranges in the application's address space that are backed by files in PMFS. This handler, which is invoked by the OS virtual memory subsystem, is modeled after *xip_file_fault* and extended to support transparent large page mappings.

During mount, PMFS maps PM as write-back cacheable, read-only memory. We found that both PMFS mount time and memory consumed by page table pages were very high using existing *ioremap* interfaces, even for moderately large PM (256GB in our case). Therefore, we implemented *ioremap_hpage* to automatically use the largest page size available (e.g., 1GB on Intel® 64-bit server processors) for mapping PM in to the kernel virtual address space. As expected, *ioremap_hpage* dramatically improves the mount time and significantly reduces memory consumed by page table pages.

POSIX System Calls: We now describe a few common file system operations in PMFS.

Creating a file: Creating a new file in PMFS involves three distinct metadata operations: 1) initializing a newly allocated inode, 2) creating a new directory entry in the directory inode and pointing it to the new file inode, and 3) updating the directory inode's modification time.

In the *create* system call, PMFS begins a new transaction and allocates space in PMFS-Log for the required log entries (5 total). PMFS first retrieves an inode from the freelist, logs the inode (two log entries), prepares it, and marks it as allocated. Next PMFS scans the directory inode's leaves for a new directory entry. Once found, PMFS logs the directory entry (one log entry) and updates it to point to the new file inode. Finally, PMFS logs and updates the directory inode's modification time (one log entry) and writes a commit record (one log entry) to complete the transaction.

Figure 3 shows the metadata updates for this operation (in dark shade). Figure 4(a) shows the corresponding log entries written during the transaction.

Writing to a file: In the *write* system call, PMFS first calculates the number of pages required to complete the operation. If no new allocations are needed and if we are only appending, PMFS writes the new data to the end of the file and uses a single 16-byte atomic write to update the inode size and modification time.

PMFS creates a new transaction for *write* only if it has to allocate new pages to complete the request. In this transaction, PMFS first allocates space for log entries in PMFS-Log, allocates the pages, and then writes data to PM using non-temporal instructions. Finally, PMFS logs and updates the inode's B-tree pointers and modification time, before writing a commit record to complete the transaction. Figure 3 shows the metadata updates for this operation (in light shade).

Deleting an inode: A file inode can be deleted only when there are no directory entries referencing it. When all the directory entries of an inode are deleted (e.g., using *rm*), the inode is marked for deletion. Once all of the handles to the inode are freed, VFS issues a callback to PMFS to delete the inode.

Freeing the inode involves atomically updating a set of fields in the inode, including the root of the inode's B-tree. By design, the inode fields for this operation are all in the same cacheline in PMFS. If RTM is available, PMFS uses a single 64-byte atomic write to free the inode. Otherwise, PMFS uses fine-grained logging. After successfully freeing the inode, all the inode data and metadata pages are returned

to the allocator. Since the allocator structures are maintained in DRAM and reconstructed on failure, freeing the pages to the allocator doesn't require logging.

3.5 Testing and Validation

Maintaining consistency in PMFS is challenging due to the requirement that PM software track dirty cachelines and flush them explicitly before issuing *pm_wbarrier*, to ensure PM store ordering and durability (§2). A natural consequence is additional complexity in testing and validation of PMFS. The same concern applies to any PM software. To address this challenge, we built *Yat*, a hypervisor-based framework designed to help validate that PMFS correctly and consistently uses cache flushes and *pm_wbarrier* [31].

Yat operates in two phases. In the first phase, Yat records a trace of all the writes, *clflush* instructions, ordering (*sfence*, *mfence*) instructions, and *pm_wbarrier* primitives executed while a test application is running. In the second phase, Yat replays the collected traces and tests for all the possible subsets and orderings of these operations, thereby simulating architectural failure conditions that are specific to PM. Yat includes an *fsck*-like tool to test the integrity of PM data in PMFS. For every simulated failure in PMFS testing, Yat automatically runs this tool, and provides diagnostic information if the verification fails. We found Yat to be extremely useful in catching several hard to find bugs in PMFS. Further discussion of Yat is beyond the scope of the paper.

4. Evaluation

We now describe the experimental setup and baselines for the evaluation of PMFS (§4.1). Then we present results from a detailed evaluation with several micro-benchmarks (*fio*, file utilities) and application benchmarks (Filebench, Neo4j).

4.1 Experimental Setup

PM Emulator System-level evaluation of PM software is challenging due to lack of real hardware. Publicly available simulators are either too slow and difficult to use with large workloads [36] or too simplistic and unable to model the effects of cache evictions, speculative execution, memory-level parallelism and prefetching in the CPU [10]. To enable the performance study of PM software for a range of latency and bandwidth points interesting to the emerging NVM technologies, we built a PM performance emulator: PM Emulation Platform (PMEP).

PMEP partitions the available DRAM memory into emulated PM and regular volatile memory, emulates configurable latencies and bandwidth for the PM range, allows configuring *pm_wbarrier* latency (default 100ns), and emulates the optimized *clflush* operation.

