A Protected Block Device for Persistent Memory

Feng Chen
Louisiana State University
fchen@csc.lsu.edu

Michael P. Mesnier
Intel Labs
michael.mesnier@intel.com

Scott Hahn
Intel Labs
scott.hahn@intel.com

***Abstract*—Persistent Memory (PM) technologies, such as Phase Change Memory, STT-RAM, and memristors, are receiving increasingly high interest in academia and industry. PM provides many attractive features, such as DRAM-like speed and storage-like persistence. Yet, because it draws a blurry line between memory and storage, neither a memory- or storage-based model is a natural fit. Best integrating PM into existing systems has become challenging and is now a top priority for many. In this paper we share our initial approach to integrating PM into computer systems, with minimal impact to the core operating system. By adopting a hybrid storage model, all of our changes are confined to a block storage driver, called *PMBD*, which directly accesses PM attached to the memory bus and exposes a logical block I/O interface to users. We explore the design space by examining a variety of options to achieve performance, protection from stray writes, ordered persistence, and compatibility for legacy file systems and applications. All told, we find that by using a combination of existing OS mechanisms (per-core page table mappings, non-temporal store instructions, memory fences, and I/O barriers), we are able to achieve each of these goals with small performance overhead for both micro-benchmarks and real world applications (e.g., file server and database workloads). Our experience suggests that determining the right combination of existing platform and OS mechanisms is a non-trivial exercise. In this paper, we share both our failed and successful attempts. The final solution that we propose represents an evolution of our initial approach. We have also open-sourced our software prototype with all attempted design options to encourage further research in this area.**

***Index Terms*—Memory, Storage, Persistent memory, Operating system, Device driver**

I. Introduction

Over the years researchers in academia and industry have expended tremendous effort in the long battle with slowly improving storage performance. Countless innovations have been made in almost every corner of computer systems to mitigate the so-called "performance gap" between *volatile* memory and *persistent* storage. Although these innovations have greatly improved storage and I/O performance, the performance gap between delivering data and processing data never stops widening, because it essentially stems from the mechanical nature of hard drives and cannot be completely removed.

A. Persistent Memory

Semiconductor storage (e.g., NAND flash) has begun to change this situation. Beyond flash memory, recent technology breakthroughs may offer us even more relief in this multi-decade struggle. Persistent Memories (PM), such as Phase Change Memory (PCM) [22], STT-RAM [10], and memristors [14], promise many desirable features – access speeds comparable to DRAM, storage-like persistence, reasonably high data endurance and retention, low power consumption, and byte addressability. Table I shows details on the performance and endurance characteristics of PM.

These revolutionary technologies exhibit radically different characteristics, compared to any prior memory and storage technology, and thus raise many critical challenges to computer system designers. A fundamental question must be first answered – what is the appropriate usage model for PM?

	Read	Write	Endurance
PCM	50-85ns	150ns-1µs	10^8-10^{12}
Memristor	100ns	100ns	10^8
STT-RAM	6ns	13ns	10^{15}
DRAM	60ns	60ns	$> 10^{16}$
NAND flash	25 µs	200-500µs	$10^4 - 10^5$

TABLE I
Characteristics of PM technologies

As a new technology, PM draws a blurry line between volatile memory and persistent storage. It cannot be viewed and used simply as non-volatile memory or fast storage. A naive integration of PM into computer systems would introduce either protection or performance problems. In particular, using PM as a byte-addressable memory device opens it up to a wide variety of corruption, such as stray writes in the OS. Yet, using PM behind the protection of an I/O controller leaves considerable performance on the table.

Of course, one approach is to completely redesign the entire system, such as merging memory and storage systems together in the OS. For example, the application programming model can be changed to be aware of volatile and persistent memory and differentiate persistent objects from volatile ones [7], [26]. Also, new file systems could be created for PM to leverage its byte addressability [8], [11], [27].

Unfortunately, such changes are non-trivial in practice. The existing system hierarchy is built on many long-standing assumptions, which have existed explicitly or implicitly for decades – memory is fast, volatile, byte addressable, while storage is slow, persistent, and block addressable. This commonsense understanding forms the foundation of today's computer system architecture but does not readily apply to PM. Directly integrating PM into the computer system requires that we develop a new understanding of the roles of memory and storage. At the same time, we must be mindful of the large amount of intellectual property associated with legacy systems, and the fact that any radical (business-disruptive) changes will

978-1-4799-5671-5/14/$31.00 © 2014 IEEE Published by the IEEE Computer Society

receive strong resistance in industry. Further, convincing users, especially commercial users, to change their heavily tuned systems or rewrite complex applications (e.g., database) is very difficult in practice.

For these reasons, we believe a complete system redesign would be, at least initially, difficult to gain significant traction in practice — and not always necessary, as shown later. From the perspective of cost efficiency, especially for general-purpose systems, our goal is to strike an appropriate balance between system redesign and exploiting the unique features of PM. That is, our design philosophy is to apply an *evolutionary* approach to this revolutionary technology and minimize disruptive changes.

In the rest of this paper, we share our experiences in building a fast, protected, and persistent block storage based on PM. We study a variety of design options and identify the most suitable ones to effectively achieve these goals. We hope the results presented in this paper encourage discussions about other PM usage models.

B. Our Contributions

Several contributions are made in this paper: (1) We present a hybrid model to combine the advantages of a memory-based model and a storage-based model, while making no changes to the core OS and its applications. (2) We implement a prototype of the hybrid model as a kernel module in Linux 2.6.34 and explore a variety of designs to realize performance, protection, persistence, and compatibility goals; we present the advantages and disadvantages of each approach. (3) We show that with existing platform mechanisms (private page table mappings, non-temporal store instructions, memory fences, and OS I/O barriers), both protection and ordered persistence can be effectively achieved with relatively small performance overhead. We also show that, even when compared to the highly efficient RAM-based file systems (tmpfs, ramfs), our solution can provide good performance. (4) We have open-sourced our prototype. Under the GPLv2 license, the source code is available for public downloading [2]. We encourage researchers and developers to take advantage of this software for further research on PM.

A primary contribution of this paper is demonstrating how existing OS mechanisms can be used in bringing memory-bus-attached PM as block devices to the market. Though obvious in hindsight, determining the right combination (e.g., uncachable vs. non-temporal stores, private-mappings vs. page-based protection, etc.) proved to be a non-trivial exercise. To this end, we also share our less successful attempts in integrating PM into the platform, as they each helped lead to the final solution.

The rest of this paper is organized as follows. Section II covers the background of PM and our storage driver model. Section III describes our experimental platform. Section IV investigates various block driver design options, based on the hybrid model approach. Section V presents our performance evaluation. Section VI introduces the related work. Section VII gives additional discussions and introduces the future work.

II. PM Usage Models

Although the usage model of PM has only indirectly been discussed in prior research, many studies [6], [7], [11], [25], [26], [27] have implicitly assumed at least two basic models:

- **Memory-based Model** – PM is integrated as part of the memory system to displace DRAM memory entirely or partially. A memory controller manages PM and connects CPU to PM via a high-speed memory bus [16], [21], [28]. The OS sees PM as memory, which may be marked as non-volatile by BIOS [27]. Aside from persistence, PM is regarded similarly to DRAM and provides byte addressability to the OS and its applications. Data stored in PM is accessible through `load` and `store` instructions.
- **Storage-based model** – PM is simply used as a faster medium to displace NAND flash and encapsulated in the same form as flash SSDs. An I/O controller manages PM and connects to the host. I/O commands and data are transferred via the I/O bus (e.g., SATA or PCI-E) between the device and the host. Data stored in PM is accessible in units of sectors (512 bytes) via the block I/O interface (`read` and `write` commands).

