PMAL: Enabling Lightweight Adaptation of Legacy File Systems on Persistent Memory Systems

Hyunsub Song, Young Je Moon, Se Kwon Lee and Sam H. Noh
School of Electrical and Computer Engineering
UNIST (Ulsan National Institute of Science and Technology)

***Abstract*—The advent of Persistent Memory (PM), which is anticipated to have byte-addressable access latency in par with DRAM and yet nonvolatile, has stepped up interest in using PM as storage. Hence, PM storage targeted file systems are being developed under the premise that legacy file systems are sub-optimal on memory bus attached PM-based storage. However, many years of time and effort are ingrained in legacy file systems that are now time-tested and mature. Simply scrapping them altogether may be unwarranted. In this paper, we look into how we can leverage the maturity ingrained in legacy file systems to the fullest, while, at the same time, reaping the high performance offered by PM. To this end, we first go through a thorough analysis of legacy Ext4 file systems, and compare it with NOVA, PMFS, and Ext4 with DAX extension, which are new PM file systems available in Linux. Based on these analyses, we then propose the Persistent Memory Adaptation Layer (PMAL) module that is lightweight (roughly 180 LoC) and can easily be integrated into legacy file systems to take advantage of PM storage. Using Ext4, we show that the performance of PMAL integrated Ext4 is in par with PM file systems for the Filebench and key-value store benchmarks.**

I. Introduction

New memory technologies such as PCM, STT-MRAM and the recently announced 3D XPoint, which we henceforth refer to as Persistent Memory (PM), boasts performance in par with DRAM while providing nonvolatility. The advent of PM is anticipated to bring about considerable changes to the well-established computing framework. We anticipate memory and storage, and even CPU caches, to be completely replaced with PM [1]–[6], which allows for system state to be retained even upon reboot. This study follows this line of thought and is based on the premise that all of DRAM is replaced with PM, that is, a PM-only system.

Recently, there have been significant effort to develop storage systems targeted specifically for PM-based storage [7]–[16]. These research are based on the premise that legacy file systems are sub-optimal on memory bus attached PM storage. New approaches such as changing the existing system software architecture and removing the I/O stack from the file system have been suggested [7]–[12]. Such changes may improve performance, but integrating new techniques into current systems are usually quite difficult and cumbersome in various ways.

In this paper, we consider making use of legacy file systems for PM-based storage so that we can leverage the maturity ingrained in legacy file systems while, at the same time, reaping the high performance offered by PM. To this end, we first go through a thorough analysis of legacy file systems, even though for this study, we concentrate and report on the functionality and performance of only the Ext4 file system. Performance-wise we identify the modules that incur the most burden when used with PM storage. Also, we find that software overhead that is considered to be negligible in traditional disk-based storage can now have a serious effect on performance as their effects are amplified with fast PM. However, overall, we find that the main ingredient lacking in legacy file systems as a file system for PM storage is a component that can efficiently exploit PM.

Based on these analyses, we propose the Persistent Memory Adaptation Layer (PMAL), a lean software module composed of roughly 180 lines of code that intercepts handling of page cache writes and converts these writes to simultaneously write to PM storage. This has the effect of making page cache handling more efficient and providing journal mode journaling, which is avoided in traditional systems due to their heavy overhead, at virtually no extra cost. PMAL can be easily integrated into legacy file systems by simply altering the I/O flow to and from PMAL at particular points of the I/O stack. PMAL allows us to make use of legacy file systems instead of developing an entirely new file system that would require considerable time and effort to develop and mature. Using Ext4, we show that the performance of the as-is file system with PMAL integrated is in par with other PM file systems such as NOVA, PMFS and Ext4 with DAX extension, for the Filebench and key-value database benchmarks that we considered on the Linux platform.

The remainder of the paper is organized as follows. In the next section, we give a short review of some background and recent work related to PM focusing on file systems for PM. In Section III, we give a detailed analysis of the legacy Ext4 file system, breaking down the components of Ext4 in terms of functionality and performance. Then, in Section IV, we describe the design and implementation of the Persistent Memory Adaptation Layer (PMAL) module. Section V-A presents the evaluation platform and in the rest of Section V, we present and discuss the evaluation results. Finally, Section VI concludes the paper with a summary.

II. Background and Related Work

In this section, we first give a review of the characteristics of Persistent Memory (PM). Then, we review some of the recent file system related studies that have considered PM as storage.

978-1-5386-3890-3/17/$31.00 ©2017 IEEE

TABLE I: Characteristics of memory technologies

	NAND	STT -MRAM	PCM	DRAM
Nonvolatility	Yes	Yes	Yes	No
Access Unit	Page, Block	Byte	Byte	Byte
Read (ns)	2.5×10^4	10	20	10
Write (ns)	2×10^5	10	100	10
Endurance (#)	10^5	10^{15}	10^8	10^{15}

A. Persistent Memory

Persistent Memory (PM) technologies represented by PRAM or PCM (Phase Change RAM) [17], RRAM (Resistive RAM) [18], and STT-MRAM [19] are being considered as high performance storage mediums as they are nonvolatile and yet, provide random byte addressability and latency similar to DRAM. A recent announcement of the 3D XPoint technology by Intel and Micron has rekindled interest in this area where non-delivery of promised products had the community doubting whether such technologies would actually result in products [20]–[22].

Table I summarizes the performance forecasts of various PM technologies in comparison with existing memory technologies as reported by O'Sullivan et al. [23]. Latency and durability of STT-MRAM and PCM are expected to be better than those of NAND flash memory. In particular, the performance of STT-MRAM is expected to be in par with DRAM in terms of both latency and durability. This is the environment this study targets where DRAM is entirely replaced with medium that has STT-MRAM-like characteristics. Due to their nonvolatility, these memory technologies open a path for development of new storage software technologies totally different from traditional ones.

B. File Systems for Persistent Memory

Considerable work have been conducted in integrating PM technology into computer systems. These studies can be categorized largely into three realms: PM as a replacement or supplement of DRAM as main memory, PM as a new form of storage, and PM as simultaneously being main memory and storage. As the focus of this paper is on file systems for PM storage, we review previous work concentrating on implementations of file systems. Readers interested in a broader view of studies on the use of PM should refer to papers by Volos et al. [10] and Zhang et al. [16] and the references within.

