SYSTEMS AND APPLICATIONS FOR PERSISTENT MEMORY

A Thesis
Presented to
The Academic Faculty

by

Subramanya R. Dulloor

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2015

SYSTEMS AND APPLICATIONS FOR PERSISTENT MEMORY

Date Approved: 27 October 2015

To the memory of my daughter, Rhea Anand Dulloor

To my wife (Ruchi Anand) and my parents (Mohan and Suguna)

To the memory of my advisor, Karsten Schwan

ACKNOWLEDGEMENTS

It has been quite a journey reaching this point; one that would not have been possible without the support of numerous people in my life.

First of all, I would like to thank my advisor Prof. Karsten Schwan. Prof. Schwan was the most supportive advisor one could ever hope for. He was there for me during the toughest times of my life, not only as an advisor but also as a mentor and as a symbol of strength. I would have never reached this stage without him and I will miss him. I would also like to thank Dr. Ada Gavrilovska and Dr. Mustaque Ahamad for mentoring and advising me at various points in my research.

My mentors at Intel (Rajesh Sankaran, Jeff Jackson, and Rich Uhlig) deserve special gratitude for their unwavering support over the last 5 years. Without their guidance and patience, I would have given up a long time ago.

The one thing that this journey has taught me is the importance of a good support system. I will forever be grateful to all my friends and colleagues who helped me stay motivated through the years. Vaibhav Dubey, Seema Dubey, Sanjay Kumar, Priyalee Kushwaha, Abhishek Dhanotia, Sabina Grover, Reagan Lopez, Bhavani Krishnan, Dheeraj Reddy, and Amitabha Roy deserve special mention.

Finally, I would like to thank my parents (Mohan and Suguna), my grandparents (Shyamala and Gauri), and my sisters (Pratima, Shalini, and Lakshmi) for their love and support. Above all, I would like to thank my wife Ruchi Anand, the most important person in my life. She has been a source of strength through the ups and downs of life for the last 10 years. We look forward to making up for the lost time.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Performance-hungry data center applications demand increasingly higher performance from their storage in addition to larger capacity memory at lower cost. While the existing storage technologies (e.g., HDD and flash-based SSD) are limited in their performance, the most prevalent memory technology (DRAM) is unable to address the capacity and cost requirements of these applications. Emerging byte-addressable, non-volatile memory technologies (such as PCM and RRAM) offer performance within an order of magnitude of DRAM, prompting their inclusion in the processor memory subsystem. Such load/store accessible non-volatile or persistent memory (referred to as NVM or PM) introduces an interesting new tier that bridges the performance gap between DRAM and PM, and serves the role of fast storage or slower memory. However, PM has several implications on system design, both hardware and software: (i) the hardware caching mechanisms, while necessary for acceptable performance, complicate the ordering and durability of stores to PM, (ii) the high performance of PM (compared to NAND) and the fact that it is byte-addressable necessitate rethinking of the system software to manage PM and the interfaces to expose PM to the applications, and (iii) the future memory-based applications that will likely employ systems coupling PM with DRAM (for cost and capacity reasons) must be extremely conscious of the performance characteristics of PM and the challenges of using fast vs. slow memory in ways that best meet their performance demands.

The key contribution of our research is a set of technologies that addresses these challenges in a bottom-up fashion. Since the real hardware is not yet available, we first implement a hardware emulator that can faithfully emulate the relative performance characteristics of DRAM and PM in a system with separate DRAM and emulated PM regions. We use this emulator to perform all of our evaluations. Next we explore system software support to enable low-overhead PM access by new and legacy applications. Towards this end, we implement PMFS, an optimized light-weight POSIX file system that exploits PM's byte-addressability to avoid overheads of block-oriented storage and enable direct PM access by applications (with memory-mapped I/O). To provide strong consistency guarantees, PMFS requires only a simple hardware primitive that provides software enforceable guarantees of durability and ordering of stores to PM. We demonstrate that PMFS achieves significant (up to an order of magnitude) gains over traditional file systems (such as ext4) on a RAMDISK-like PM block device.