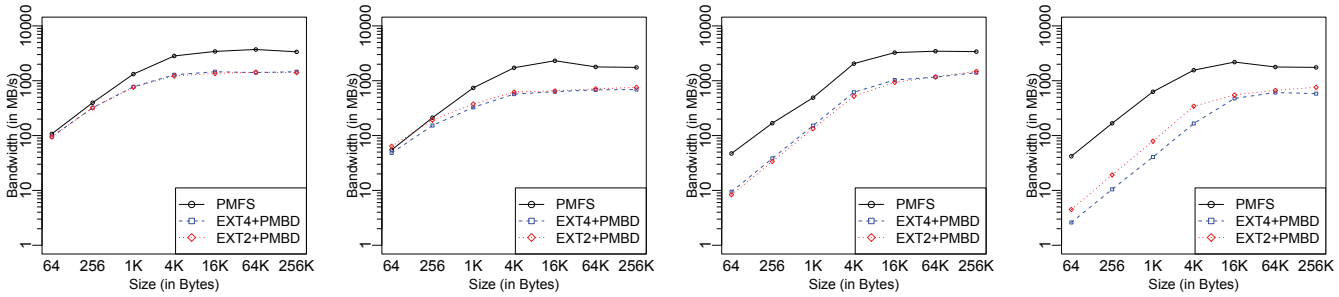
PMEP is implemented on a dual-socket Intel® Xeon® processor-based platform, using special CPU microcode and custom platform firmware. Each processor runs at 2.6GHz, has 8 cores, and supports up to 4 DDR3 Channels (with up to 2 DIMMs per Channel). The custom BIOS partitions available memory such that channels 2-3 of each processor are hidden from the OS and reserved for emulated PM. Channels 0-1 are used for regular DRAM. NUMA is disabled for PM channels to ensure uniform access latencies. Unless specified otherwise, PMEP has 16GB DRAM and 256GB PM, for a 1:8 capacity ratio. Next we describe the details of PMEP operation.

PM Latency Emulation: Emulating read latency is complicated due to CPU features such as speculative execution, memory parallelism, prefetching, etc. In PMEP, we exploit the fact that the number of cycles that the CPU stalls waiting for data to be available on a LLC miss ($C_{stalled}$) is proportional to the actual memory access latency (L_{dram}). We made use of debug hooks in the CPU and special microcode to program the hardware performance counters and monitor $C_{stalled}$ over very small intervals (by default, every 32 LLC misses). Smaller intervals lead to better emulation, but interval sizes smaller than the default could cause noticeable emulation overhead for memory-intensive applications. The CPU microcode emulates the desired PM latency (L_{pm}) by injecting $(C_{stalled} * (L_{pm}/L_{dram}))$ additional stall cycles for each interval. Correctness of this model follows from above mentioned proportionality. For the evaluations in the paper, L_{pm} is always set at 300ns (per Table 1), unless specified otherwise. Latency to local DRAM memory is 90ns.

PM Bandwidth Emulation : Emerging NVM technologies have significantly lower write bandwidth than DRAM (Table 1). While the overall write bandwidth could be improved using ample hardware scale-out of PM devices and power-fail-safe caches in the PM modules, the sustained bandwidth would still be lower than that of DRAM. For this reason, PMEP supports bandwidth throttling, by using a programmable feature in the memory controller [16] that can limit the maximum number of DDR transactions per μsec on a per-DIMM basis. Maximum sustained PM bandwidth (B_{pm}) for the entire platform is set to $9.5GB/s$, about $8\times$ lower than available DRAM bandwidth on the unmodified system.

PMBD : For a fair comparison of PMFS with traditional file systems designed for block devices, it is important to not only use the same storage media (emulated PM in our case), but to also choose a representative and optimized implementation of block device. Previous work used a Linux RAMDISK-like block device that relies on OS VMM for memory management [26, 27, 40]. However, RAMDISK cannot partition between PM and DRAM and is subject to interference from OS paging. For this reason, we use Persistent Memory Block Driver (*PMBD*) [23], an open-source implementation that assumes partitioned PM and DRAM. For block writes, PMBD uses non-temporal stores followed by a *pm_wbarrier* operation (§3.2).

We compare PMFS with both ext4 and ext2 on PMBD. Ext2, a much simpler file system with no consistency mechanisms, provides an interesting baseline for comparison with



a) Sequential read b) Sequential write c) Random read d) Random write

Figure 6: Evaluation of File I/O performance; X-axis is the size of the operation; Y-axis is the bandwidth in MB/s.

PMFS. We first evaluate the performance of traditional file-based I/O (using *fio*) and file system operations (with file utilities and Filebench). We demonstrate the performance benefits of atomic in-place updates and fine-grained journaling in PMFS, by comparing the (raw) overhead of metadata consistency in PMFS with that in ext4 and BPFS [27]. Then we evaluate the performance of memory-mapped I/O with both micro-benchmarks (*fio*) and application workloads (Neo4j). Finally, we evaluate the performance of the PMFS write protection scheme.