File systems for memory bus attached PM storage that follow the traditional storage I/O path have been developed. BPFS is one of the earliest work on file systems for PM [7]. The key distinction of BPFS from traditional block unit file systems is that it guarantees consistency and ordering through a technique called Short Circuit Shadow Paging, which short circuits the cascading of copy-on-writes that can happen in conventional shadow paging file systems. This is based on the assumption that hardware primitives for 8-byte atomic writes and epoch barriers are supported. Another file system for PM storage is PMFS, a PM-dedicated file system that is open source available [9]. PMFS manages its metadata via a B-tree and introduces the hardware primitive `pm_wbarrier` to efficiently ensure the durability of every write to PM. It also makes use of fine-grained logging and copy-on-write techniques to guarantee consistency. There is also NOVA, a recent PM-dedicated file system that extends the traditional log-structured file system on to hybrid memory systems (DRAM/PM) [11]. This file system is also open source available. NOVA manages its metadata separately per CPU to ensure good scalability. It provides a cheaper atomic update technique than conventional journaling or shadow paging resulting in enhanced performance.

Efforts in developing file systems for PM have resulted in new file systems as just discussed. These new file systems, unfortunately, are born through a lot of effort and expense. They also have a limitation that as they are new, they are not as mature or time-tested as legacy file systems and thus, generally need considerable time before they are accepted as main stream file systems.

The Linux community has put effort into another line of work. In particular, the Linux kernel has introduced DAX for data access [24]–[26]. DAX makes use XIP (eXecution In-Place), which has been available for code execution since Ext2, for direct access of data and is supported in Ext4 and XFS. DAX allows the use of both the `mmap()` system call flow and the file I/O flow using the POSIX interface [27].

DAX, however, has a number of limitations. First, as DAX bypasses the page cache using Direct I/O, even though it is implemented within the framework of existing file systems, DAX does not make use of the page cache that inherently supports the mature and efficient features of Linux file systems [28]. For example, copying data between files can be done efficiently by making use of the page cache [24], [28]–[31], while other features such as mirroring, redundancy, repair, compression, consistency, encryption, etc. exploits the page cache infrastructure [24], [28]. Such cannot be simply replaced with mapping and issuing I/O operations to PM pages directly. Second, journaling of Direct I/O, which is being used by DAX, comes in limited form [32]. It currently supports ordered mode journaling with metadata commits to the journal being done only asynchronously. This results in compromised reliability even with fast media such as PM. Also, due to its inherent design journal mode journaling, which could be provided with little overhead with PM, cannot be provided.

In this paper, we seek a solution that keeps the key components of a file system intact. In particular, for Linux, we view the page cache as a significant component of the file system and seek a solution that makes efficient use of the page cache in a PM environment.

III. Analysis of File Systems

Recall that our goal is to retain the key characteristics of legacy file systems to take advantage of the time-tested maturity of these systems, while at the same time, taking advantage of the beneficial performance characteristics of PM without overhauling the file system. To this end, we propose the Persistent Memory Adaptation Layer (PMAL), a module

TABLE II: Description of components and their running time (ns) in `write()` I/O execution flow for various file systems

Component		Description	Ext4		DAX	PMFS	NOVA
			Async	Sync			
System Call	System call gate	Internal system call function	291	276	211	328	330
VFS Layer	VFS objects	Set structure related to VFS	980	973	899	923	845
	I/O type switch	Change type of I/O	3,182	5,715	2,223		
SFS Layer	Page cache	Work related to page cache	17,317	16,812			
	Memory I/O	Write data to memory	445	471			
	Page cache flush	Flush dirty page to storage		33,108			
	FS consistency	Mechanism for FS consistency		101,118	7,115		
	DAX|PMFS|NOVA	Write data to storage			13,213	19,058	19,256
Total Elapsed Time			22,215	158,473	23,661	20,309	20,431

that intercepts and alters the flow of I/O from traditional file systems to meet our goal.

In this section, we analyze the empirical findings such that they form the basis for the design of PMAL. Even though we have analyzed various popular file systems such as Btrfs and F2FS, our discussion focusses only on Ext4 as the findings from other file systems are generally similar to those of Ext4. For comparison, we also perform a breakdown of the I/O flow of Ext4-DAX, PMFS, and NOVA that are relatively new PM-dedicated file systems.

Through this analysis we wish to compare the various aspects of file systems and see how they affect the performance of file systems. We first analyze the file systems in function units (e.g., `vfs_write()` and `new_sync_write()` within, for example, the `write()` system call) and then, for ease of analysis and understanding, we classify these functions into components based on their main functionality within the file system as shown in Table II. This classification is based on analysis of source code of various file systems as provided in the Linux version 4.3.

Specifically, the components comprise the three layers of a call to the file system, that is, the system call, virtual file system, and specific file system layers. The components for the system call and virtual file system layers provide common functionality in the I/O flow differing only in their implementations. The components of the specific file system vary in terms of their functionality, the flow of execution depending on specific calls invoking the I/O, and in their implementations.

Based on this classification of functionality, we measure the time consumed by each of these components while executiing the `write()` call. As this study concentrates on Ext4, we explain our findings for Ext4 through comparison with other PM-dedicated file systems. The Ext4 file system is mounted on Ramdisk used to emulate a PM-based storage system. (The exact hardware platform in which these measurements were taken is discussed in Section IV-A.) The DAX, PMFS, and NOVA file systems are mounted on an emulated PM-based storage by using the *pmem* driver that is provided for PM-dedicated file systems in Linux [33]–[35]. The `ktime` [36] Linux kernel library is used to make the measurements. In particular, we measure the time spent at each of these components as a write request of a small sized (less than 4KB) data is made. The right hand side of Table II shows the results of the measurements, with the numbers being the average of 5 executions of each call. We now analyze the results in more detail.

System Call and VFS Layers: In the system call and virtual file system layers, we find that the main performance difference between Ext4 and other file systems comes from the 'I/O type switch' component. Basically, Ext4 has three I/O modes, namely, asynchronous, synchronous, and direct. On the other hand, PMFS and NOVA have only one I/O mode that directly performs I/O on PM storage. In the case of DAX, as DAX is implemented upon the Direct I/O flow of Ext4, DAX has an 'I/O type switch' component. Additionally, the difference between the asynchronous and synchronous I/O mode in Ext4 in the 'I/O type switch' component comes from the fact that synchronous I/O is simply asynchronous I/O plus some extras to immediately flush the page cache. Hence, synchronous I/O involves an additional I/O flow switch for page cache flush incurring higher overhead.

Specific File System `write()` call: Let us now observe the write I/O within the specific file system layer for Ext4. First, observe the asynchronous I/O results. Time is spent at only two components, namely, 'Page cache' and 'Memory I/O' as a write completes when data is written to the page cache. We see that the 'Page cache' component, which is mainly the software overhead for handling the page cache, consumes much more time than the 'Memory I/O' component, which is the component that does the actual memory write operation. Traditionally, software overhead for handling the page cache has been considered to be insignificant in terms of total performance in block-based storage systems [37]. However, the results here show that in systems with low latency storage such software overhead can have a large effect on overall storage performance.