A. Memory-based Model vs. Storage-based Model

Both usage models have their advantages and disadvantages:

- **Performance** – The memory-based model can provide higher performance than the storage-based model. Even compared with high-speed PCI-E bus, main memory bus throughput and latency both are noticeably better. Further considering the internal overhead of I/O controllers, such as interface command decoding and ECC, the storage-based model cannot provide sufficient headroom for PM, whose speed is close to DRAM.
- **Protection** – The memory-based model has much greater risk of data corruption than the storage model. If we directly map PM, which is attached to the memory bus, into an address space (user or kernel), it will be subjected to the types of data corruption typically caused by stray writes (bad memory pointers). Leaving PM exposed to bugs in the kernel code (e.g., device drivers [5]) can easily corrupt a large amount of persistent data, which often leads to catastrophic results.
- **Ordered persistence** – An issue related with the memory-based model is that data written to PM is subject to CPU caching effects, as applications usually store data in volatile CPU caches for performance. As such, a power failure could cause data loss. In contrast, the storage model provides a mechanism (write barriers) to safely persist data.
- **Compatibility** – The storage-based model provides the best compatibility to existing systems and can be used as a 'drop-in' solution. POSIX I/O can be used as it is today, atop a legacy block-based interface. In contrast, the memory-based model requires non-trivial system and application changes in order to exploit the byte addressability of PM.

B. A Hybrid Model

We propose a *hybrid* model. In this model, we separate the physical and logical architectural designs, as shown in Figure 1: (1) **Physical architecture** – similar to the memory model, PM DIMMs are physically attached to the high-speed memory bus and managed by a memory controller. (2) **Logical architecture** – PM is exposed as a block device in the OS. However, relative to the conventional storage-based model, there is no hardware storage controller (e.g., SAS, SATA, or PCI-E). Data accesses are performed through a `read`/`write` block I/O interface and then converted to memory `load`/`store` instructions by a driver in the OS.

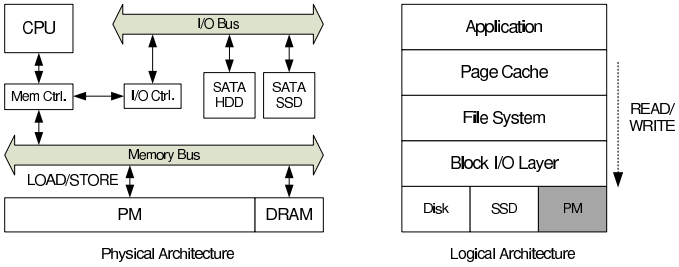


Fig. 1. A hybrid usage model of PM

Such a hybrid design provides several benefits: (1) **High performance** – PM is directly attached to the memory bus, which provides high bandwidth and low latency. (2) **Protection** – We only allow accesses to PM via the `read`/`write` interface. Any unauthorized direct access to PM can be detected and prevented (to be described later). As such, the potential data corruption risk caused by stray writes in the kernel code is minimized. (3) **Ordered persistence** – Since accesses to PM are contained in a single entity (an OS driver), we can enforce ordered persistence through conventional OS write barriers. (4) **Backwards compatibility** – PM is exposed to the rest of the OS kernel in the form of a regular block device, thereby supporting existing OSes and applications. A limitation of the hybrid model is the loss of byte addressability. However, considering that byte addressability demands fundamental code changes to applications, we choose backward compatibility as our top design priority.

In our design, this block device is implemented as a standalone kernel module, called *PMBD*, which requires no change to any other system components. A device driver is loaded and responsible for converting `read`/`write` commands into `load`/`store` instructions. One may note that this design shares the same principle as the RamDisk [24]. However, unlike a RamDisk, our block driver separates page cache and PM and is designed particularly for using PM as *persistent storage*, rather than *volatile memory*. Special considerations, especially protection and ordered persistence, must be carefully explored in the design.

III. Experimental Environment

We first introduce our experimental system and the workloads used in this study. Unless otherwise noted, all the experiments are conducted on the following system setup.

A. System Setup

Our experimental system is a storage server running Fedora Core 14 with Linux kernel 2.6.34 and the Ext4 file system. It has two Intel Xeon X5680 3.3GHz processors, each with 6 cores. In our experiments, we use a 1TB 7,200RPM Seagate hard drive and another 64GB Intel X25-E SLC SSD as the target devices to compare with PM. Another 1TB Seagate hard drive is used as the system disk. All the storage devices are connected through the on-board SAS connectors.

B. PM emulation

As PM devices are unavailable yet, we use DRAM for experiments. We assume that in future systems BIOS will expose PM DIMMs as contiguous physical memory, labeled as non-volatile to the host OS. We also assume hardware handles wear-leveling. We emulate such an architecture by changing the e820 table to reserve the high 16GB of memory space as PM, and the rest as DRAM. Similar to prior work [8], [27], we study the design options with raw DRAM. In Section V-C, we emulate various PM speeds by inserting extra operations to poll the timestamp register (`tsc`), which is similar to [26], and study its end-to-end impact to application performance. To reduce variance, we disable NUMA, HyperThreading, SpeedStep, and lock the memory bus operation speed at 1600MHz.

C. Workloads

We use two sets of workloads. Each workload runs for three iterations, and we show both average values and error ranges.

- **Micro-benchmark** – We use the Intel Open Storage Toolkit [1] to generate various types of micro-benchmark workloads. It can produce I/O workloads with different configurations, such as read/write ratio, random/sequential ratio, request size, and queue depth (the number of concurrent requests). It reports bandwidth, IOPS, and latency.
- **Macro-benchmarks** – We use 8 different workloads as listed in Table II. Five workloads are read intensive, including *sfs*, *tpch*, *glimpseindex*, and *clamav*. TPC-C (*tpcc*) is a database online transaction processing (OLTP) workload and the most write-intensive (63.7%) one. The other two workloads, *tar* and *untar*, have roughly the same amount of reads and writes. We use execution time as the main performance metric. For *tpcc*, which reports the throughput (new order transactions per minute), we convert it to the number of seconds to complete 1 million transactions.

IV. PM Storage Design

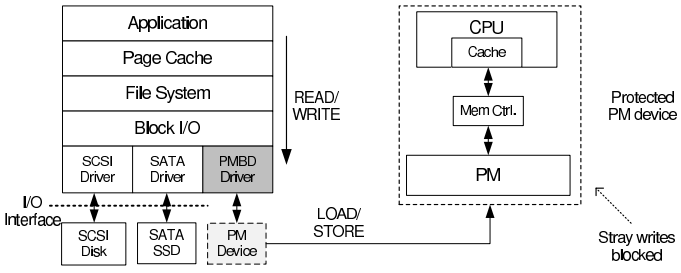


Fig. 2. PM block device architecture

Name	Write	Dataset (MBs)	Amount (MBs)	Description
sfs	7.4%	11,210	146,674	SPECsfs 2008: 10K files, 500K trans., 1K dir.
tpch	9.7%	10,869	78,126	TPC-H (OLAP DB): Scale Factor 4, PostgreSQL 9
tpcc	63.7%	11,298	98-419K	TPC-C (OLTP DB): 80 WH, 20 conn., 60 sec.
tar	46.9%	11,949	11,493	compressing a kernel code tar ball
untar	52.2%	11,970	11,413	uncompressing a kernel code tar ball
devel	38%	2,033	3,470	software development: untar, patch, tar, diff.
glimpseindex	5.5%	12,504	6,019	text index engine: index 12GB Linux kernel code
clamav	0.3%	14,495	5,270	virus scanning: 14GB files generated by sfs

TABLE II
MACRO-BENCHMARK WORKLOAD DESCRIPTION

In our design (Figure 2), the OS views PM as contiguous memory. The PM block driver performs the following tasks: (1) managing the PM space and providing a block device interface for the upper-level components, such as file systems and applications, to access PM via the standard block I/O interface (`read`/`write`), (2) translating incoming `read`/`write` requests into `load`/`store` instructions to access data in PM, (3) providing write protection to prevent PM from being accessed by stray writes (i.e., attempts to modify PM data without going through the PM block device interface), and (4) enforcing ordered persistence through non-temporal store instructions and write barriers.