4.2 File-based Access

4.2.1 File I/O

Many applications use traditional file I/O interfaces (*read* and *write* system calls) to access file data. In a block-based file system, this usually requires two copies, one between the block device and the page cache, and one between the page cache and the user buffers. PMFS requires only one copy, directly between the file system and the user buffers. In Figure 6, we show a comparison of raw *read* and *write* performance using *fio*, a highly tunable open source benchmarking tool for measuring the performance of file I/O (*read*, *write*, *mmap*) for various access patterns [6]. We measure the performance of a single *fio* thread reading from or writing to a single large file (64GB). In case of the *write* tests, we flush data explicitly (using *fdatasync*) after every 256KB bytes.

As seen in Figure 6, PMFS performs better than both ext2 and ext4 for all the tests, with improvements ranging from 1.1× (for 64B sequential reads) to 16× (for 64B random writes). The drop in performance of writes for sizes larger than 16KB is due to the use of non-temporal instructions. These instructions bypass the cache but still incur the cache-coherency overhead from snooping CPU caches to invalidate the target cacheline. At larger sizes, this overhead is large enough to congest the write-combining buffers and cause a drop in performance. We found that the performance using non-temporal instructions is still better than if normal cacheable writes were used along with (optimized) cache flushes.

4.2.2 File Utilities

Next we evaluate the performance of common file and directory operations. We perform these tests with a tarball of

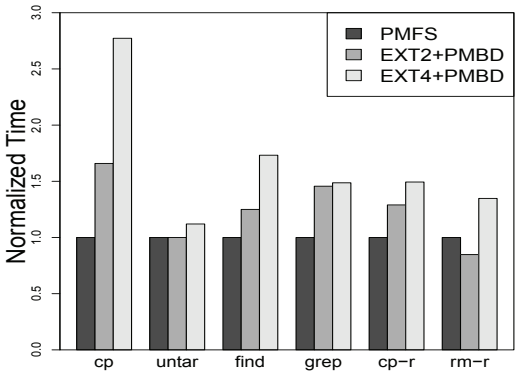


Figure 7: Evaluation of File utilities

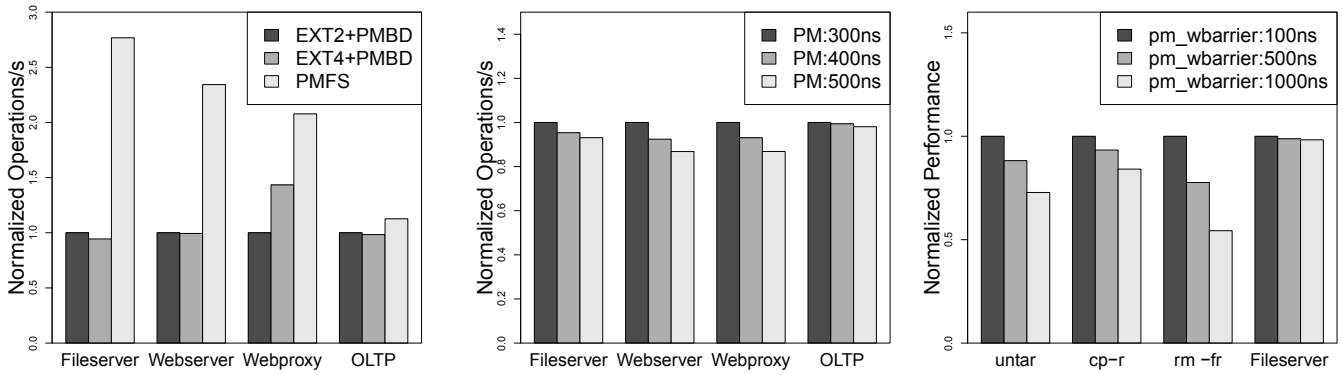
Linux Kernel sources in the file system under test. *cp* creates a copy of the tarball in the same directory. *untar* uncompresses and unpacks the tarball. *find* searches for an absent file in the uncompressed directory and *grep* searches for an absent pattern in the same directory. Finally, *cp-r* copies the entire directory and *rm-r* deletes the old copy. We measured the time taken for each of these operations. Figure 7 shows the relative performance. In all the tests other than *rm-r*, PMFS is faster than both ext2 and ext4, even while providing same consistency guarantees as ext4. PMFS performance gains, ranging from 1.01× (for *untar*) to 2.8× (for *cp*), are largely attributable to fewer copies. However, PMFS sees significant benefits from optimized consistency and atomic in-place updates as well, as seen with metadata intensive *rm-r* (in comparison with ext4). PMFS shows very little improvement with *untar* because *untar* involves only sequential writes from a single thread and is not very sensitive to latency.

Ext2 is faster than ext4 by up to 70% for some metadata intensive operations, due to the absence of journaling-related overheads. For the same reason, ext2 is about 20% faster than PMFS for *rm-r*.

4.2.3 PMFS Consistency

Next we evaluate the benefits of atomic in-place updates in PMFS and the raw logging overhead in PMFS logging compared to both block-based ext4 and PM-optimized BPFS.