We now turn to synchronous I/O. Recall synchronous I/O is simply asynchronous I/O plus some extras to flush the page cache. The 'Page cache flush' component is related to page cache flushing, whose main task is to manage data structures and I/O to backing storage. We see that considerably more time is spent in this component compared to other components in the upper layers. Another component that incurs significant overhead is the 'FS (File System) consistency' component. This component is the part that is performed to ensure consistency of the file system. For this, Ext4 performs journaling with ordered mode as default. The reason why these compo-

nents incur high overhead is because the implementation of the code is quite complicated. We emphasize again that when low latency storage such as PM is used, code efficiency can have a considerable effect on overall performance. This is unlike slow storage devices where storage is so slow that software efficiency has little effect.

In the Ext4 implementation (as well as other file systems in Linux), we find code that cleanly separates the 'Page cache flush' and 'FS consistency' components for easy manipulation. We take advantage of this later in our design and implementation given in Section IV.

We see that DAX, PMFS, and NOVA have no components related to the page cache as shown on the right end part of Table II. Each of the PM file systems has a component that writes to PM storage designated by the file system name. We see that this portion is significant for each of these file systems. DAX also has a separate 'FS consistency' component as DAX also uses journaling for file system consistency similarly to Ext4. However, there is high discrepancy between DAX and Ext4 (Sync) as journaling in DAX asynchronously commits metadata to the journal area. For reference, PMFS and NOVA use logging and lightweight atomic update to guarantee file system consistency, and this overhead is included in the respective components of the last row.

Summary of Observations and Analysis: Our analysis shows that there are a few key components in traditional file systems that incur considerable overhead. Specifically, they are the components that comprise the synchronous I/O portion of the `write()` call, namely, the 'Page cache flush' and 'FS consistency' components. To take advantage of PM storage with legacy file systems, these components, which are the biggest factors of performance degradation, need to be slimmed down and made efficient. Unfortunately, they cannot simply be replaced with a more efficient implementation. This is because these components, integrated within the page cache mechanism, are the ones that retain the mature attributes of legacy file systems as management of data and metadata is strongly integrated within them.

Recall that our goal is to retain the key characteristics of legacy file systems enabling us to take advantage of the time-tested maturity of these systems. In so doing, we want to also take advantage of the beneficial performance characteristics of PM. To this end, we devise a method to intercept the flow of I/O from traditional file systems at such a point that allows the traditional features to be retained as much as possible. Then, at this point, we insert a lightweight mechanism that makes use of PM. We implement this whole procedure as a module that we call the Persistent Memory Adaptation Layer (PMAL). We next discuss PMAL in detail.

IV. PMAL Design and Implementation

In this section, we describe the design and implementation of PMAL. In the next subsection, we discuss the PM setting that is assumed and enables us to accomplish our goal. Subsequently, we discuss the overall architecture of the system integrating PMAL. Then, the two components that comprise PMAL, that is, PMAL Flush and PMAL Consistency, are described in detail.

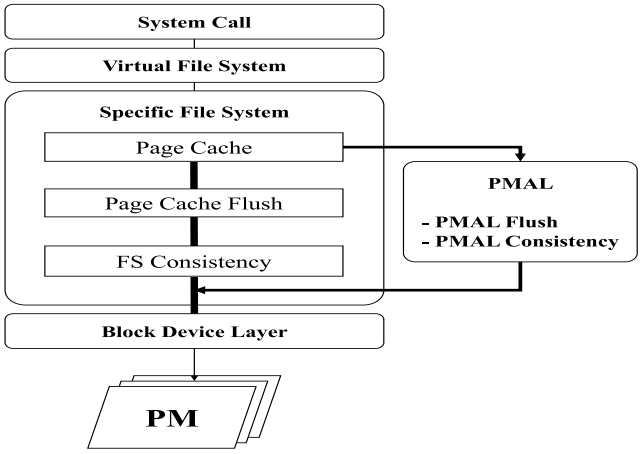


Fig. 1: PMAL integration into legacy file systems

A. PM-only Environment

Recall we assume that our system operates under the PM-only environment, where storage as well as main memory both have nonvolatile characteristics. This assumption provides a natural multi-versioning structure as was suggested by Lee et al. [38], [39] and Zhao et al. [4], where a modified version is retained in (nonvolatile) page cache and the (nonvolatile) LLC cache, respectively, while the original data is in another nonvolatile area. In this section, we first discuss how this environment is emulated. Then in subsequent sections, we discuss how we exploit this environment through PMAL.

To implement this system, we set the storage portion of DRAM emulated as PM through Ramdisk and format the storage portion for Ext4. This approach allows for the particular characteristics of the legacy file system to remain intact, while, for our case, having the extra benefit of eliminating the I/O scheduler functionality of the I/O stack. More specifically, with Ramdisk, Linux itself removes the I/O scheduler code and the disk I/O functions of the device driver are changed to memory I/O functions (e.g `memcpy()`), which is how I/O functions for devices should be implemented in PM-dedicated file systems. The remaining DRAM area, which is regarded as PM, is used as main memory. For memory persistency with PM, we adopt the same technique used with memory writes as was done with other PM-dedicated file systems [9], [33], [40], [41]. Specifically, we make use of the `clflush` & `sfence` instructions after calling the memory copy function in the Ramdisk and the page cache code section used by the PMAL module.

Moreover, we consider byte-level I/O in our system. Basically, because legacy file systems are designed for block-based storage, the byte-level I/O user requests are translated into block-level I/O requests at the generic block layer. To preserve byte-level I/O requests, we simply modify the existing write flow; specifically, where byte-level data is converted into block/page units to perform I/O at the Ramdisk driver level. In

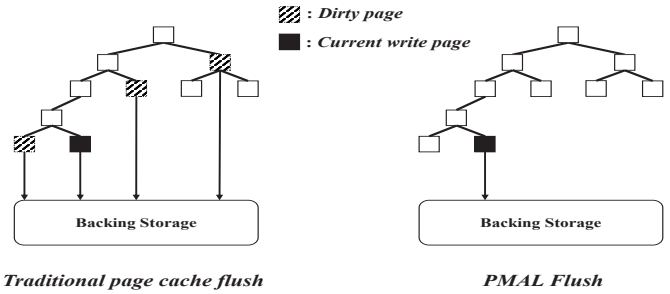


Fig. 2: Traditional page cache flush vs. PMAL Flush

PMAL, the Ramdisk driver code is modified to use the byte-level size of the write request that is obtained at the beginning of the write flow. This portion is implemented in roughly 30 lines of code (LoC).

B. PMAL Architecture

Our architecture of PMAL is established based on a key observation of measurement results given in Table II. That is, the results here show that the components related to the page cache and file system consistency make up the majority of the running time. The key to achieving our goal is, then, to optimize this portion of the file system. A simple method would be to bypass the page cache altogether as PM does not need a page cache as is done with DAX. However, this method is not feasible as the page cache is strongly integrated into the structure of legacy file systems [28]. Hence, pulling the page cache component out while leaving the inherent features of the file system intact is virtually impossible.