A. Device Driver Overview

We have implemented the PM block driver (PMBD) as an OS kernel module in Linux 2.6.34. It consists of about 5,000 lines of code with comments. After being loaded into the OS, the driver creates a virtual block device, which provides a standard block I/O interface. From the perspective of other system components and applications, the PM device is no different than any physical block device. Users can create partitions and file systems on it, providing compatibility for legacy applications.

PM is exposed to the OS as a contiguous range of physical memory marked as non-volatile. The PMBD driver is responsible for mapping the PM physical pages (4KB each) into the kernel virtual address space to make it accessible. Upon receiving a request, the driver first computes the physical address of the demanded PM page and then translates `read`/`write` into `load`/`store` instructions. All I/Os are handled synchronously (i.e., no context switch or queuing).

Besides managing PM pages and providing a block I/O interface, the PMBD driver also provides protection from stray writes and ordered persistence. In the following, we will explore a variety of design options to achieve the protection goals without sacrificing performance.

B. Protection from Stray Writes

A fundamental distinction between volatile memory and PM is the requirement for the protection of data. Once written into PM, data is assumed to be *persistent*, as we assume for a hard drive. The challenge is that since PM is physically managed like DRAM and exposed to the OS as memory, reading or writing data requires PM to be mapped into the kernel virtual memory address space, which is shared among all kernel processes. A wild pointer in kernel-level code (e.g., a buggy device driver) can quickly corrupt a large amount of PM data. An example is the famous hardware-destroying bug in Linux 2.6.27-rc8, which overwrites the non-volatile memory on e1000e network adapters [9]. Although it is possible that data could also be corrupted in memory and eventually materialized into storage, it is notably more difficult to pass the sanity checks in multiple layers – page cache, file systems, generic block layer, device I/O, etc. We believe that the same level of protection as a block device must be provided for PM. Also note that our focus is not to prevent malicious attacks but kernel bugs. We have examined several options to provide strong and low-overhead protection as follows.

1) Page Table based Protection: In modern operating systems, *paging* is used to separate the address spaces of processes and share limited physical memory. With paging, physical memory is segmented into multiple fixed-sized frames (4KB, typically). Each process is associated with a *page table* (PT), which is a multi-level tree-like structure translating process-viewable virtual memory addresses to physical memory addresses. The OS is responsible for constructing the page table, allocating and managing physical page frames. The hardware is responsible for automatically translating virtual addresses to physical addresses via the page table or a translation lookaside buffer (TLB) in CPU. The page table is also used for access control. In the last level of the page table, the page table entry (PTE) contains a set of flags. The `R/W` flag controls whether the page is read-only or writable. If the `R/W` flag is 0, the page is read-only, and writing to such a page would be blocked and trigger a page fault.

Based on this page table mechanism, protection from stray writes can be implemented as follows. (1) When the driver is loaded, PM pages are mapped into the kernel virtual address space using `ioremap()` and initialized as read-only by disabling the `R/W` bit. (2) Upon a read, no additional operation is needed. (3) Upon a write, the `R/W` bit is enabled to set the page as writable and perform the write operation, and then disabled to set it back to read-only. (4) When the driver is unloaded, the PM pages are unmapped.

This solution is simple. However, it incurs high performance overhead. To demonstrate, we use the Intel Open Storage Toolkit [1] to generate eight write-only workloads. All workloads use direct I/O to access the PM device (i.e., no page cache) and run for 30 seconds. Figure 3 compares the write

bandwidths of running the micro-benchmarks with and without this protection mechanism, which are denoted as *PT-Naive* and *Baseline* respectively. In the figure, RD and WR denote reads and writes, SEQ and RND denote access patterns (sequential or random), sz4 and sz256 denote the request size (4KB and 256KB), and Q1 and Q32 denote the queue depth (1 or 32 outstanding requests). We can see that write bandwidth is severely reduced by a factor of 18, from 14.8GB/sec to only 808MB/sec.

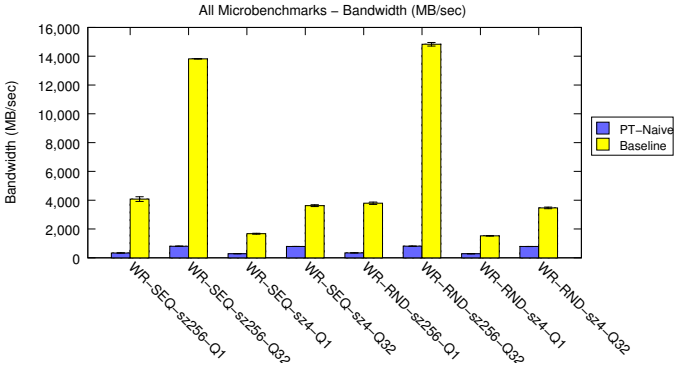


Fig. 3. Performance of page-table based protection

The performance degradation is due to two factors. (1) When writing a page, we need to perform two PTE bit changes (from read-only to writable, and then flip it back). Unfortunately, each PTE attribute change incurs a costly 'TLB shootdown.' Because there is no coherence protocol between TLBs, after a page table entry changes, a high-cost Inter-processor Interrupt (IPI) is needed to flush the TLB entries in all processors. (2) The OS treats changing page table attributes as an infrequent operation, and it is less optimized. For example, the TLB flush happens inside a system-level lock to prevent other processors with stale TLB entries from changing page attributes in parallel, which further lowers the performance.

This protection mechanism, though it can prevent stray writes, violates our performance goal. Although the achievable peak bandwidth is still much higher than most SSDs and HDDs, it only exploits a tiny fraction (5.4%) of its performance potential.

Optimization 1: Protection with Buffer/Batching – Buffering and batching can reduce the cost of manipulating the PTE flags and improve performance, for two reasons. Firstly, a small DRAM space can be used as a temporary buffer to absorb a burst of writes, which acts as an on-device buffer in a hard drive. Secondly, and more importantly, the DRAM buffer can be a staging ground to reorganize pages before updating the page table. Since only one TLB shootdown is needed for updating the `R/W` flags of a sequence of contiguous pages, we can effectively batch up the PTE flag changes and amortize the related overhead.

Buffering – As shown in Figure 4, a circular buffer is used by the PMBD driver to temporarily buffer dirty pages. Upon a write, the dirty page is placed in the buffer. Two pointers, p_{start} and p_{end}, track the first and the last dirty page in the buffer, which segments the buffer into the allocated space and

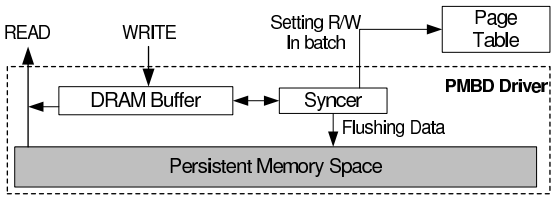


Fig. 4. An illustration of buffering and batching

the free space. We keep track of the number of dirty pages in the buffer. If it exceeds a high watermark, a syncer daemon thread (`nsync`) is woken up to flush the buffer.