Finally, we address the problem of designing memory-based applications for systems with both DRAM and PM by extending our system software to manage both the tiers. We demonstrate for several representative large in-memory applications that it is possible to use a small amount of fast DRAM and large amounts of slower PM without a proportional impact to an application.s performance, provided the placement of data structures is done in a careful fashion. To simplify the application programming, we implement a set of libraries and automatic tools (called X-Mem) that enables programmers to achieve optimal data placement with minimal effort on their part. Finally, we demonstrate the potentially large benefits of application-driven memory tiering with X-Mem across a range of applications.

CHAPTER I

INTRODUCTION

The computing industry in undergoing a major transformation due to the emergence of ”big data” problems that necessitate data-intensive processing at a scale not seen before [53, 107, 75]. Data center applications require increasingly more compute, memory, and storage performance to process this deluge of data in a timely manner. A parallel trend that is engendering the demand for performance is the rise of virtualized (cloud) infrastructures that are reaping the economic benefits of server consolidation [56, 100] by sharing platform resources among multiple applications.

While the compute power has been improving steadily over the years, the storage and memory performance have languished up until recently. On the storage side, while NAND flash has recently helped bring down the high performance gap between traditional storage and memory [65, 62, 106], it has hit the performance wall and is still up to three orders of magnitude slower than DRAM. In addition, the fact that NAND flash is suited only for use as block devices (i.e., SSDs attached to the IO interfaces) means additional overheads associated with the traditional block-oriented storage stacks [105].

On the other hand, memory-based data center applications like key-value stores, in-memory databases, and data analytics require servers with increasingly larger memory capacities. However, scaling the performance or capacity of DDR-based DRAM memory (DDR-DRAM) is becoming challenging due to physical limitations [73, 90].

To address these challenges, industry is exploring new memory technologies. non-volatile DIMMs (*NVDIMMs*), for instance, attach storage directly to the scalable memory (DDR) interface [94, 19]. NVDIMMs are gaining popularity due to their ability to provide low-latency predictable performance at high rated IOPS [62, 94]. But, despite being attached to the CPU, large capacity NAND-based NVDIMMs are still accessed as block devices [94] in a separate address space, due to the inherent properties of NAND [90]. Battery-backed

DRAM is a high performance (and more expensive) alternative with some adoption in data centers [55], but the capacity of the technology is still subject to the challenges to scaling DRAM [90].

On-package stacked DRAM (3D-DRAM) [73, 26] addresses the performance limitations of DDR-DRAM with vertically stacked DRAM modules that are attached to high-density on-package (or on-chip) interconnects. It has lower latency and much higher bandwidth compared to DDR-DRAM, but its capacity is limited to a few gigabytes vs. the hundreds of gigabytes to terabytes possible with DDR-DRAM [26].

The most promising of these technologies – expected to provide large capacity, byte-addressable, persistent memory – are the storage class memories such as PCM and RRAM [63, 90, 35]. These non-volatile memory (NVM) technologies are positioned between DRAM and secondary storage (such as NAND flash), both in terms of performance and cost. Further, unlike NAND based devices, NVM can be attached directly to the CPU and addressed by regular load/store instructions. In other words, NVM (also referred to as persistent memory or PM) can be used as either fast storage, or as high capacity and low cost (but slower) memory.