Our solution is to leave the page cache mechanism intact, but to insert a software module, which we call PMAL (Persistent Memory Adaptation Layer), that will take advantage of PM at the appropriate point of the I/O flow. PMAL is integrated into legacy file systems so that it intercepts the I/O flow from the page cache as shown in Figure 1. More specifically, after the data structures within the page cache are manipulated and the data is written to the page cache, an acknowledgement is sent to the upper layer. At this point, PMAL, which has two components, namely, PMAL Flush and PMAL Consistency, intercepts the acknowledgement and initiates a write to the file system. PMAL Flush is a component that improves on the traditional flushing mechanism, while PMAL Consistency provides journaling with minimal overhead. We describe each of these components in detail in the following sections. Once this write is done, the intercepted acknowledgement is resumed and sent to the upper layer where it will be accepted as completion of the write. We emphasize that all of PMAL is implemented in roughly 150 LoC. Specifically, ∼10 LoC for intercepting the I/O, ∼40 LoC for PMAL Flush, and ∼100 LoC for PMAL Consistency.

C. PMAL Flush Component

The PMAL Flush component that we devise allows us control over how and when to synchronously flush the current write request. In the traditional page cache flush mechanism of legacy file systems, a flush incurs writes of all dirty pages in

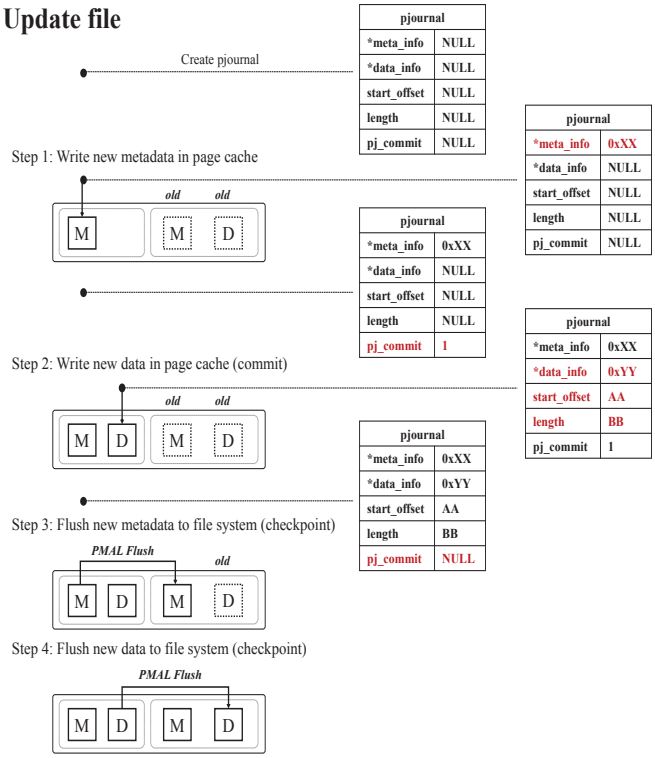


Fig. 3: Execution sequence of PMAL Consistency mechanism that provides journal mode journaling

the page cache to backing storage, as shown on the left hand side of Figure 2. This is a natural consequence of optimizing writes to slow disks. With PM, the story is different as writes are more efficient. Hence, in PMAL, we choose to flush data written to the page cache at our convenience. A positive effect of this is that the flush mechanism is simplified as flush occurs at any write point for only a small amount of data instead of waiting to make bulky writes. This leads to simple management code compared to the relatively heavy code in legacy systems. More importantly, this enhancement allows us to exploit it to provide a higher level of consistency, that is, journal mode journaling, with virtually no overhead, which we describe below.

D. PMAL Consistency Component

The second part of our solution, which we refer to as PMAL Consistency, is based on the ability to control flushes through PMAL Flush as described above. Recall that we assume that the entire memory is PM. Hence, the page cache is also PM meaning that writes to the page cache is already persistent. Hence, similarly to the observations of Lee et al. [38], [39], the page cache can simultaneously be considered to be the journal. By providing data structures to manage the data in the page cache, that is, the journal, and controlling the flushes to the file system area through PMAL Flush, commits and checkpointing of the journal can be controlled. This is what the PMAL Consistency component does, and in the end, this allows us to provide journal mode journaling, which is a higher

level of consistency than the default one provided by current legacy file systems, with virtually no overhead.

In the following, we describe the needed data structures and how we control the write sequence to achieve journal mode journaling. Then, we proceed to discuss the recovery mechanism used to maintain consistency upon system failure in conjunction with the journaling scheme.

Write Sequence for Journal Mode Journaling: Let us now discuss how journal mode journaling is provided in PMAL. For this, we make use of Figure 3, which depicts the step by step sequence for a write updating a file.

Upon a write request, metadata that we refer to as `pjournal`, which has elements `pj_commit`, `*metadata_info`, `*data_info`, `start_offset` and `length` all with initial values of NULL, is created. This `pjournal` is created per file when the file is opened and is removed when the file closes. Note that `pj_commit` will hold the current status of the write sequence, `*metadata_info` will hold the pointer to the new metadata (M) to be written (the address of the `inode` structure in Linux), and `*data_info` will hold the pointer to the kernel structure that manages the page cache where the new data (D) is to be written (in particular, the address of the `address_space` structure in Linux). The `start_offset` and `length` is the position within the file where the new data (D) is written to and the size of the write request, respectively. The only assumption regarding persistency needed for this process to work is that 8-byte writes to change the `pj_commit` value be atomic [9], [40], [41].

The sequence of changes occurring between Step 1 and Step 2 are as follows. First, before new metadata (M) is written into the journal (page cache), the pointer value to M is written to `*metadata_info`. Then, M is written into the journal. Finally, `pj_commit` is set to 1. Similarly, in between Step 2 and Step 3, first, the pointer value that points to the structure managing the page cache, where the new data (D) is to be written to, is written to `*data_info`. Then, the position and size values of the write request are written to `start_offset` and `length`, respectively. Then, after the new data (D) is written to the journal, `pj_commit` is set to NULL. Note that the order of the sequence is important to ensure recovery upon failure, which we discuss in more detail later.

The setting of `pj_commit` to NULL in Step 2 is equivalent to the journal commit upon which recovery of new data is guaranteed. Thereafter, Steps 3 and 4 are simply the checkpointing process of copying the data in the journal to the file system (backing storage). The sequence of writes in Steps 3 and 4 is controlled through PMAL Flush.