Atomic in-place updates: PMFS exploits atomic updates to avoid logging whenever possible. To illustrate the benefits



a) Comparison with traditional file systems b) Effect of PM latency c) Effect of *pm_wbarrier* latency

Figure 8: Evaluation of Filebench performance

of 16-byte atomic updates, we measured the performance of a *write* system call that uses 16-byte atomic updates as described in §3.4. Using atomic updates, instead of fine-grained logging, speeds up the operation by 1.8×.

For the evaluation of 64-byte atomic updates, since *RTM* is not available in PMEP, we measured the RTM performance on Intel® Core™ i7-4770 processor [20]. Then we simulated the RTM performance in PMFS by replacing fine-grained logging with timed loops based on the above measurement. For deleting an inode (as described in §3.4), using 64-byte atomic updates is 18% faster than fine-grained logging.

Logging overhead : To illustrate the benefits of fine-grained (cacheline-sized) logging in PMFS, we measured the raw overhead of metadata consistency in PMFS, ext4, and BPFS. For PMFS and ext4, we measured the amount of metadata logged; for BPFS, we measured the amount of metadata copied (using CoW).

Table 3 shows the results for a number of file system operations that modify only the metadata. *touch* creates an empty file, *mkdir* creates an empty directory, and *mv* moves the file to the newly created directory. Since PMFS uses 64-byte log-entries instead of 4KB log-entries (ext4) or 4KB CoW blocks (BPFS), the raw overhead is 1-2 orders of magnitude lower for PMFS, even though BPFS itself uses in-place updates to reduce CoW overhead.

	PMFS	BPFS (vs PMFS)	ext4 (vs PMFS)
touch	512	12288 (24x)	24576 (48x)
mkdir	320	12288 (38x)	32768 (102x)
mv	384	16384 (32x)	24576 (64x)

Table 3: Metadata CoW or Logging overhead (in bytes)

4.2.4 Filebench

Filebench is a file system and storage benchmark that can simulate a variety of complex application workload I/O patterns [5]. In Figure 8, we present results from the evaluation of four multi-threaded application workloads from the Filebench suite.

Fileserver emulates I/O activity of a simple file server with workload similar to SPECsfs, and consists of creates, deletes, appends, reads, and writes. *Webserver* emulates a web-server with activity mostly comprised of complete file reads and log appends. *Webproxy* emulates a simple web proxy server, with a mix of create-write-close, open-read-close, and delete operations simultaneously from a large number of threads, combined with a file append to simulate proxy log. Finally, *OLTP* is a database emulator modeled after Oracle 9i, and consists of small random reads and writes coupled with synchronous writes to the log file.

In each case, we used the default number of threads — *Fileserver* has 50 threads; *Webserver* has 100 threads; *Webproxy* has 100 threads; and *OLTP* has a single thread doing log writes, 10 database writer threads, and 200 database reader threads. In each case, we scaled the dataset to at least 32GB to ensure that it does not fit in the page cache.

Figure 8(a) compares these workloads for PMFS, ext4, and ext2. PMFS performs better than both ext4 and ext2 for all the workloads, with gains ranging from 1.15× (for *OLTP*) to 2.9× (for *Fileserver*), confirming that PMFS does not have any obvious scaling issues for multi-threaded applications. As with the micro-benchmarks, the performance improvements with PMFS are due to fewer copies and optimized consistency.

OLTP shows relatively modest improvement with PMFS (about 15%), which is attributable to its low access rates to the file system. The small improvement with PMFS is due to the lower average latency of synchronous writes to the log file.

Effect of PM latency: Since PMFS accesses PM directly, we study its sensitivity to PM latency using Filebench as a case study. Figure 8(b) shows Filebench performance with PMFS as PM latency increases from 300ns to 500ns. For workloads other than *OLTP*, the observed drop is only about $10-15\%$, accumulated over both data and metadata accesses. Once again, *OLTP* shows minimal impact because it is sensitive only to synchronous write latency.

Effect of *pm_wbarrier* latency: PMFS uses the *pm_wbarrier* primitive to enforce durability of stores evicted from the cache. Figure 8(c) shows the performance of several file utilites (described above) and one of the Filebench work-

loads (*Fileserver*), as the latency of *pm_wbarrier* primitive is varied from 100ns to 1000ns. The drop in performance directly corresponds to the rate of *pm_wbarrier* operations for the workload. For instance, we measured a *pm_wbarrier* rate of over 1M/sec for metadata intensive *rm-r*, which shows the largest drop in performance (85%). This observation justifies the optimizations described in §3.2 to reduce the number of *pm_wbarrier* operations. Without these optimizations, the drop in performance would have been double that observed. *Fileserver*, on the other hand, has a *pm_wbarrier* rate of only around 40K/sec and doesn't show any noticeable drop in performance. We observed similarly low rates and results for other Filebench workloads.