Batching – A syncer daemon in the driver can flush the dirty pages in a batch: (1) sort the dirty pages in the order of their virtual memory addresses of the corresponding PM pages; (2) pack contiguous pages into one batch; (3) switch the `R/W` flag of pages in the sequence to be writable; (4) write pages to PM; and (5) switch the PTE flag back to read-only. This process repeats for multiple iterations until all the pages are written to PM.

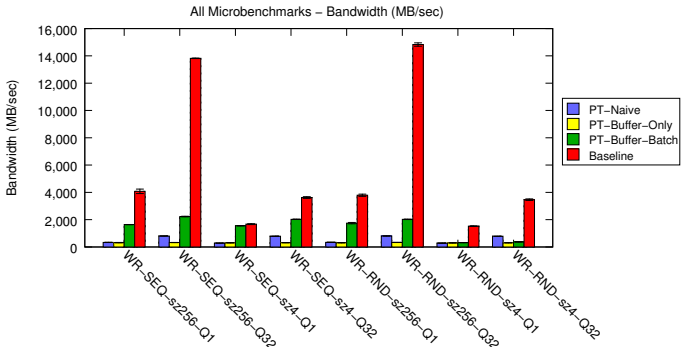


Fig. 5. Performance of buffering and batching

Figure 5 shows the performance comparison between the naive page table based write protection (PT-Naive), the protection with a 16MB buffer (PT-Buffer-Only), the page table protection with buffering and batching (PT-Buffer-Batch), and no protection (Baseline). We have three observations. (1) With buffering only, write performance cannot be improved, sometimes it even gets worse, especially with many concurrent I/O requests. For example, the performance of sequential writes of 4KB with 32 jobs drops from 789 MB/sec to 315 MB/sec. This is because these workloads have no locality and thus benefit little from the buffer, and in the meantime, buffer allocation and flushing become a bottleneck limiting the scalability of handling parallel writes. Without batching, the benefits received from buffering cannot offset this negative impact. (2) With both buffering and batching, sequential write performance can be significantly improved due to the effectively amortized TLB shootdown cost. For example, the performance of sequential writes of 4KB with 32 jobs is improved by 2.5 times to 2028 MB/sec. (3) Random writes receive little performance benefit, as expected, since there is no opportunity for amortizing the cost at all. In fact, with a high queue depth, buffering and batching degrade random write performance. For example, random writes of 4KB with 32 jobs degrade by 2.1 times, from 786 MB/sec to 374 MB/sec.

This case shows that buffering amortizes the cost of page table manipulation but also creates an unexpected new performance bottleneck – For sequential writes, buffering highly benefits from effectively amortized PTE flag change overhead. For random writes, however, the negative impact of losing parallelism dominates and causes substantial performance degradation (2.1 times).

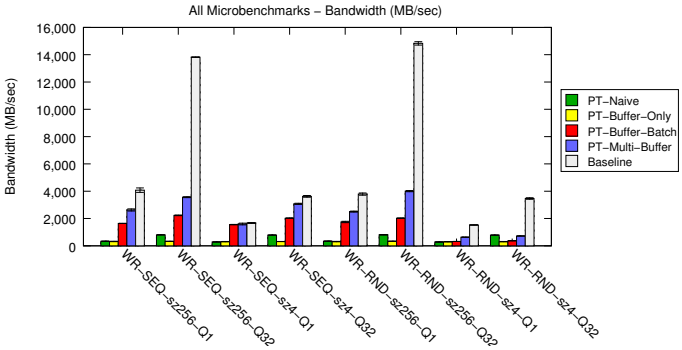


Fig. 6. Performance of multi-buffering

Optimization 2: Protection with Multiple Buffers – In order to improve scalability, multiple buffers can be used, each managed by an individual syncer daemon. Incoming writes are distributed across the buffers in a round-robin manner. Figure 6 shows the performance improvement of using multi-buffering (16 buffers). We see that using multiple buffers can effectively improve system performance. For example, the bandwidth of sequential writes of 4KB with 32 jobs improves from 2028 MB/sec to 3074 MB/sec. For random writes of 4KB with 32 jobs, the worst case for single buffer protection, its performance is improved by 1.9 times, from 374 MB/sec to 724 MB/sec. In these workloads, there is no data access locality, which means adding more buffers cannot bring any caching benefits. However, using multiple buffers enables concurrent buffer accesses in parallel. For workloads with a high queue depth, this optimization removes the buffer bottleneck and substantially improves performance.

In summary, page table based protection incurs a high performance overhead. In the above, we have studied a variety of solutions to address the performance overhead problem. We have improved write performance by nearly 5 times (up to 4,000 MB/sec). Other optimizations, such as disabling write protection bit in CR0 register, are also possible. Besides the performance problem caused by the TLB shootdown, the page table based protection also introduces several other issues. For example, we found that building a page table for a large amount of PM is intolerably slow during the module loading time. Also, for a large capacity of PM, the page table size would become huge and consumes a large amount of memory space. The TLB pollution problems would also emerge [27]. A fundamental reason behind these issues is that the existing page protection mechanism is originally designed for DRAM and not a natural fit to managing hundreds of GBs or even TBs of PM as persistent storage. We need a low-overhead protection for PM.

2) Private Mapping based Protection: We find that *private mapping* can effectively address all the aforesaid issues. Private mappings provide write protection by dynamically mapping PM pages into the kernel space *only when needed*, rather than by controlling the accessibility of each page. When the PM device driver is loaded, the PM pages are not immediately mapped into the kernel virtual address space. Instead, each page is mapped only when the page needs to be accessed (upon a `read`, or a `write`). As so, of most time, the PM space is 'invisible' to the OS kernel code, which prevents the potential damage caused by stray writes.

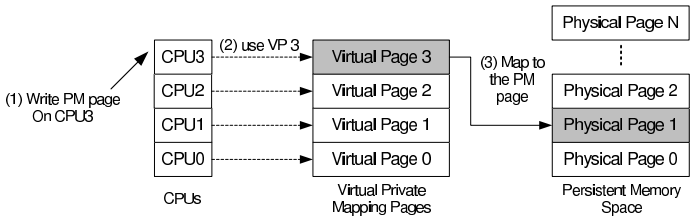


Fig. 7. An illustration of private mapping

For private mappings, we directly manipulate the page table. As shown in Figure 7, assuming a K-core system, we first allocate an array of virtual memory pages, $pmap[K]$, each entry of which is indexed by the CPU ID and private to the process running on it. When a process running on CPU n requests access to PM page p, the physical PM page p is temporarily mapped to the virtual page $pmap[n]$. Then we perform the `memcpy` operation to write data into the specified PM page. After the memory copy is done, we clear the $pmap[n]$ mapping entry. The interrupt is disabled to prevent any context switch. Since no concurrent requests running on other cores will use the same mapping entry, there is no need to shootdown the TLBs on the other cores, and we can completely avoid the related overhead.

Private mappings can provide sufficient write protection. Since we do not enforce the TLB shootdown, if a virtual page happens to be accessed by other cores during the short time window of performing the `memcpy` (typically thousands of cycles), the TLB translation would be loaded into the local TLB. In the worst case, at most $K-1$ TLB entries that are not supposed to be accessible could appear in a TLB, and in total, at most $K \times (K-1)$ entries may be affected at any time. Even so, this is a very small number, compared to the PM storage size, and system events, such as context switches, also flush TLB entries frequently, which further lowers the risk.