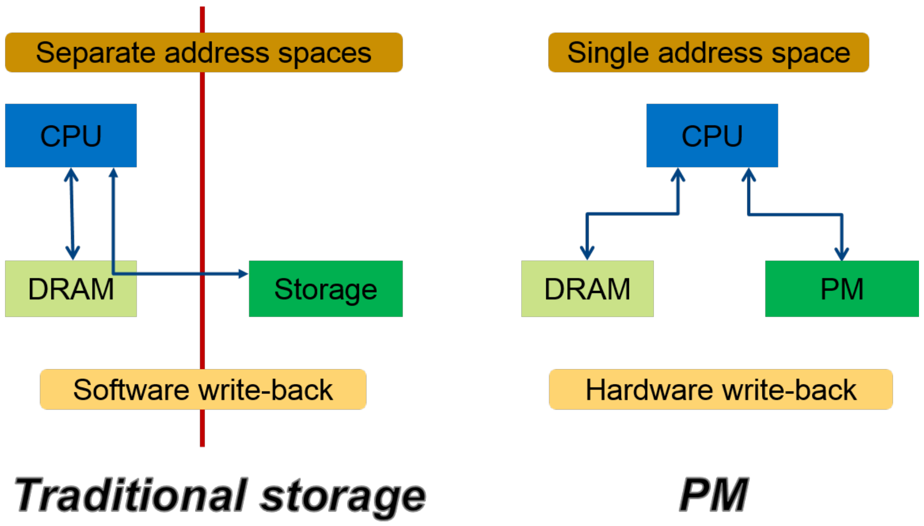


Figure 1: A comparison of traditional vs. PM storage architecture.

PM has several implications on system architecture, system software, libraries, and applications [90, 52, 101, 51]. This dissertation addresses some of the opportunities and challenges arising from the advent of fast load/store accessible storage that can also be used as high capacity, slower memory coupled with DRAM in a hybrid memory system. The main requirement of our research is that both DRAM and PM in a hybrid memory system are managed completely in software, and not by transparent hardware mechanisms [90]. We explore the use of PM as both fast storage and scalable (but slower) memory, and address the following key research problems:

- *Architectural support*: PM's byte-addressability and the fact that PM can be used as regular (write-back) cacheable memory introduces new challenges to software that must ensure consistency (i.e., ordering and durability) of stores to PM. Traditionally, the storage has always resided in a different address space from memory (Figure 1). For software to access the storage, it has to first copy the data to DRAM and then access the data. Similarly, to flush the data back to the storage, software has to explicitly initiate the flush operation. Software is always in control of when and how the data becomes persistent.

  With PM, the storage resides in the same address space as memory, and is part of the cpu's memory hierarchy. Software can access PM directly using regular load/store instructions. Exploiting this capability, we can eliminate many layers of software overhead by optimizing the software stack for PM (discussed next). And, for performance reasons, software would also want to map PM as (write-back) cacheable. But, in doing so, software no longer has control over when the dirty data is flushed to the PM. The dirty data in the CPU caches could be evicted anytime by the hardware, and the writes to PM can be reordered at several places in the hardware. This complicates the ordering, durability, and therefore consistency mechanisms in the software, and necessitates additional architectural support.

- *Low overhead access to PM*: PM is orders of magnitude faster than traditional storage (HDDs and even NAND flash based SSDs). Research has shown that, as storage becomes faster, software overheads tend to become the most dominant source of wasted effort, therefore necessitating rethinking of the software stacks [105]. As discussed earlier, traditional storage stacks assume that storage is in a different address space, and operate on a block device abstraction. They implement intermediate layers such as page cache to stage the data. When using PM, such a layered design results in unnecessary copies and translations in the software. It is possible to eliminate these overheads by completely avoiding the page cache and the block layer abstraction. Providing low overhead (but managed) access to PM is critical to ensure that applications harness the full potential of PM, and one of the key problems addressed in this dissertation.

- *Application design in hybrid systems*: The likely deployment of PM in systems coupling DRAM and PM is interesting to memory-based data center applications that require scalable, cost-effective main memory. In such scenarios, PM is used mainly for its low cost and high capacity (persistence is secondary), and can therefore be expected to constitute the majority memory in such hybrid memory systems. The main concern then would be the performance overheads with the memory-intensive applications (with limited locality), particularly if PM is used in a transparent manner. As a result, hybrid memory places the burden on applications to use fast vs. slow memory in ways that best meet their performance demands. The research problem is to develop principled tiering techniques for matching application data with the underlying memory types.