4.3 Memory-mapped I/O

Memory-mapped I/O (*mmap*) in PMFS eliminates copy overheads by mapping PM directly to application's address space. In Figure 9(a)(b), we use *fio* to compare the performance of random reads and writes with *mmap*'ed data. We use *fallocate* to create a single large (64GB) file for the *mmap* test, thereby allowing the use of large (1GB) pages with *mmap* (§3.1). PMFS-D refers to the default case where *mmap* is done using regular 4KB pages. PMFS-P is exactly the same as PMFS-D, except for the use of MAP_POPULATE option with *mmap*. MAP_POPULATE causes the *mmap* system call to pre-populate the page tables at the time of *mmap*, thereby eliminating the runtime overhead of page faults. Finally, PMFS-L refers to the configuration where *mmap* is done using large (1GB) pages.

Compared to ext2 and ext4 on PMBD, PMFS-D is better by 2.3× (for 256KB random writes) to 14.3× (for 64B random writes) and PMFS-L is better by 5.9× (for 1KB random reads) to 22× (for 64B random writes). This significant improvement with PMFS is due to the fact that *mmap* in PMFS has zero copy overhead. Ext2 and ext4 have the overhead of moving the data between the page cache and PMBD. This overhead is much higher if the *mmap*'ed data does not fit in DRAM.

Compared to PMFS-D, PMFS-L is 1.3× to 2.3× better for random reads and 1.6× to 3.8× better for random writes. Performance gains with PMFS-L are due to several reasons: fewer page faults, better TLB efficiency, and shorter page table walks. In Figure 9(a), we quantify the overhead of page faults alone with PMFS-P. By eliminating the page faults, PMFS-P is able to improve the performance by 1.0× to 1.8× over PMFS-D. However, PMFS-L is still faster than PMFS-P by 1.1× to 1.4× due to better TLB efficiency and shorter page table walks. Figure 9(c) shows a more detailed breakdown of *mmap* performance, using 4KB random reads as an example. "Kernel" refers to the time spent in ring 0, either servicing page faults or copying data between DRAM and PMBD. "Page Table Walk" is the time that hardware spends walking the page table structures due to a TLB miss. "User" represents the time spent doing actual work in the application. Ext2 and ext4 spend only about 10% of the time doing useful work. As expected, PMFS-L has negligible overhead from page faults and page table walks, and therefore spends most of the time in "User". Counter-intuitively, PMFS-L also achieves a reduction in the actual application work. This behavior is attributable to the fact that PMFS-L has fewer instruction and LLC misses, because the fewer and shorter page walks in PMFS-L do not pollute the cache.

Note that apart from the above mentioned performance gains, PMFS-L also uses far less DRAM for page table pages. In the Intel 64-bit architecture, each page table leaf entry is 8B. PMFS-L requires only a 2-level page table, with each leaf entry covering a 1GB region. PMFS-D and PMFS-P require a 4-level page table in which each leaf entry covers only 4KB. To cover a 1GB region, PMFS-D and PMFS-P need 256K leaf entries or 2MB (256K×8B). These savings are particularly important, given the large PM capacities and limited DRAM.

4.3.1 Neo4j Graph Database

We use Neo4j to evaluate the benefits of PMFS to an unmodified application that uses memory-mapped I/O to access storage. Neo4j is a fully ACID (NoSQL) graph database aimed at connected data operations [9], with the database itself comprised of multiple store files. By default, Neo4j accesses these files by *mmap*'ing them, using Java's NIO API [8].

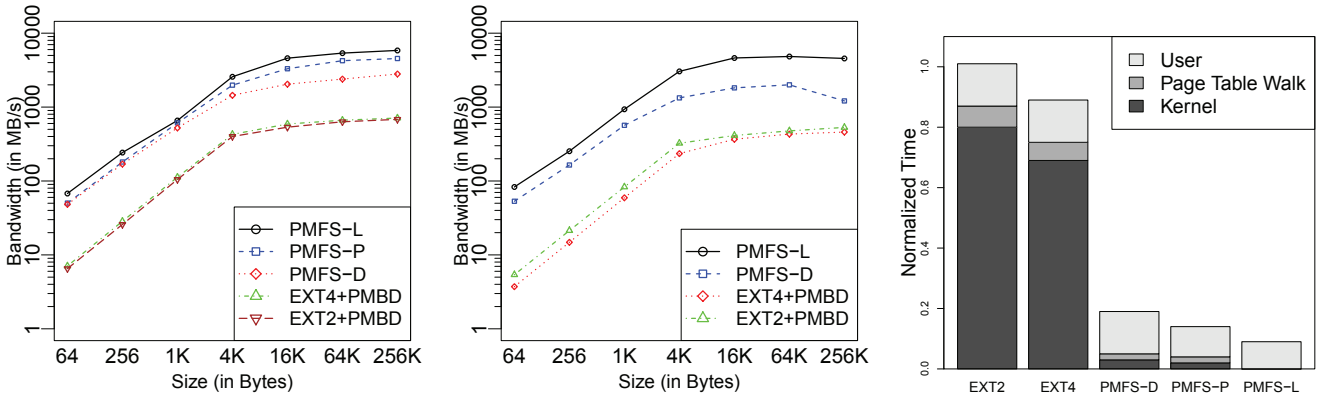
Neo4j supports both embedded (stand-alone) and client-server applications. For the evaluation in this paper, we wrote a sample Neo4j embedded application to measure the performance of common graph operations. We created a Neo4j graph database from the Wikipedia dataset [12]. Titles in the dataset are the nodes in the graph and links to the titles are the edges or relationships. The resulting graph has about 10M nodes and 100M edges. Both the nodes and edges have no other properties. We ran the following tests on this graph database :

(1) *Delete:* Randomly deletes 2000 nodes and their associated edges from the database, in a transaction.