Figure 8 shows the performance of using private mappings. We have two key observations. (1) Private mapping can significantly improve write performance close to that of doing no protection (Baseline). For example, random writes of 256KB with 32 jobs can reach 13,389 MB/sec, which is 90% of the optimal case (no protection). (2) For reads, private mappings perform well too. For example, sequential reads of 256KB with 32 jobs reach a bandwidth of 16,076 MB/sec, which is around 85% of the optimal case performance (18,761 MB/sec). The read overhead is mostly due to the fact that when a page

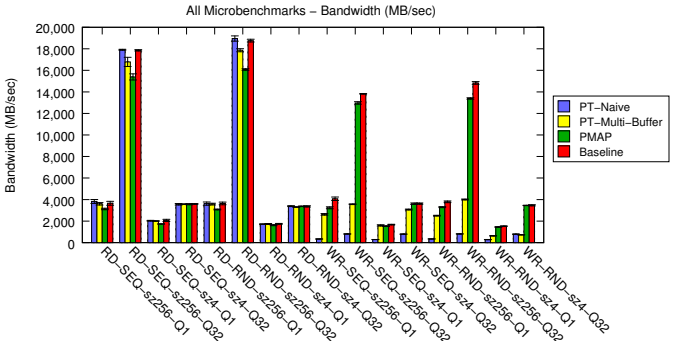


Fig. 8. Performance of private mappings

is to be accessed, it needs to be mapped into the kernel and then the mapping needs to be cleared and invalidated. Such a mapping/unmapping operation would incur additional TLB misses and affect performance, which explains why private mapping slightly underperforms than the others for intensive large reads. However, relative to the huge performance benefit for writes, such a subtle performance loss is acceptable.

Besides performance, private mapping also provides other benefits: (1) **Minimized vulnerable window** – With private mappings, only when an PM page needs to be accessed, is it mapped into the kernel, which minimizes the chance of corruption. The vulnerable window is typically thousands of cycles (while the page is being mapped). (2) **Protection for both reads and writes** – Any attempt to directly access PM data, not only writes but also reads, without going through the driver interface would be immediately blocked. (3) **Scalability with parallel accesses** – With no need for a buffer, the bottleneck for parallel I/O accesses can be removed. It also reduces design complexities and removes the risk of data loss due to the volatile buffer. (4) **Reduced page table size** – Since only one page mapping is needed when being accessed, there is no need to consume a large amount of memory space for building a huge page table for the entire PM storage, which could be as large as multiple Terabytes in the future. (5) **Reduced TLB pollution** – Since we only map the on-access pages, at most one TLB entry is needed on each core at any moment, which removes the risk of TLB pollution [27]. (6) **Fast loading/unloading time** – Since we do not have to build up the page table for all the PM pages, the time of loading and unloading the device driver can significantly be reduced. For all of these reasons, we choose the private mapping as our write protection mechanism.

C. Ordered Persistence

Ensuring ordered data persistence is important for the OS and applications. Storage devices usually provide an on-device buffer and interface ("flush") to write data in two phases – Data is first written into the buffer, and upon an OS write barrier (or the buffer is filled up), data is flushed from the buffer to the medium.

With PM, it is similar but more complicated. Since PM is physically managed by a memory controller and accessed by the `load`/`store` interface, the written data may reside in CPU caches. In other words, the CPU cache acts as a volatile buffer for the PM. Applications, if not changed, may lose data that is supposed to be persistent upon power failures. In our hybrid model, the PMBD driver is the single entity enforcing persistent and ordered writes, which removes the need to change applications. In this section, we first discuss several uncached write schemes for persistence and then how to ensure the write ordering.

1) Uncached Write Schemes: In the existing CPU architecture, we can explicitly make data persistent in several ways: (1) Specifying PM pages as *uncachable* or *write through*. In both modes, writes completely bypass the CPU cache and thus are extremely slow. We do not further consider them in this paper. (2) In write-back mode, use a non-temporal store (e.g., `movntq`), which bypasses the CPU cache and uses the write-combining buffer, and use `sfence` to flush the buffered data to PM. (3) In write-back mode, use regular store instructions (e.g., `mov`) and `clflush` to flush the specified data from the cache, and then use `mfence` to ensure the data is written to PM. (4) In write-back mode, use regular store and `wbinvd` to flush the entire cache. This option is slow and can only be applied during a write barrier, which will be discussed later.

Figure 9 shows the performance of different write schemes: using private mapping in write-back mode, using private mapping with `clflush`/`mfence` or `movntq`/`sfence`, and using no protection in write-back mode (Baseline). When the request size is small (4KB) and the queue depth is high (32 requests), `clflush`/`mfence` outperforms `movntq`/`sfence`. The reason is that `clflush`/`mfence` loads data into the CPU cache, which is much bigger than the buffer used by a non-temporal store, and then flushes the cachelines. This creates more opportunities to pipeline the operations when handling multiple requests in parallel, while `movntq`/`sfence` would be congested in the buffer. However, in all the other cases, `movntq`/`sfence` performs significantly better than `clflush`/`mfence`.

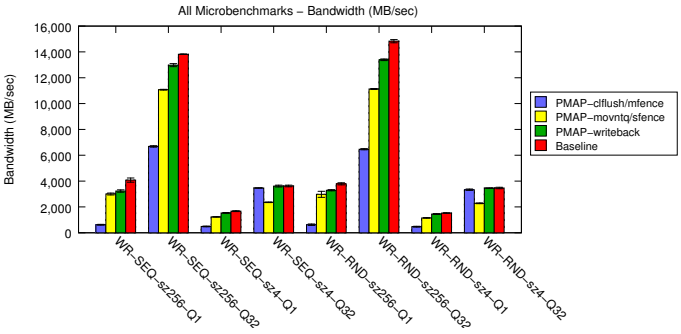


Fig. 9. Micro-benchmark performance of write schemes

We have also examined the write schemes for macro-benchmarks. Figure 10 shows execution times normalized to the baseline case (no protection) of various write schemes without write barriers. As we see, using regular stores with `clflush`/`mfence` performs the worst in most cases, however, the performance difference is smaller than that we see in the micro-benchmarks. Due to the effect of the page cache, most writes are performed asynchronously in the background,

which reduces the impact of write latency. Using a non-temporal store with `sfence` can achieve performance comparable to using private mapping without any persistence guarantees. This conclusion is consistent with that we see in the micro-benchmarks.

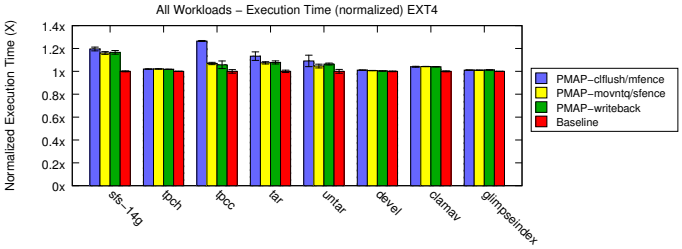


Fig. 10. Macro-benchmark performance results of various write schemes. Normalized to the baseline case.

2) Ordering Writes: Write ordering is needed for applications and systems to safely store data. For example, file system journaling must ensure all the transaction content be materialized to the persistent storage before completing the transaction. Database logging is also similar. With the write schemes described previously, data can be pushed to the memory bus and eventually persistent in PM. However, write ordering is still needed, because multiple parallel writes may exist, and the order of writes being processed may be random.

Write Scheme	Operations in Write Barrier
`movntq/sfence`	N/A
`mov/clflush/mfence`	N/A
`mov`	`wbinvd`

TABLE III
WRITE BARRIER OPERATIONS

Operating systems use *write barriers* to provide such a facility for the system (e.g., file system) and applications (e.g., database) to enforce write ordering and data persistence at a specific time. Upon receiving a write barrier, the OS instructs the storage device to flush its cache before processing any additional incoming writes. We need a similar write ordering mechanism for PM.