Finally, note that we have described the update process here. However, creating and writing a new file follows exactly the same sequence and the only difference is that the old values, depicted in Figure 3 as single dotted boxes and denoted *old* on top, would not exist. Hence, the PMAL Consistency mechanism does not save any particular field of `pjournal` and there is nothing to recover in case of failure.

Recovery Mechanism: The recovery process upon system failure does not have a special mechanism, but simply relies on the page cache flush daemon that periodically flushes the page cache. Since the page cache is the (nonvolatile) journal, upon reboot all data that existed in the page cache will still remain. However, some of them may not have been committed and hence, be invalid. So for the recovery mechanism, all we need to do is to make sure that all invalid data in the page cache is removed and valid data is kept as-is upon reboot. The page cache flush daemon will do the rest, that is, at its convenience write them to their respective file system areas. Distinguishing invalid data is done using the `pjournal` metadata in PMAL, whose value before failure is retained when the file system is recovered after system failure. Note that as we assume a PM-only system, all data and data structures in main memory before the system failure will remain intact after reboot. Such a state along with `pjournal` is all that is required.

Then, we first need to find the address of `pjournal` before recovery can start. This can be found in the descriptor of the files that were open during abnormal system termination due to failure, which are found in the process descriptor (in Linux, `task_struct`). Hence, we first find the processes that were modifying files in our PM storage before system failure, then search for files that remained open, where we find the `pjournal` address for that file. In particular, in Linux, the member structures of interest in the process descriptor are `files_struct` and `fdtable`.

When `pjournal` is found, the PMAL Consistency mechanism takes recovery action based on the `pj_commit` value:

`pj_commit` is NULL upon reboot: This status is divided into two cases. The first case is when `pjournal` was created but never used. Here, removing `pjournal` is all that is needed to do. The second case is when `pjournal` is set to NULL in Step 2. This means that the new write was committed and hence, they are valid data. Then, they are simply left in the page cache. This is all that needs to be checked. The page cache flush daemon will take care of the rest.

`pj_commit` is 1 upon reboot: This means that the new write was not able to commit, hence whatever data that had been written should be considered invalid and removed from the page cache. This is performed using the values of `*metadata_info`, `*data_info`, `start_offset` and `length`. NULL values for `*metadata_info` or `*data_info` means the data, whether M or D, was not written to the journal and no action needs to be taken for that data. If it is non-NULL, then using their respective pointer values, the data is evicted from the journal as the data is invalid. Specifically, in Linux, the functions used to perform such actions are `delete_from_page_cache()` and `remove_inode_hash()`.

E. Hybrid Memory and `mmap()` Support

In this subsection, we briefly touch on two matters that we did not study in depth, but is relevant to this study; one, on relaxing the PM-only assumption and the other, supporting the `mmap()` system call.

TABLE III: System configuration

	Description
CPU	Intel i7-4820K 3.7GHz (4 cores / 8 threads)
Memory	Samsung DDR3 8GB PC3-12800 × 8 (64GB)
OS	Linux CentOS 6.6 (64bit) kernel v4.3
PM storage	Emulated with Ramdisk (32GB)

We have so far discussed PMAL under the PM-only assumption. PMAL can also be implemented under the DRAM+PM hybrid memory assumption, in particular, assuming that the page cache is volatile. In such a case, extra multi-versioning area such as the journal would be needed within the PM. The main benefit of this version of PMAL compared to legacy file systems is that the PMAL Flush component can take advantage of small sized flushes to reduce the overhead of page cache flushes. Such changes has performance implications as we discuss in Section V-E.

Regarding the `mmap()` system call, though we do not implement it, `mmap()` can also be supported in PMAL with no additional design effort. This is because the `mmap()` flow is similar to the write/read flow as it uses the page cache, the main difference being that `mmap()` uses `msync()` to flush the data. Thus, `mmap()` can be supported in PMAL simply by calling the PMAL Flush component instead of `msync()`.

V. Experimental Results

In this section, we first present the experimental platform and the benchmarks used. We then discuss various aspects of the performance results.

A. Experiment Platform and Benchmarks

To evaluate the effectiveness of the PMAL approach, we implement PMAL and integrate it into the Ext4 file system in the Linux kernel version 4.3. This version is chosen to compare PMAL against recent PM-dedicated file systems as the only recent Linux version in which both PMFS and NOVA are supported is version 4.3 [34], [35]. Three variations of Ext4 are used, namely, PMAL integrated Ext4 denoted PMAL, Ext4 using DAX denoted DAX, and Ext4 using asynchronous I/O denoted Ext4-A, which is provided as a reference. In particular, in integrating PMAL into Ext4, we remove from Ext4 the code section related to file system consistency. Specifically, within the flushing of data from the page cache to backing storage, we remove all code that commits data to the journal in the synchronous write flow.

The specifications of the experimental platform on which the experiments are conducted are summarized in Table III. Specifically, of the 64GB DRAM space, 32GBs are used to emulate PM storage and is set as the capacity of the file system, leaving the rest to be used as memory. This PM is set as Ramdisk as mentioned in Section IV-A.

For the experiments, we use the Filebench macro benchmarks [42], which are popular real life-like workload generating benchmarks used to evaluate file systems. In particular, we make use of the Fileserver, Webserver, Webproxy, Varmail and OLTP benchmarks. Each workload represents write intensive, read intensive, strong access locality, sync intensive, and database workloads, respectively. The basic characteristics of these benchmarks are summarized in Table IV with the footprint that each of the workloads see being set to be between 10 to 15GB.

TABLE IV: Characteristics of workloads

	R:W	Mean file size	# of files	# of threads
Fileserver	1:2	128K	100K	50
Webserver	10:1	32K	500K	50
Webproxy	5:1	32K	400K	50
Varmail	1:1	16K	800K	50
OLTP	1:1	1.5G	10	W:10 R:200
	R:W	Record selection	Dataset size	# of threads
YCSB-A	1:1	Zipfian	10G	5
ForestDB	2:1	Zipfian	15G	5