1.1 Thesis Statement

Emerging non-volatile (or persistent) memories bridge the performance and capacity gap between memory and storage, thereby introducing a new tier. To harness the full potential of future hybrid memory systems coupling DRAM with PM, we must build new system software and application mechanisms that enable the optimal use of PM as both fast storage and scalable low cost (but slower) memory.

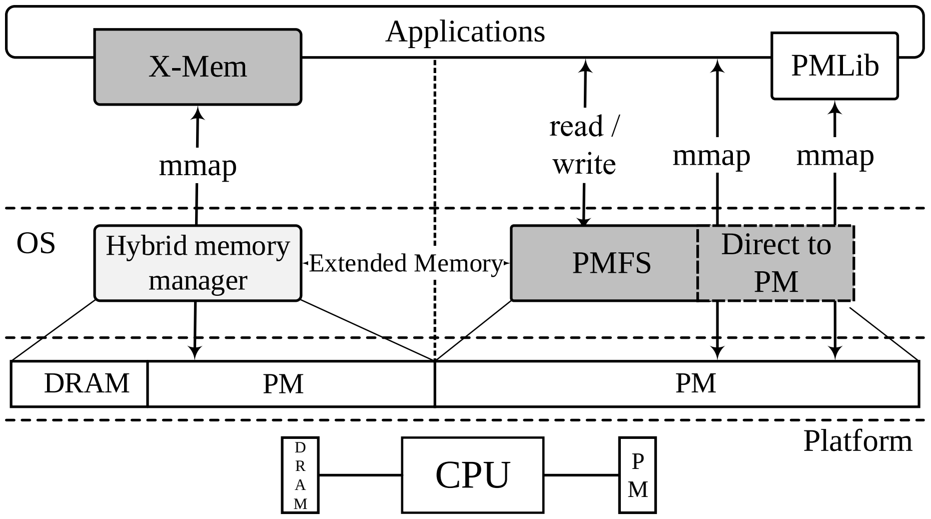


Figure 2: An overview of the PM (and hybrid memory) software stack.

1.2 Contributions

The key contribution of our research is a set of technologies (Figure 2) that addresses the aforementioned challenges. Specifically, this dissertation makes the following contributions:

- *Hybrid memory emulator*: Since the real hardware is not yet available, we implement a hardware emulator that can faithfully emulate the relative performance characteristics of DRAM and PM in a system with distinct DRAM and emulated PM regions. In addition, we emulate the performance of the proposed hardware primitives necessary to ensure consistency of the PM software. We use this emulator to perform all of our evaluations.

- *System software for PM*: To enable low overhead access to PM for legacy and new applications, we implement PMFS, a light-weight POSIX-compliant file system that is optimized for byte-addressable PM. By avoiding the block layer and eliminating the copy and translation overheads from maintaining a separate address space, PMFS improves the performance of several legacy applications by an order of magnitude. In addition, PMFS provides applications with direct access to PM via the memory-mapped interface, and implements features such as transparent large page support [23] to further optimize memory-mapped I/O. Finally, PMFS exploits PM's byte-addressability yet again to optimize consistency using a combination of atomic in-place updates, logging at cacheline granularity (fine-grained journaling), and copy-on-write (CoW).

- *Data tiering in hybrid memory systems*: We present a case for application data tiering in hybrid memory systems by demonstrating that a large portion of the overheads from using slower memory can by removed by judiciously placing some application data structures in the small amount of available DRAM. The selection of these data structures of course depends on their access pattern. We implement a memory management infrastructure (called X-Mem) to help programmers automatically identify the properties of application data structures and to enable automatic data placement at run time. The resulting system supports a standard allocator API with minor extensions (hence the name "extended memory" or X-Mem), and requires minimal modifications to the application code.

1.3 Organization

The remainder of this dissertation is organized as follows.

Chapter 2 describes the hybrid memory emulator and presents a detailed performance characterization of the platform.

Chapter 3 describes the design and implementation of PMFS, along with an evaluation of the system on the emulator platform.