In the PM device, the flush command can be implemented as described in Table III. (1) If the non-temporal store (`movntq/sfence`) or (2) the ordered temporal store (`mov/clflush/mfence`) is used, data persistence is provided, and we only need to block the incoming requests (via a spinlock) until all in-progress requests complete for write ordering. (3) If we do not apply `clflush` after each store, we need to flush the entire CPU cache using `wbinvd` during the write barrier.

The abovesaid three schemes have different performance implications. The first two uncached write schemes invalidate the data in the CPU cache immediately after writes, which potentially affects performance due to the lack of cache usage. The third one, using regular store (`mov`) and flushing the entire CPU cache (`wbinvd`) on a write barrier, may provide a potential cache benefit for reads whose data has been written and still in the cache. However, since writebacks performed by the CPU are invisible to the PMBD driver, we cannot exactly track which data (cacheline) is resident in the cache or not. Although a bookkeeping mechanism could be developed to track every updated cacheline, it needs to simulate the CPU cache replacement accurately and is thus infeasible in practice. As so, `wbinvd` has to be used to flush the CPU cache during write barriers. Unfortunately, this could affect performance adversely.

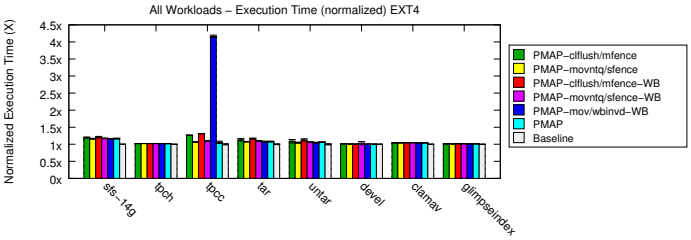


Fig. 12. Performance impact of write barriers

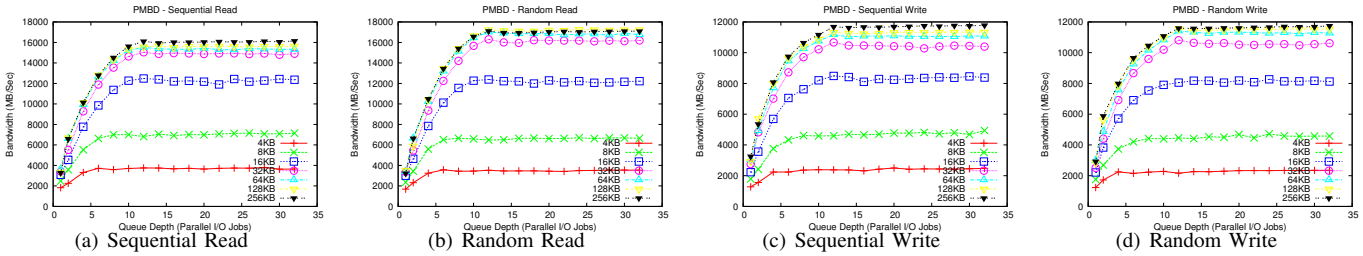
We run 8 macro-benchmarks on the PM device with the Ext4 file system to study the effect of write barriers. Figure 12 shows the performance of using write barriers combined with various write schemes. First, we find that using write barriers for `clflush/mfence` and `movntq/sfence` does not increase the performance overhead, since both write schemes only need to wait for in-progress accesses to finish, if there are any. Secondly, the expected benefit from using the CPU cache is negligible. This is mostly because the CPU cache is too small to effectively hold the working set for typical storage I/Os. In contrast, when handling workloads that generate a large number of write barriers, such as *tpcc*, which is a database workload, the negative impact of frequently flushing CPU cache is significant (a 4.1x slowdown). In almost all workloads, using non-temporal stores (`movntq/sfence`) provides the best performance.

D. Summary

Through the above studies, we have two key conclusions: (1) For write protection, we find that the private mappings are very effective and efficient. Although page table protection can be improved significantly with buffering and batching, it still incurs high performance overhead. (2) For ordered persistence, we find that using a non-temporal store (`movntq`) and `sfence` with write barriers performs the best. Using `clflush/mfence` would result in a performance loss in many cases. Using writeback and flushing the CPU cache with `wbinvd` for write barriers does not improve performance, and in some workloads, it degrades performance significantly. Figure 11 provides a performance overview of our PMBD driver using private mappings, non-temporal stores, and write barriers. With these techniques, reads and writes can achieve bandwidths of about 16GB/sec and 11GB/sec, respectively, which are about 75-85% of the peak (write-back mode with no protection).

V. MACRO-BENCHMARKS

In the above sections, we have investigated the design options to achieve performance, protection from stray writes and ordered persistence. In this section, we focus on the end-to-end performance of applications. We are interested in



(a) Sequential Read (b) Random Read (c) Sequential Write (d) Random Write

Fig. 11. PM performance with various queue depths, request sizes, and patterns

several issues: (1) The impact of file systems running on the PM block device, (2) the performance difference between memory-based file systems and legacy file systems running atop PM, (3) the performance benefit of the PM block device, compared to other block devices (e.g., flash SSDs), and (4) the end-to-end application performance impact of different PM characteristics (read/write speeds).

A. PM Devices vs. Alternatives

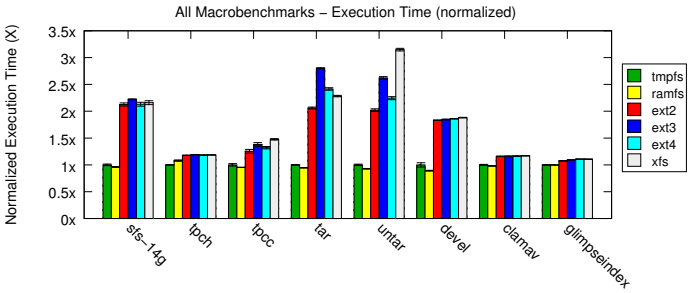


Fig. 13. Performance of tmpfs, ramfs, and PM devices (Ext2/3/4, XFS)

One interesting question is if we adopt the memory based model and design a completely new file system for PM (e.g., BPFS [8]), how much performance difference we would see compared to running a legacy disk-based file system (e.g., Ext4) on PM, which provides necessary protection and ordered persistence transparently. Since BPFS is implemented with FUSE as a user-level file system, its performance is incomparable to a native file system. Thus, we use the two RAM-based file systems in Linux, *tmpfs* and *ramfs*, to estimate the maximum performance difference. Both *tmpfs* and *ramfs* are directly integrated with the page cache and have no protection, so their performance can be regarded as the optimal case for memory-based file systems.

Figure 13 shows the experimental results. We can see that the two RAM-based file systems show higher performance than any stock disk-based file system running on PM. This is not surprising for several reasons. (1) Designed for DRAM, tmpfs and ramfs do not provide special mechanisms (e.g., write ordering and private mappings) for protection and persistence, which means less overhead. (2) Since tmpfs and ramfs are directly integrated with the page cache, there is no extra memory copy between the page cache and the storage, not to mention additional overhead for file system metadata accesses. (3) Since tmpfs and ramfs would not be filled up immediately, the applications have more available memory space for use during execution. In contrast, the PMBD driver would reserve all the memory space at the beginning. However, we should note that, even so, the performance difference between using a stock disk-based file system running on PM and a simple memory based file system (the optimal case) is workload dependent and not as significant as expected in some cases. In workloads, such as *tpch* (19%), *clamav* (16%), and *glimpseindex* (10%), the performance difference is rather small. In the worst case (*untar*), which is completely dominated by I/Os, the difference is about 3 times. This indicates that for most workloads, which are mixed with both computation and I/Os, adopting the block-level solution can deliver reasonable performance without sacrificing compatibility. For extremely I/O intensive workloads, such as *untar*, a customized file system for PM can bring additional performance benefits.