(2) *Insert:* Transactionally adds back the previously deleted 2000 nodes and their relationships to the database.

(3) *Query:* For each query, selects two nodes at random and finds the shortest path between them. Traversal depth is bounded to avoid runaway queries. The test measures time taken for 500 such queries.

Since the size of the entire Neo4j database is less than 8GB, we limit the amount of DRAM in the system to 4GB for these tests, thereby ensuring that the database does not completely fit in the page cache. We ran these tests three times and measured their average run times. There was very little variation in the run times across the different runs. Figure 10 shows a comparison of PMFS with ext2 and ext4 over PMBD. In all the tests, PMFS is faster than both ext2 and ext4, with performance gains ranging from 1.1× (for *Insert*) to 2.4× (for *Query*). This improvement is attributable



a) Random read b) Random write c) Breakdown of *mmap* performance

Figure 9: Evaluation of memory-mapped I/O performance

to the fact that memory-mapped I/O in PMFS has no copy overhead.

Insert in Neo4j involves writing modifications to a Write-Ahead-Log (WAL) first, and then updating the graph store asynchronously. As with *OLTP*, performance of the *Insert* operation is sensitive only to synchronous sequential write latency. As a result, we see only a modest (11%) improvement with PMFS vs. ext2 and ext4.

Similarly, *Delete* in Neo4j uses WAL. But, the *Delete* operation requires more traversals to gather all of the nodes' relationships and properties, which will also be deleted in the transaction. PMFS is 20% faster than both ext2 and ext4. About half of this speedup is attributable to faster synchronous writes (as with *Insert*), and the remainder is due to faster traversals with PM.

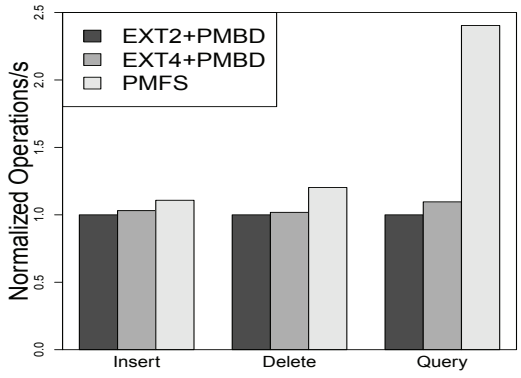
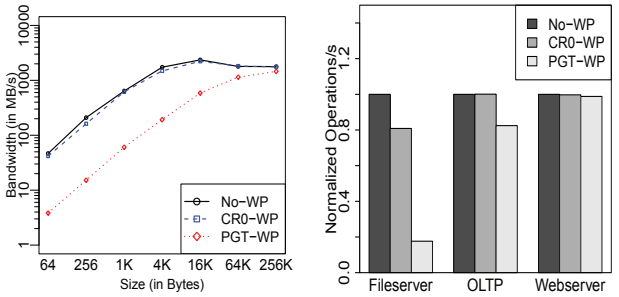


Figure 10: Evaluation of Neo4j performance

Although the performance gains, particularly for *Query*, are impressive for an unmodified application, they could be further improved by re-designing applications as suggested by prior work [26, 40] and using PMFS's memory-mapped I/O to access PM.

4.4 Write Protection

As described in §3.3, PMFS protects PM from stray writes in the OS using an existing processor write control feature (CR0.WP). In this section, we evaluate the performance overhead of this write protection scheme.



a) *fio* file write b) Filebench

Figure 11: Evaluation of PMFS write protection overhead

In Figure 11, we compare the write protection scheme in PMFS (CR0-WP) with the baseline case of no write protection at all (No-WP), and an alternate implementation that uses page table permissions (PGT-WP) (§3.3). As shown in Figure 11(a), CR0-WP is close to No-WP in performance and 1.2× to 11× faster than PGT-WP for file writes. Figure 11(b) shows a similar comparison for select Filebench workloads. For *Fileserver*, a multi-threaded workload with writes from several threads, CR0-WP is 23% slower than No-WP, but still 5× faster than the alternate PGT-WP scheme. The slowdown with CR0-WP is because writes to a processor control register (to toggle CR0.WP) are serializing operations with latency close to an uncacheable (UC) write.

With *OLTP*, where writes are mostly sequential and from a single log thread, CR0-WP overhead is negligible compared to No-WP, while PGT-WP is still 20% slower. For less write-intensive workloads, such as *Webserver*, the overheads are negligible for both CR0-WP and PGT-WP (as expected).