Chapter 4 motivates the importance of data tiering in hybrid memory systems and presents the design, implementation, and a thorough evaluation of X-Mem.

Chapter 5 discusses related work, including brief architectural specifics of future hybrid memory systems and their impact on software design.

Chapter 6 concludes the dissertation and presents ideas for future research.

CHAPTER II

THE HYBRID MEMORY EMULATOR

2.1 Background

Modern processors like the Intel Xeon processor used in this study support muliple features aimed at hiding the memory latency for various access patterns [27]. These features can be summarized as follows:

(a) *Out-of-order execution*: The execution cores in modern processors are massively (and increasingly) out-of-order. For example, the out-of-order engine in Intel 3^{rd} generation Core processors (used for all the experiments in this paper) can handle 168 μops (or instructions) in flight, with the number increasing to 192 μops in Intel 4^{th} generation Core processors [27]. Large out-of-order windows enable speculative execution and aggressive reordering of μops. The improvements in the front end (particularly branch prediction) and the increasingly large load buffers enable a large number of in-flight loads (up to 64 in our test processor).

(b) *Hardware prefetchers*: Modern processors implement several hardware cache prefetchers that monitor the pattern of the downstream memory accesses to predict the data program is likely to consume and bring it from the lower levels of cache hierarchy and memory. The prefetchers are very effective for sequential accesses, and are increasingly aggressive. In our test processor, the streamer prefetcher of the second level cache tracks up to 32 streams of data accesses and brings in data up to 20 cachelines ahead of the load request [27].

Depending on the actual implementation and the memory access pattern, mitigating factors to the high memory latency are modern processors' (e.g., $Intel^{®}$ $Xeon^{®}$) extensive use of out-of-order execution and aggressive hardware prefetching [27]. These features can successfully reduce the average latency of memory reads, for certain access patterns, by reducing the number of cache misses and increasing memory-level parallelism (MLP) [50].

These effects are demonstrated in Figure 3, which shows the average latency of memory reads (for various access patterns) on an Intel Xeon E5-4620 system. In these experiments,

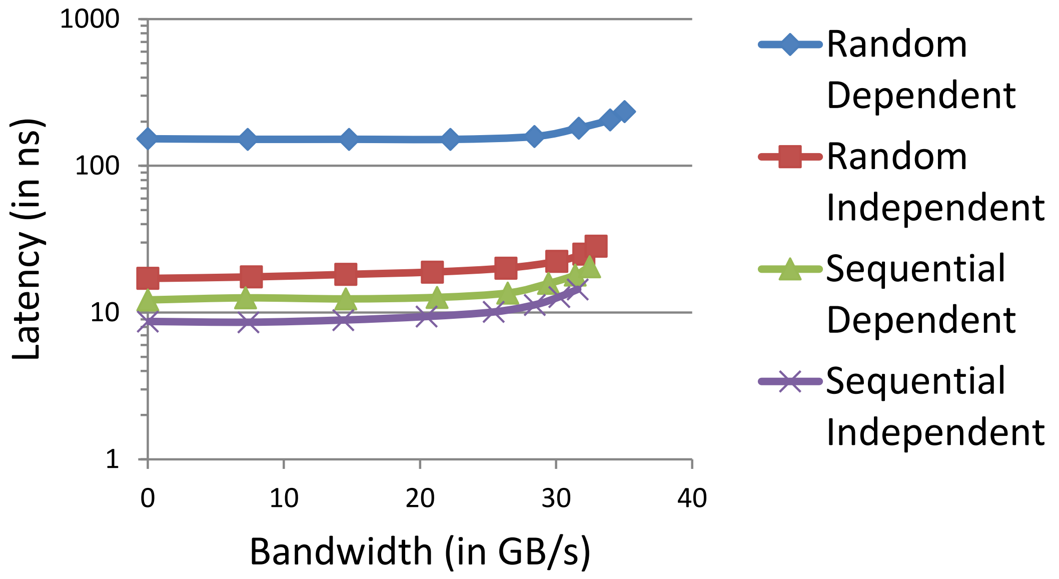


Figure 3: Memory read latency for various access patterns.

one thread reads memory (in the specified pattern) and measures average latency while other threads consume memory bandwidth by accessing their private memory. For *dependent* accesses, the latency thread's memory is initialized for pointer chasing. The locations for *independent* accesses are generated on the fly without any dependencies. Dependent can have only one memory load waiting for execution, and therefore does not benefit from out-of-order execution. Independent can have many in-flight loads.