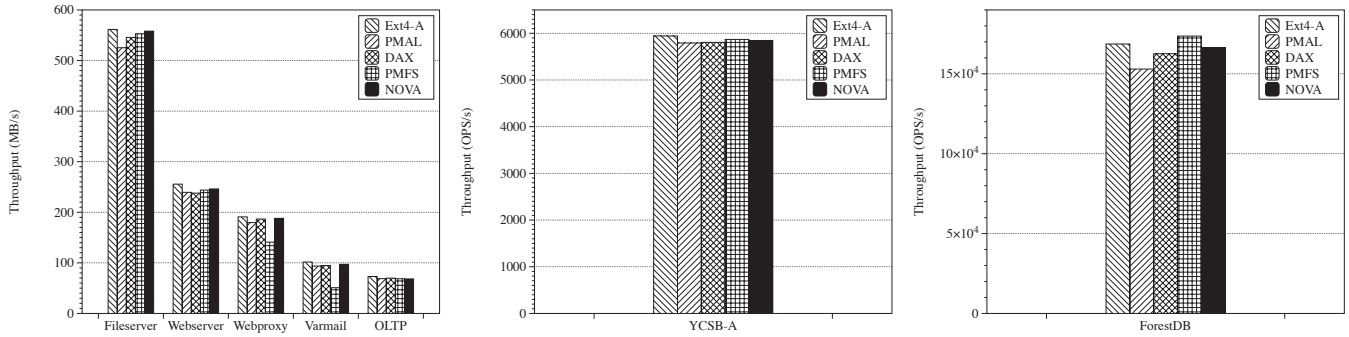
We also use two key-value store benchmarks for our experiments. YCSB is an open source benchmark program suite provided by Yahoo generally used to evaluate NoSQL database systems [43]. Among these we make use of workload A, denoted YCSB-A, which is a write heavy workload (50/50 reads and writes) that represents applications such as a session store recording recent actions. (Though we experimented with other YCSB workloads, we present only workload A as the results are more or less similar.) The other benchmark is ForestDB-Benchmark, denoted ForestDB for brevity, which is a realistic key-value benchmark introduced by Couchbase [44]. Table IV summarizes the characteristics of these two benchmarks.

B. Overall Performance

The overall performance results are given in Figure 4, where the *x*-axis represents the workload used, while the *y*-axis is the throughput. We make a few observations based on the results.

First, we see that Ext4-A indeed shows the best performance. However, we also see that PMAL is not far behind. Also, PMAL performs in par with other PM-dedicated file systems even though PMAL journaling is in journal mode. The PMAL column of Table V, whose values are obtained in a way similar to Table II, shows the approximate time spent in each of the components. The main observation is that the PMAL component, which performs journal mode journaling, does not incur extensive overhead compared to the 'Page cache flush' and 'FS consistency' components of Ext4 (Sync) in Table II.

The second observation is that PMFS does considerably worse than other comparisons for the Webproxy and Varmail workloads. The Webproxy and Varmail workloads create a large number of files and directories, which strongly affect PMFS performance. This is due to the heavy metadata indexing structure construction required for PMFS. This is supported by the results shown in Figure 5, which shows the performance for the Fileserver workload as the number of files is increased, while the mean file size is set so that the total footprint is a constant 20GBs. We see here that as the number of files grows, PMFS performance drops considerably performing far worse than PMAL.



(a) Filebench macro benchmarks (b) YCSB-A benchmark (c) ForestDB benchmark

Fig. 4: Overall performance of various benchmarks

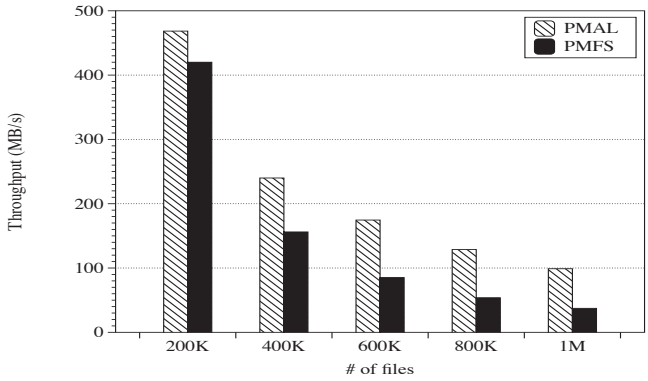


Fig. 5: Performance comparison of PMAL and PMFS for the Fileserver workload as the number of files managed is increased

Overall, we see that PMAL is comparable to other PM-dedicated file systems. The performance of PMAL is only slightly below those of the ideal Ext4-A case, where all writes are being asynchronously written within the page cache. This is consistent throughout all workloads. Our results show that the lean implementation of PMAL allows us to attain the goal we set out to achieve.

C. Journaling Effects on Performance

In this section, we consider a variation of both PMAL and DAX. Recall that PMAL journals in journal mode, which is typically shunned in disk-based storage systems due to their low performance. Here, we also consider the effect of relaxing journaling to ordered mode for PMAL, which we refer to as PMAL-O (for ordered mode). PMAL-O can be implemented by simply not performing PMAL Flush on the metadata so that the metadata is not written into the metadata portion of the file system until checkpointing time. For a variation of DAX, which does asynchronous metadata commits, we consider synchronously committing the metadata into the journal. This sacrifices performance, but allows for stronger consistency than the current version of DAX, and we refer to this version as DAX-S (for synchronous).

The performance effect of these changes in the I/O flow is shown in the right two PMAL-O and DAX-S columns of Table V. We see that for PMAL-O, the time consumed at the PMAL component is reduced, while for DAX-S, the time

TABLE V: Running time (ns) of components in `write()` I/O execution flow of PMAL and DAX obtained similarly to Table II

	PMAL	DAX	PMAL-O	DAX-S
System call gate	298	211	313	288
VFS objects	815	899	916	957
I/O type switch	3,208	2,223	3,037	2,892
Page cache	15,781		15,600	
Memory I/O	481		490	
PMAL	17,280		13,373	
FS consistency		7,115		98,358
DAX		13,213		14,813
Elapsed Time	37,863	23,661	33,729	117,308

consumed at the FS Consistency component increases considerably. The macro effect of these changes on the performance of the benchmarks are shown in Figure 6 and discussed below.

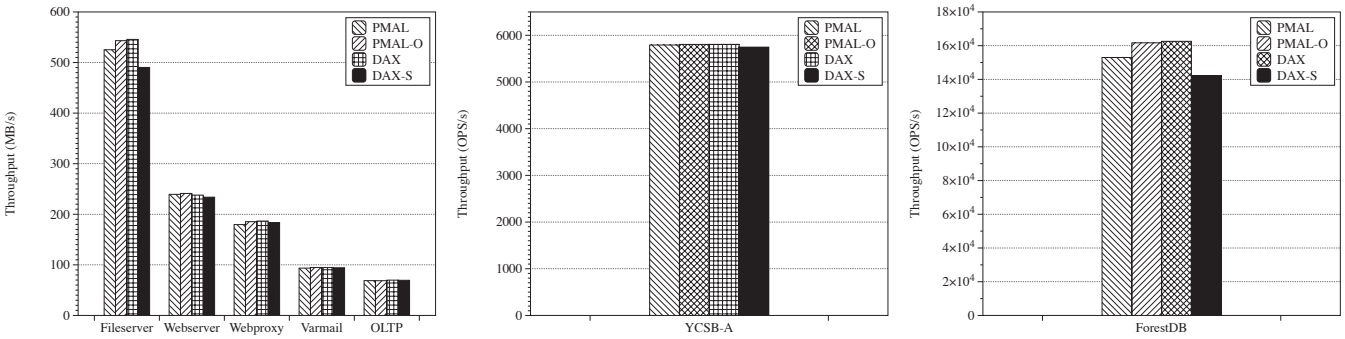
As expected, the performance of the benchmarks with PMAL-O does better than with PMAL and DAX-S does worse than DAX. Overall, PMAL-O does consistently better than the others, while DAX-S is consistently worse than the others. However, the gains by PMAL-O over PMAL is smaller than the loss by DAX-S over DAX.