5. Related Work

File systems have always been optimized for the storage media [30, 37, 41]. As an example, recently several file systems have either been completely designed for flash or optimized for flash [1]. DirectFS, a file system optimized for (vendor-specific) flash-based storage, is one such exam-

ple [30]. PMFS is optimized for PM and the processor architecture.

Researchers have long explored attaching storage to the memory interface. eNVy is one such system that proposed attaching flash to the memory bus, using controller logic and SRAM to achieve high throughput at fairly low latency [42]. However, it is only recently that NVDIMM solutions are entering the mainstream, with a number of vendors providing either small capacity, byte-addressable NVDIMMs [15] or large capacity, block-addressable NVDIMMs [38]. Large capacity, byte-addressable PM is a natural evolution made possible by emerging NVM technologies.

Systems such as Rio File Cache [24] and, more recently, ConquestFS [41] suggested using a hybrid NVM-Disk or NVM-Flash for better performance, with the file system responsible for managing NVM efficiently. PMFS is designed for large capacity PM. We intend to explore tiering with other, possibly cheaper, storage technologies in the future.

Some researchers have proposed using a single-level store for managing PM, obviating the need to translate between memory and storage formats [32]. However, this requires significant changes to applications and does not benefit legacy storage applications. PMFS allows for a smooth transition from file-based access to memory-like semantics with PM by implementing a light-weight POSIX file interface and optimizing memory-mapped I/O.

PMFS is more directly comparable to file systems optimized for PM-only storage, such as BPFS [27]. BPFS uses copy-on-write (CoW) and 8-byte in-place atomic updates to provide metadata and data consistency. PMFS, in contrast, uses larger in-place atomic updates with fine-grained logging for metadata consistency, and CoW for data consistency. PMFS optimizes memory-mapped I/O, through direct mapping of PM to the application's address space and use of transparent large page mappings with *mmap*. BPFS doesn't support *mmap*. In PMFS, we also provide low overhead protection against stray kernel writes. BPFS doesn't address this issue. Finally, BPFS is designed assuming a very elegant hardware extension (*epochs*) for software enforceable guarantees of store durability and ordering. However, support for *epochs* requires complex hardware modifications. We assume only a simple *pm_wbarrier* primitive to flush PM stores to a power fail-safe destination. The decoupled ordering and durability primitives proposed in this paper are similar to the file system primitives, *osync()* and *dsync()*, proposed for OptFS [25].

SCMFS, like PMFS and BPFS, is a file system that is optimized for PM [43]. SCMFS leverages the OS VMM and the fact that the virtual address space is much larger than physical memory to layout the files as large contiguous virtual address ranges. PMFS, on the other hand, manages PM completely independent of the OS VMM, by implementing a page allocator that is optimized for the goals of PMFS. While the authors note that SCMFS uses only *clflush*/*mfence* for ordering, the details of consistency (in the presence of failures) are unclear. On the other hand, in PMFS, the design and implementation of consistency mechanisms are among the key contributions. Furthermore, consistency in PMFS has undergone careful and thorough validation (§3.5).

Programming for PM is tricky, particularly when PM accesses are cached write-back. We faced many challenges with PMFS ourselves, not least of which was validating and testing PMFS for correctness. Applications that want to operate directly on PM (for performance) encounter similar issues too. While *mmap* in PMFS does provide (efficient) direct access to PM, it is too low-level for many application programmers. *Failure-atomic msync* [33] addresses this by enabling atomic commit of changes to *mmap*'ed files. Implementing failure-atomic *msync* in PMFS is not difficult, but it is unclear if it is the right approach for PM. We will explore that in the future. Other researchers have proposed interesting programming models [26, 40] and library solutions [39] to simplify PM programming. These solutions, referred to as PMLib in Figure 1, complement our work in this paper, and can be built on the system-level PMFS services, such as naming, access control, and direct access to PM with *mmap*.

6. Conclusions and Future Work

This paper presents a system software architecture optimized for Persistent Memory (PM). We believe that the file system abstraction offers a good trade-off between supporting legacy applications and enabling optimized access to PM. We demonstrate this by implementing PMFS, a file system that provides substantial benefits (up to an order of magnitude) to legacy applications, while enabling direct memory-mapped PM access to applications.

However, a memory-mapped interface is too low level for use by many applications. User level libraries and programming models built on PMFS's memory-mapped interface could provide simpler abstractions (e.g., persistent heap) to applications. We are exploring this further, along with optimizations in PMFS for NUMA and PM's read and write performance asymmetry.

Acknowledgments

We thank Sergiu Gheti, Feng Chen, Vish Viswanathan, Narayan Ranganathan, and Ross Zwisler for their contributions to PMEP, PMBD, and PMFS. We also thank Karsten Schwan, Michael Mesnier, Alain Kägi, and Andy Rudoff for their feedback. Finally, we thank the anonymous Eurosys reviewers and our shepherd Kimberly Keeton for their help with the final revision.