Random dependent represents the worst case scenario, with every load experiencing the entire memory latency. In comparison, *random independent* is an order of magnitude faster due only to MLP. Similarly, *sequential dependent* is an order of magnitude faster than random dependent, but entirely due to hardware prefetchers. *Sequential independent*, which benefits from both MLP and prefetchers, shows the best performance of all.

The key observation from this experiment is that the performance of memory-intensive applications depends heavily on the pattern of their memory accesses.

2.2 The Emulator Platform

The *hybrid memory emulation platform (HMEP)* enables the study of hybrid memory with real-world applications by implementing – (i) separate physical memory ranges for DRAM and emulated NVM, and (ii) fine-grained emulation of their relative latency and bandwidth characteristics.

HMEP is based on a dual-socket Intel Xeon E5-4620 platform, with each processor containing eight 2.6 GHz cores. Hyperthreading is disabled. Each CPU supports four DDR3 channels and memory is interleaved across the CPUs.

Separate DRAM and NVM physical ranges: Using custom BIOS firmware, HMEP partitions the four memory channels of a CPU equally between DRAM and emulated NVM. The NVM region is available to software either as a separate NUMA node (managed by the OS) or as a reserved memory region (managed by PMFS) [57].

Read latency emulation: Memory read latency is often critical to the performance of memory-intensive applications. If there is a dependency in the program flow, then the CPU execution core stalls waiting for the data to be read. Since NVM has higher read latency than DRAM, these stalls are longer, resulting in worse performance. On the other hand, if there are no dependencies, CPU's MLP results in lower average read latency and fewer wasteful core stalls. HMEP emulates read latency on the NVM physical range using special CPU microcode, which uses debug hooks in the CPU to implement a performance model for latency emulation. The model monitors a set of hardware counters over very small intervals, and for each interval estimates (and applies) the additional cycles that the core would have stalled if the underlying memory was slower than DRAM. A naive method of calculating *stall cycles* would be to count the number of actual memory accesses (i.e., last level cache misses) to NVM and multiply it by the desired extra latency. This method, however, is suited only for simple in-order processors and highly inaccurate for modern out-of-order CPUs (§2.1).

We implement a model based on the observation that the number of cycles that the core stalls waiting for the memory reads to complete is proportional to the actual memory latency. If Lp is the target latency to emulated NVM, then the additional (proportional) stalls that the model applies for the time interval is:

$\delta_{stall} = S \times \frac{L_p - L_d}{L_d}$, where S is the actual number of stall cycles due to accesses to the emulated NVM range, L_p is the desired NVM latency, and L_d is the actual latency to DRAM.

In calculating S, we are limited to the following available counters on our test processor:

- Core execution stall cycles due to second level cache (L2) misses (S_{L2}).
- Number of hits in LLC (H_{LLC}).
- Number of last level cache (LLC) misses to DRAM (M_{dram}) and NVM (M_{nvm}) ranges.

Using these counters, the model first computes the execution stalls due to LLC misses (S_{LLC}) as follows:

$S_{LLC} = S_{L2} - (H_{LLC} \times K)$, where K is the difference in latency of a LLC hit and a L2 hit.