In the figure, we also can see that different file systems show different performance. Ext2 on PM performs the fastest for all workloads, while XFS and Ext3 perform the worst in most cases, since both are journaling file systems, which perform extra I/Os. This indicates that simplifying file systems can bring performance benefits. We also note that Ext4 performs better than Ext3, especially when a large number of small files are involved, such as *tar* and *untar*.

B. PMBD vs. HDD and SSD

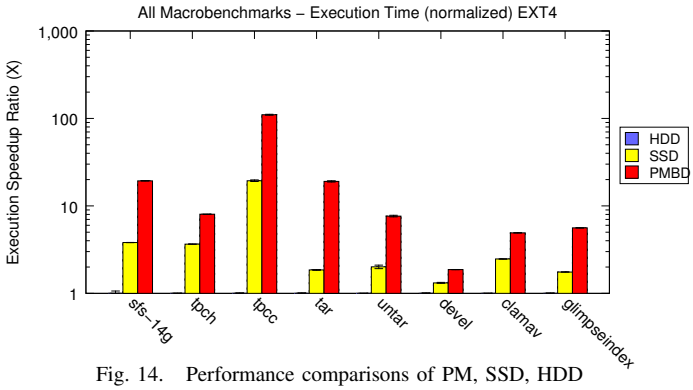


Fig. 14. Performance comparisons of PM, SSD, HDD

One may also ask how much benefit we would see in terms of end-to-end application performance by using PM, compared to a hard drive or an SSD. Figure 14 shows experimental results of running the eight workloads with HDD, SSD, and PMBD. As expected, PMBD clearly shows performance advantages in all cases. The relative performance improvement depends on the I/O intensity of workloads. For example, *devel*

involves much computation, and the speedup by using PMBD is relative smaller (1.8 times faster than HDD). In *tpcc*, the best case, PMBD is 110 times faster than HDD and 5.7 times faster than SSD. This is because *tpcc* generates a large number of small writes with syncs and write barriers, which is the worst case for magnetic disk drives, while PMBD shows superior performance for such workloads. This result also indicates that for databases, PM is a natural fit and can significantly improve system performance.

C. End-to-end Performance Impact of PM Speeds

As a technology in development, there is little device-level specification reported in public literature. Most specification numbers are based on cell-level speed, which cannot reflect the true device-level access speed. Rather than using the absolute time, as used in prior work, we specify the PM device performance by using *relative* speeds to DRAM. In particular, in our PMBD driver, the user can specify two slowdown parameters, `rdsx` and `wrsx`. Accordingly, we track the cycles spent on each memory copy operation and proportionally inject a delay by polling the `tsc` (timestamp counter) register to slowdown DRAM speeds. Using DRAM as a baseline, we can study how the performance would be affected without knowing the exact device specifications.

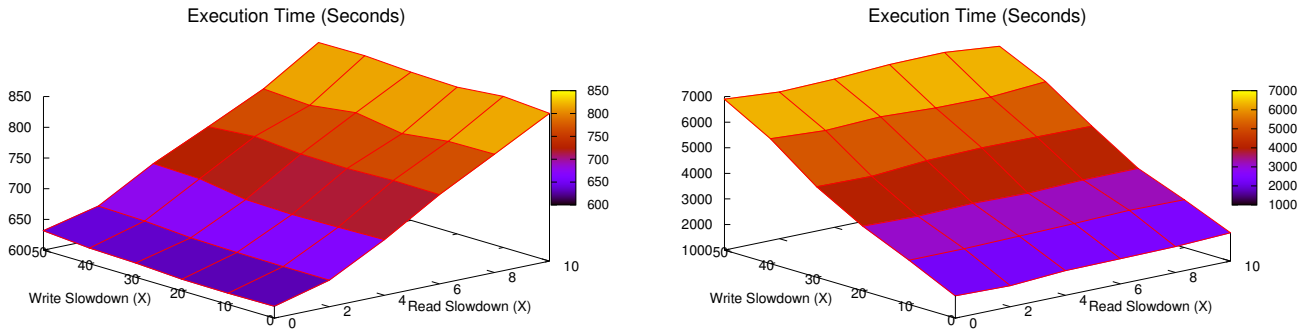
Figure 15 shows the performance impact of PM speed (slowdown factor: 1-10x for reads, 1-50x for writes). We show two representative workloads, *tpch* and *tpcc*. We have several observations: (1) The performance degradation is not proportional to the PM speed. For example, although the read latency of PM has been increased by 10 times, *tpch* performance is only reduced by 36%. (2) The impact to performance varies across applications. For example, the *tpcc* performance degrades by over 3.6 times in the worst case, while *tpch* performance degrades only by 1.4 times. (3) Applications show different sensitivities to read and write speeds. In particular, *tpch* is more sensitive to read performance, while *tpcc* is more sensitive to write performance.

VI. Related Work

Persistent memory has received increasingly high interest in academia and industry. Some suggest using PM to displace DRAM [16], [20], [21], [28]. Some propose to use a storage-based model, such as Onyx [3] and Moneta [6]. Some consider applications of PM, such as providing whole system persistence [18] and unioning the buffer cache and journaling layers [17]. Compared to the impact of flash memory to file and database systems [12], [23], PM is a more disruptive technology and also more deeply changing existing computing practices, from programming models to system designs. For example, Mnemosyne [26] provides a simple interface for programming with persistent memory, such as declaring persistent data with the keyword `pstatic`. CDDS [25] emphasizes more on providing a consistent and durable data structure for programmers to safely exploit the performance and persistence of PM. NV-Heaps [7] provides transactional semantics in an easy-to-use model. SoftPM [13] provides a new memory abstraction to allow `malloc` style allocations. Pâris et al. proposes to use storage-class memory for enhancing reliability of RAID [19]. Kang et al. proposes an object-based model for storage-class memory [15]. Prior work also considers designing new file systems for PM, where PM is directly attached to the memory bus. BPFS [8] is a file system designed specifically for PM. It uses shadow paging techniques to provide fast updates to critical file system structures, and hardware change is needed to provide epoch barrier for write ordering. In contrast, we use PM as a storage device. SCMFS [27] uses the page table based solution to manage PM files. In their results, they also found that TLB pollution can become a problem when handling a large amount of PM. In this paper, we show that using private mapping is efficient to address the TLB cost issue [4]. PMFS [11] provides a file system interface and allows user programs to use memory mapping (mmap) for directly accessing persistent memory. Such an approach can potentially deliver better performance by reducing extra memory copies, however it requires users to migrate to a new file system, and application modification is needed to fully take advantage of memory mapped I/Os. In contrast, we strive to avoid such changes. For write protection, PMFS manipulates the write protection bit in CR0 to avoid the TLB shootdown cost. We uses private mapping to address the same problem. A unique benefit enabled by private mapping is that it also solves the TLB pollution and page table size problems, since it does not require a page table for the entire PM storage. Our work and these prior studies represent distinct but consistent efforts in exploring the potential design space. The final design choices, though different, are simply the results of carefully balancing various factors on how to utilize this new technology with different emphasis.