References

[1] Solid State Drives. https://wiki.archlinux.org/index.php/Solid_State_Drives.

[2] Btrfs Wiki. https://btrfs.wiki.kernel.org.

[3] GNU Core Utilities. http://www.gnu.org/software/coreutils/.

[4] Ext4 Wiki. https://ext4.wiki.kernel.org.

[5] Filebench. http://sourceforge.net/apps/mediawiki/filebench.

[6] Flexible IO (fio) Tester. http://freecode.com/projects/fio.

[7] HDFS Architecture Guide. http://hadoop.apache.org/docs/stable/hdfs_design.html.

[8] Oracle Java Package java.nio. http://docs.oracle.com/javase/7/docs/api/java/nio/package-summary.html.

[9] Neo Technology. http://www.neo4j.org.

[10] pcmsim: A Simple PCM Block Device Simulator for Linux. https://code.google.com/p/pcmsim/.

[11] PMFS source code. https://github.com/linux-pmfs/pmfs.

[12] Wikimedia Downloads. http://dumps.wikimedia.org.

[13] Intel Architecture Instruction Set Extensions Programming Reference (sec 9.3), 2012.

[14] Crossbar Resistive Memory: The Future Technology for NAND Flash. http://www.crossbar-inc.com/assets/img/media/Crossbar-RRAM-Technology-Whitepaper-080413.pdf, 2013.

[15] Hybrid Memory: Bridging the Gap Between DRAM Speed and NAND Nonvolatility. http://www.micron.com/products/dram-modules/nvdimm, 2013.

[16] Intel Xeon Processor E5 v2 Product Family (Vol 2). http://www.intel.com/content/dam/www/public/us/en/documents/datasheets/xeon-e5-v2-datasheet-vol-2.pdf, 2013.

[17] Intel64 Software Developer's Manual (Vol 2, Ch 3.2), 2013.

[18] Intel64 Software Developer's Manual (Vol 3, Ch 4.5), 2013.

[19] Intel64 Software Developer's Manual (Vol 3, Ch 8.2), 2013.

[20] Intel64 Software Developer's Manual (Vol 1, Ch 14), 2013.

[21] A. Badam and V. S. Pai. SSDAlloc: Hybrid SSD/RAM Memory Management Made Easy. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, 2011.

[22] K. Bhandari, D. R. Chakrabarti, and H.-J. Boehm. Implications of CPU Caching on Byte-addressable Non-Volatile Memory Programming. http://www.hpl.hp.com/techreports/2012/HPL-2012-236.pdf, 2012.

[23] F. Chen, M. Mesnier, and S. Hahn. A Protected Block Device for Non-volatile Memory. LSU/CSC Technical Report (TR-14-01). 2014. URL http://www.csc.lsu.edu/~fchen/publications/abs/TR-14-01.html.

[24] P. M. Chen, W. T. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VII, 1996.

[25] 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, 2013.

[26] 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 XVI, pages 105–118, 2011.

[27] 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, 2009.

[28] C. Gray, M. Chapman, P. Chubb, D. Mosberger-Tang, and G. Heiser. Itanium: A System Implementor's Tale. In *Proceedings of the USENIX 2005 Annual Technical Conference*, ATC '05, 2005.

[29] K. Grimsrud. IOPS schmIOPS! What Really Matters in SSD Performance (Intel Corp). In *Proceedings of the 2013 Flash Memory Summit*, 2013.

[30] W. K. Josephson, L. A. Bongo, K. Li, and D. Flynn. DFS: A File System for Virtualized Flash Storage. *ACM Trans. Storage*, 6(3):14:1–14:25, Sept. 2010.

[31] P. Lantz, S. Dulloor, S. Kumar, R. Sankaran, and J. Jackson. Yat: A Validation Framework for Persistent Memory Software. Under Submission, 2014.

[32] J. Meza, Y. Luo, S. Khan, J. Zhao, Y. Xie, and O. Mutlu. A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory. In *Proceedings of Fifth Workshop on Energy Efficient Design*, WEED, 2013.

[33] 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, 2013.

[34] V. Prabhakaran, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Model-Based Failure Analysis of Journaling File Systems. In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, pages 802–811, 2005.

[35] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, SOSP '05, pages 206–220, 2005.

[36] 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, 2009.

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

[38] J. Scaramuzzo. Reaching the Final Latency Frontier (SMART Storage Systems). In *Proceedings of the 2013 Flash Memory Summit*, 2013.

[39] S. Venkataraman, N. Tolia, P. Ranganathan, and R. H. Campbell. Consistent and Durable Data Structures for Non-volatile Byte-addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies*, FAST'11, pages 5–5, 2011.

[40] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, pages 91–104, 2011.

[41] A.-I. A. Wang, G. Kuenning, P. Reiher, and G. Popek. The Conquest File System: Better performance Through A Disk/Persistent-RAM Hybrid Design. *ACM Trans. Storage*, 2(3):309–348, Aug. 2006.

[42] 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, 1994.

[43] 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, 2011.

[44] 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, 2012.