Finally, the model computes S as:

$$S = S_{LLC} \times \frac{M_{nvm}}{M_{dram} + M_{nvm}}.$$

Validation: To validate the model, we emulate the latency of slower NUMA memories in multi-processor platforms and compare the performance of several application on emulated NVM vs. actual NUMA memory. Following this approach, we validated the latency emulation model for a large number of applications – including several microbenchmarks (e.g., various sort algorithms) and benchmarks from SPEC CPU2006. Performance with NVM (emulating remote memory latency) is always within 7% of the performance with actual remote memory.

Limitations: NVM device characteristics are very different from that of DRAM. For instance, reads and writes to a PCM device have to wait for the preceding writes to the same memory line to complete [87]. The HMEP latency emulation model emulates only the average latencies and not NVM's device-specific characteristics. This restriction is primarily due to the limited internal CPU resources available for NVM latency emulation.

CPU hardware prefetchers can drastically improve the performance of sequential and strided memory accesses. HMEP assumes that the prefetchers will continue to be at least as effective with NVM as they are today with slow remote memory (of comparable latency) on large NUMA platforms. This assumption is reasonable even if we ignore the fact that, if needed, CPU prefetchers could be assisted by some form of prefetching on the NVM modules as well.

Bandwidth emulation: NVM has lower sustained bandwidth than DRAM, particularly for writes (Table 1), though that could be improved using ample scale-out of NVM devices and buffers. [1] HMEP emulates read and write bandwidths by programming the memory controller to limit the maximum number of DDR transactions per μsec. This throttling feature can be programmed on a per-DIMM basis [20], and is applied only to the NVM range.

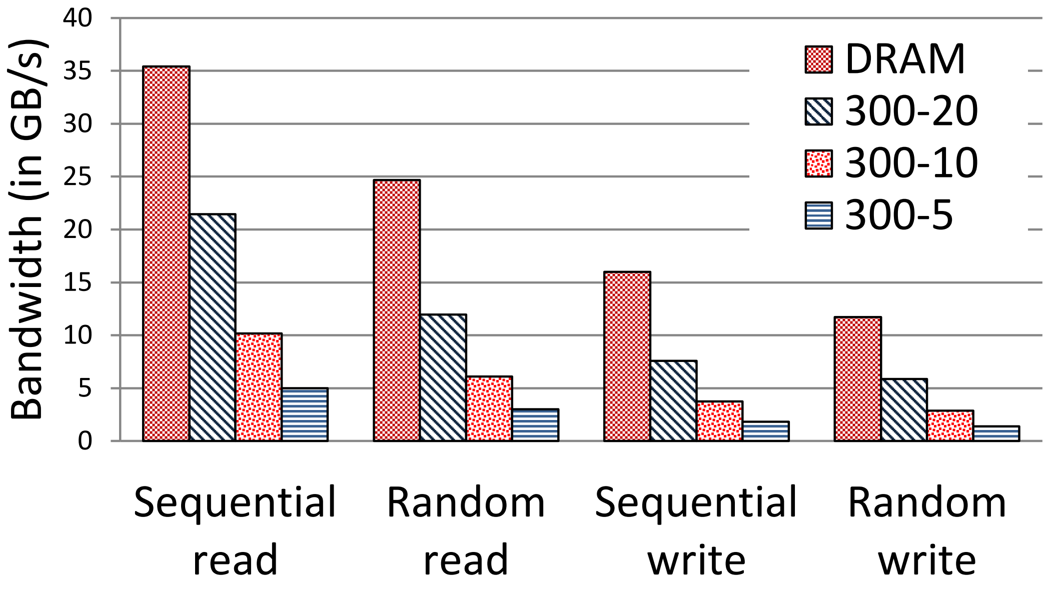
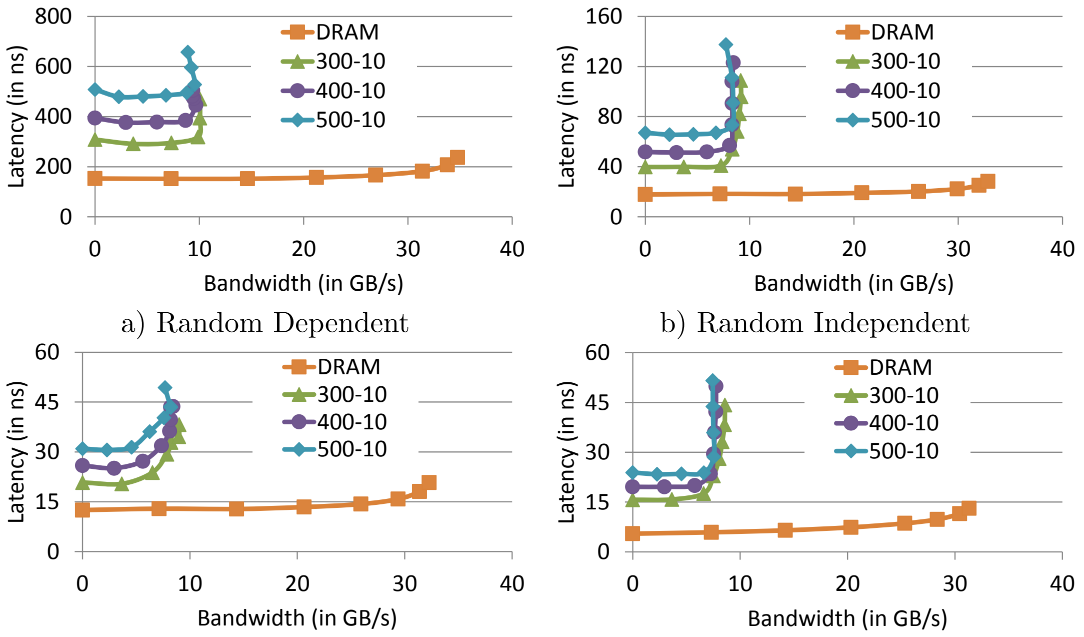