VII. Discussions and Future Work

Integrating PM into today's computing systems is challenging. It demands a careful consideration of various factors, such as performance, cost, compatibility, etc. In contrast to many prior studies, we strive to reach a balance between exploiting the potential of PM and minimizing system changes. As an important goal of this work, we attempt to minimize disruptive changes to the existing eco-system. By creating a block-addressable device interface atop byte-addressable memory, certain optimization opportunities, such as byte addressability, could be lost. However, such a tradeoff is often worthwhile and necessary in practice. In fact we find that in many cases we can get most of potential performance benefits of PM without any disruptive changes. For example, many workloads running on a memory file system is only 10% to 20% faster than on a legacy disk file system on PM. This is for several reasons. First of all, many workloads access persistent storage in a reasonably large granularity (4KB or larger), and the cost of small I/Os (e.g., file system metadata accesses) are hidden by prefetching and caching effects. As so, enabling sub-page I/Os to PM may bring extra performance gains, but very likely at a limited scale. Also, since many real-life workloads (e.g., *devel*) are fixed with both computation and I/Os, the



(a) TPCH (b) TPCC

Fig. 15. Application performance with various PM specifications (read: 1-10x, write: 1-50x)

overall application performance is not purely dominated by I/O speed. Thus, further speeding up PM accesses may not lead to an extraordinary performance boost as we expect. On the other hand, we also note and confirm that for certain workloads with highly intensive I/Os, especially small I/Os and frequent storage syncs (e.g., *tpcc*), a PM-specific design is beneficial. Leveraging the byte addressability of PM and removing extra memory copies for such workloads can result in noticeable performance benefits. This also inspires us to further improve our solution in the future. For example, we can provide `mmap` support to allow programmers to directly access PM storage at a sub-page granularity. We may also develop other new interfaces to optimize certain applications as well. In short, we believe that as PM technology continues to develop, researchers need to invest more efforts to identify an suitable usage model to adopt this revolutionary technology.

VIII. Conclusion

Persistent memory technologies draw a blurry line between memory and storage. As a result, neither a memory-based model or a storage-based model is a best fit to achieve the goals of performance, protection from stray writes, and ordered persistence. In this paper, we present a hybrid model. We find that with private mappings, non-temporal store, memory fences, and write barriers, we can effectively achieve these goals with little impact to the OS and its applications. With all the lessons learned during this research, it is shown that determining the right combination of existing platform and OS mechanisms for PM is a non-trivial exercise. We have open-sourced our software prototype for public downloading. We hope that this work can encourage researchers to further explore various ways to exploit the benefits of PM.

Acknowledgment

The authors would like to thank anonymous reviewers for their constructive comments to improve this paper. We also thank Subramanya Dulloor, Sanjay Kumar, Rajesh Sankaran, Jeff Jackson, Paul Brett, Dheeraj Reddy, David Koufaty, and Ross Zwisler for discussions and support during this work.

References

[1] Open Storage Toolkit. http://www.sourceforge.net/projects/intel-iscsi.

[2] Persistent Memory Block Driver. http://www.github.com/linux-pmbd.

[3] A. Akel, A. M. Caulfield, T. I. Mollov, R. K. Gupta, and S. Swanson. Onyx: A Prototype Phase Change Memory Storage Array. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage 2011)*, Portland, OR, June 14 2011.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schupbach, and A. Singhania. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009)*, Big Sky, MT, October 2009.

[5] S. Boyd-Wickizer and N. Zeldovich. Tolerating Malicious Device Drivers in Linux. In *Proceedings of the 2010 USENIX Annual Technical Conference (USENIX 2010)*, Boston, MA, June 23-25 2010.

[6] A. M. Caulfield, A. De, J. Coburn, T. I. Mollov, R. K. Gupta, and S. Swanson. Moneta: A High-Performance Storage Array Architecture for Next-generation, Non-volatile Memories. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2010)*, Atlanta, Georgia, Dec 4-8 2010.

[7] J. Coburn, A. M. Caufield, 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 Memory. In *Proceedings of the 2011 Architectural Support for Programming Languages and Operating Systems (ASLPOS 2011)*, Newport Beach, CA, March 5-11 2011.

[8] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, D. Burger, B. C. Lee, and D. Coetzee. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 09)*, Big Sky, MT, October 2009.

[9] J. Corbet. The State of the e1000e Bug. In *Linux Weekly News*, Oct. 1 2009.

[10] B. Dieny, R. S. G. Prenat, and U. Ebels. Spin-dependent Phenomena and Their Implementation in Spintronic Devices. In *Proceedings of 2008 International Symposium onVLSI Technology, Systems and Applications (VLSI-TSA 2008)*, pages 70–71, April 2008.

[11] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, and R. S. J. Jackson. System Software for Persistent Memory. In *Proceedings of the 2014 European Conference on Computer Systems (EuroSys 2014)*, Amsterdam, ST, Netherlands, April 13-16 2014. The ACM.

[12] G. Graefe. The Five-Minute Rule 20 Years Later. In *Communications of ACM*. The ACM, July 2009.

[13] J. Guerra, L. Mármol, D. Campello, C. Crespo, R. Rangaswami, and J. Wei. Software Persistent Memory. In *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, June 13-15 2012.

[14] Y. Ho, G. Huang, , and P. Li. Nonvolatile Memristor Memory: Device Characteristics and Design Implications. In *Proceedings of 2009. IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers (ICCAD 2009)*, pages 485–490, Feb 5 2009.

[15] Y. Kang, J. Yang, and E. L. Miller. Object-based SCM: An Efficient Interface for Storage Class Memories. In *Proceedings of the 27th IEEE Conference on Mass Storage Systems and Technologies: Research Track (MSST 2011)*, Denver, CO, May 23-27 2011.

[16] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting Phase Change Memory as a Scalable DRAM Alternative. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, 2009.

[17] E. Lee, H. Bahn, and S. H. Noh. Unioning of the Buffer Cache and Journaling Layers with Non-volatile Memory. In *Proceedings of the 11th USENIX Conference on File and Storage Technologoies (FAST 2013)*, San Jose, Feb 12-15 2013.

[18] D. Narayanan and O. Hodson. Whole-system Persistence. In *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2012)*, March 2012.

[19] J.-F. Pâris, A. Amer, and D. D. E. Long. Using Storage Class Memories to Increase the Reliability of Two-Dimensional RAID Arrays. In *Proceedings of the 17th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS 2009)*, London, UK, Sept. 2009.

[20] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing Lifetime and Security of PCM-based Main Memory with Start-gap Wear Leveling. In *Proceedings of the 42th International Symposium on Microarchitecture (MICRO 2009)*, Dec 2009.

[21] 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 International Symposium on Computer Architecture (ISCA 2009)*, June 2009.

[22] S. Raoux, G. W. Burr, M. J. Breitwisch, C. Rettner, Y.-C. Chen, r. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. Lam. Phase-change Random Access Memory: A Scalable technology. In *IBM Journal of Research and Development*, volume 52(4), pages 465–479, 2008.

[23] J. Ren and Q. Yang. I-CASH: Intelligently Coupled Array of SSD and HDD. In *Proceedings of the 17th IEEE International Symposium on High Performance Computer Architecture (HPCA 2011)*, San Antonio, TX, Feb 2011.

[24] P. Synder. tmpfs: A Virtual Memory File System. In *Proceedings of the Autumn European UNIX User's Group Conference*, September 1990.

[25] 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 2011)*, San Jose, CA, February 15-17 2011.

[26] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Light Weight Persistent Memory. In *Proceedings of the 2011 Architectural Support for Programming Languages and Operating Systems (ASLPOS 2011)*, Newport Beach, CA, March 5-11 2011.

[27] X. Wu and A. L. N. Reddy. SCMFS: A File System for Storage Class Memory. In *Proceedings of Supercomputing (SC 2011)*, Seattle, WA, Nov 12-18 2011.

[28] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proceedings of the 36th International Symposium on Computer Architecture (ISCA 2009)*, June 2009.