The conclusion from these results is that given the same consistency level (PMAL-O and DAX-S), the performance of PMAL is consistency better. Also, PMAL is the only mechanism that can support journal mode journaling and even with this higher level of consistency guarantee, performance is in par with DAX.

D. Page Cache as the Focal Point

Legacy file systems retain years of understanding and maturity. In Linux file systems, the page cache, through which key data related activities such as mirroring, compression, and encryption occur, plays a vital role. While these features are constantly being enhanced, new features are also being added. At the center of these activities, the page cache plays a central role. On the other hand, approaches such as DAX, PMFS, or NOVA that avoid the page cache altogether can be regarded as a new PM focused feature rather than an integration into the existing file system even though it is implemented within a particular file system.

In this section, we take a concrete example, in particular, the `copy_file_range()` system call [28]–[31]



(a) Filebench macro benchmarks (b) YCSB-A benchmark (c) ForestDB benchmark

Fig. 6: Performance comparison of PMAL and DAX for different consistency guarantees

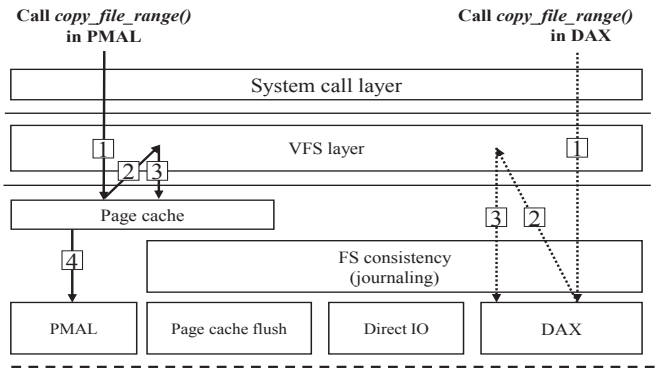


Fig. 7: Call flow of `copy_file_range()` in PMAL and DAX where boxed numbers represent the execution steps

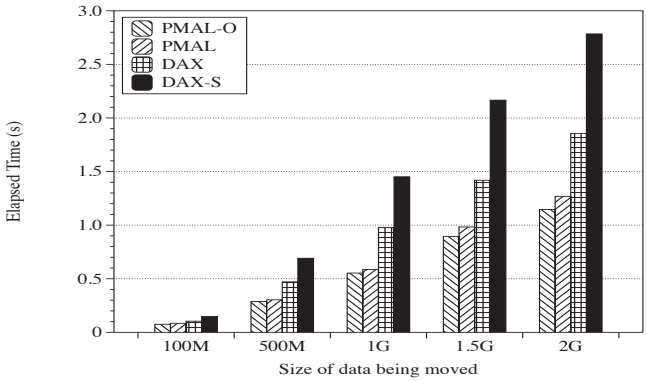


Fig. 8: Function `copy_file_range()` performance according to whether or not page cache is used

and observe the effect of PMAL in comparison to DAX. `copy_file_range()` is a system call that has been added to the Linux kernel since version 4.5, whose role is to optimize the performance of data passing between files through a single user/kernel mode change unlike traditional data passing techniques. This is done by performing data passing through asynchronous page cache reads and writes in the kernel. This process is depicted on the PMAL (left) side of Figure 7.

In contrast to PMAL, DAX must take a different route in servicing `copy_file_range()` as it cannot make use of the page cache. Hence, for DAX, this system call goes through the DAX I/O layers as shown on the DAX (right) side of Figure 7. This incurs considerable extra overhead compared to PMAL, in particular, for Steps 1 and 2 (denoted by the boxed numbers), which are file read requests. Moreover, the fact that there needs to be two different implementation flows within the same Ext4 framework disperses the efforts that can be put into kernel development.

To see the quantitative effects of the two I/O flows, we measure the elapsed time of the `copy_file_range()` system call for various file sizes. The Linux kernel version 4.7, with PMAL ported, is used for these measurements. The results, depicted in Figure 8 where the *y*-axis is the elapsed time, show that PMAL, which makes use of the page cache, performs considerably better than DAX. While we cannot say that PMAL will bring about these kinds of large performance benefits for all cases, we can say that through the PMAL approach, we should be able to directly and easily transfer current and future traditional storage related kernel development efforts into future PM devices.

E. PMAL in Hybrid Memory Environment

As mentioned previously, PMAL can be implemented in a hybrid memory environment by maintaining an extra multi-versioning area like the journal in PM. We also implement and conduct experiments with such a scheme, but do not show the results due to space limitations. Overall, however, we find that there is roughly a 10% performance degradation compared to the PM-only version, which is mainly due to the journal commit and checkpoint overhead.

VI. Summary and Conclusions

New memory technologies such as PCM, STT-MRAM and 3D XPoint are expected to make an impact on all levels of computing. These new memory, which we refer to as Persistent Memory (PM), boasts performance in par with DRAM while providing nonvolatility. Recently, there have been numerous efforts to develop storage systems targeted specifically for PM storage [7]–[12], [15], [16]. This is based on the premise that legacy file systems are sub-optimal on memory bus attached PM storage. However, many years of time and effort are ingrained in legacy file systems that are now mature and time-tested, and making use of them seems a logical choice.

In this paper, we considered making use of legacy file systems for PM storage so that we can leverage the maturity ingrained in legacy file systems while, at the same time, reaping the high performance offered by PM. Through empirical evaluation, we analyzed the workings of legacy file systems, in particular, the Ext4, and PM-dedicated file systems, DAX, PMFS and NOVA. Based on these analyses, we developed the Persistent Memory Adaptation Layer (PMAL) module that allows us to attain our goal of retaining maturity as well as high performance. PMAL is lightweight comprising roughly 180 lines of code in total and can easily be integrated into legacy file systems. Using Filebench and key-value store benchmarks we showed that performance of PMAL integrated Ext4 performs in par with PM-dedicated file systems. This is in spite of the fact that PMAL integrated Ext4 provides stronger consistency guarantees.

Finally, and more importantly, while the approach of PM-dedicated file systems separates traditional storage related kernel development efforts from those for PM, the approach that PMAL takes allows kernel developers to concentrate on the outcome that have matured with time and that are available today. Furthermore, future development efforts also need not take a two-track approach, one for traditional storage and another of PM storage, but can be concentrated to one concerted effort.

ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their construction comments. This work was supported by Samsung Research Funding Centre of Samsung Electronics under Project Number SRFC-IT1402-09.

REFERENCES

[1] S. Fujita, K. Nomura, H. Noguchi, S. Takeda, and K. Abe, "Novel Nonvolatile Memory Hierarchies to Realize "Normally-Off Mobile Processors"," in *Proc. ASP-DAC*, 2014.

[2] M. S. Haque, A. Li, A. Kumar, and Q. Wei, "Accelerating Non-Volatile/Hybrid Processor Cache Design Space Exploration for Application Specific Embedded Systems," in *Proc. ASP-DAC*, 2015.

[3] A. Jog, A. K. Mishra, C. Xu, Y. Xie, V. Narayanan, R. Iyer, and C. R. Das, "Cache Revive: Architecting Volatile STT-RAM Caches for Enhanced Performance in CMPs," in *Proc. DAC*, 2012.

[4] 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 *Proc. MICRO*, 2013.

[5] D. Narayanan and O. Hodson, "Whole-System Persistence," in *Proc. ASPLOS*, 2012.

[6] I. H. Doh, Y. J. Kim, E. Kim, J. Choi, D. Lee, and S. H. Noh, "Towards Greener Data Centers with Storage Class Memory," *Future Generation Computer Systems*, 2013.

[7] 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 *Proc. SOSP*, 2009.

[8] X. Wu and A. Reddy, "SCMFS: A File System for Storage Class Memory," in *Proc. SC*, 2011.

[9] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson, "System Software for Persistent Memory," in *Proc. EuroSys*, 2014.

[10] H. Volos, S. Nalli, S. Panneerselvam, V. Varadarajan, P. Saxena, and M. M. Swift, "Aerie: Flexible File-System Interfaces to Storage-Class Memory," in *Proc. EuroSys*, 2014.

[11] J. Xu and S. Swanson, "NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories," in *Proc. FAST*, 2016.

[12] J. Ou, J. Shu, and Y. Lu, "A High Performance File System for Non-Volatile Main Memory," in *Proc. EuroSys*, 2016.

[13] 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 *Proc. ASPLOS*, 2011.

[14] H. Volos, A. J. Tack, and M. M. Swift, "Mnemosyne: Lightweight persistent memory," in *Proc ASPLOS*, 2011.

[15] T. Hwang, J. Jung, and Y. Won, "HEAPO: Heap-Based Persistent Object Store," *ACM Transactions on Storage (TOS)*, 2014.

[16] Y. Zhang, J. Yang, A. Memaripour, and S. Swanson, "Mojim: A Reliable and Highly-Available Non-Volatile Memory System," in *Proc. ASPLOS*, 2015.

[17] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, T. C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S. H. Chen, H. L. Lung, and C. H. Lam, "Phase-Change Random Access Memory: A Scalable Technology," *IBM Journal of Research and Development*, 2008.

[18] M. Jung, J. Shalf, and M. Kandemir, "Design of a Large-Scale Storage-Class RRAM System," in *Proc. ICS*, 2013.

[19] E. Kultursay, M. Kandemir, A. Sivasubramaniam, and O. a. Mutlu, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative," in *Proc. ISPASS*, 2013.

[20] Micron Technology, "3D XPoint Technology," http://www.micron.com/about/innovations/3d-xpoint-technology.

[21] AnandTech, "Intel's 140GB Optane 3D XPoint PCIe SSD Spotted at IDF," http://www.anandtech.com/show/10604/intels-140gb-optane-3d-xpoint-pcie-ssd-spotted-at-idf.

[22] J. Handy, "Understanding the Intel/Micron 3D XPoint Memory," in *Proc. SDC*, 2015.

[23] E. J. O'Sullivan, M. J. Gajek, J. J. Nowak, S. L. Brown, M. C. Gaidis, G. Hu, J. Z. Sun, P. L. Trouilloud, D. W. Abraham, R. P. Robertazzi *et al.*, "Recent Developments in ST-MRAM, Including Scaling," *224th ECS Meeting Abstract*, 2013.

[24] J. Corbet, "Supporting filesystems in persistent memory," https://lwn.net/Articles/610174/.

[25] M. Wilcox, "DAX: Page cache bypass for filesystems on memory storage," http://lwn.net/Articles/618064/.

[26] ——, "Support ext4 on nv-dimms," https://lwn.net/Articles/588218/.

[27] Phoronix, "Xfs will get dax support in the linux 4.2 kernel," https://www.phoronix.com/scan.php?page=news_item&px=XFS-Linux-4.2-DAX-And-More.

[28] LWN.net, "Directly mapped persistent memory page cache," http://lwn.net/Articles/644120/.

[29] Anna Schumaker, "vfs:Add vfs_copy_file_range() support for pagecache copies," http://www.spinics.net/lists/linux-btrfs/msg47738.html.

[30] LWN.net, "copy_file_range()," https://lwn.net/Articles/659523/.

[31] ——, "VFS:In-kernel copy system call," https://lwn.net/Articles/663746/.

[32] Kernel Documentation, "Ext4 Filesystem," https://www.kernel.org/doc/Documentation/filesystems/ext4.txt.

[33] Persistent Memory Wiki, "Persistent memory," https://nvdimm.wiki.kernel.org/.

[34] GitHub, "Porting pmfs to the latest linux kernel," https://github.com/Andiry/pmfs-3.19.

[35] ——, "Nova: Non-volatile memory accelerated log-structured file system," https://github.com/Andiry/nova.

[36] J. Corbet, "The High-Resolution Timer API," https://lwn.net/Articles/167897/.

[37] E. Lee, H. Bahn, S. Yoo, and S. H. Noh, "Empirical Study of NVRAM Storage: An Operating System's Perspective and Implications," in *Proc. MASCOTS*, 2014.

[38] E. Lee, H. Bahn, and S. H. Noh, "Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory," in *Proc. FAST*, 2013.

[39] ——, "A Unified Buffer Cache Architecture that Subsumes Journaling Functionality via Nonvolatile Memory," *ACM Transactions of Storage (TOS)*, 2014.

[40] S. Pelley, P. M. Chen, and T. F. Wenisch, "Memory Persistency," in *Proc. ISCA*, 2014.

[41] A. Rudoff, "In a world with persistent memory," 6th Annual Non-Volatile Memories Workshop (NVMW), 2015.

[42] Sourceforge, "Filebench," http://filebench.sourceforge.net/wiki/index.php/Main_Page.

[43] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking Cloud Serving Systems with YCSB," in *Proc. SoCC*, 2010.

[44] GitHub, "ForestDB-Benchmark," https://github.com/couchbaselabs/Forest-DB-Benchmark.