Figure 4: Bandwidth of HMEP configurations

Limitations: The bandwidth throttling feature in the memory controller is a single knob that limits the rate of all DDR transactions. Therefore, HMEP is unable to vary the read and write bandwidths independently.

Characterization: Figure 5 shows latency and bandwidth characteristics for memory reads to the NVM range in various HMEP configurations. These configurations are denoted by *x*-*y*, where *x* is the emulated NVM read latency (in ns) and *y* represents the peak bandwidth (in GB/s) to the NVM range. Access patterns are as described earlier (§2.1). Read latency to NVM depends heavily on the access pattern (as with DRAM) – sequential and independent reads are much faster than random and dependent reads.

[1]Since writes to write-back caches are posted, NVM's slower writes result in lower bandwidth and not higher latency on every write.



a) Random Dependent

b) Random Independent

c) Sequential Dependent

d) Sequential Independent

Figure 5: Read latency-bandwidth plots for several HMEP configurations and all access patterns

Figure 4 shows the measured sustained bandwidth to NVM for various HMEP configurations and access patterns. As expected, sequential accesses achieve higher bandwidth than random accesses, and read bandwidth is higher than write bandwidth.

To summarize, despite the stated limitations, HMEP adequately emulates the relative characteristics of DRAM and NVM in a hybrid memory system, and also the performance behavior of various memory access patterns. During the course of this dissertation, HEMP is alternatively referred to as Persistent Memory Emulator Platform (or PMEP).

CHAPTER III

SYSTEM SOFTWARE FOR PERSISTENT MEMORY

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

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

3.1 Background

In recent years, NAND flash has helped bring down the historically high performance gap between storage and memory [65]. As storage gets faster, the trend is to move it closer to the CPU. Non-Volatile DIMMs (*NVDIMMs*), for instance, attach storage directly to the scalable memory (DDR) interface [94, 19]. NVDIMMs are gaining popularity due to their ability to provide low-latency predictable performance at high rated IOPS [62, 94] But, despite being attached to the CPU, large capacity NAND-based NVDIMMs are still accessed as block devices [94] in a separate address space, due to the inherent properties of NAND [90].

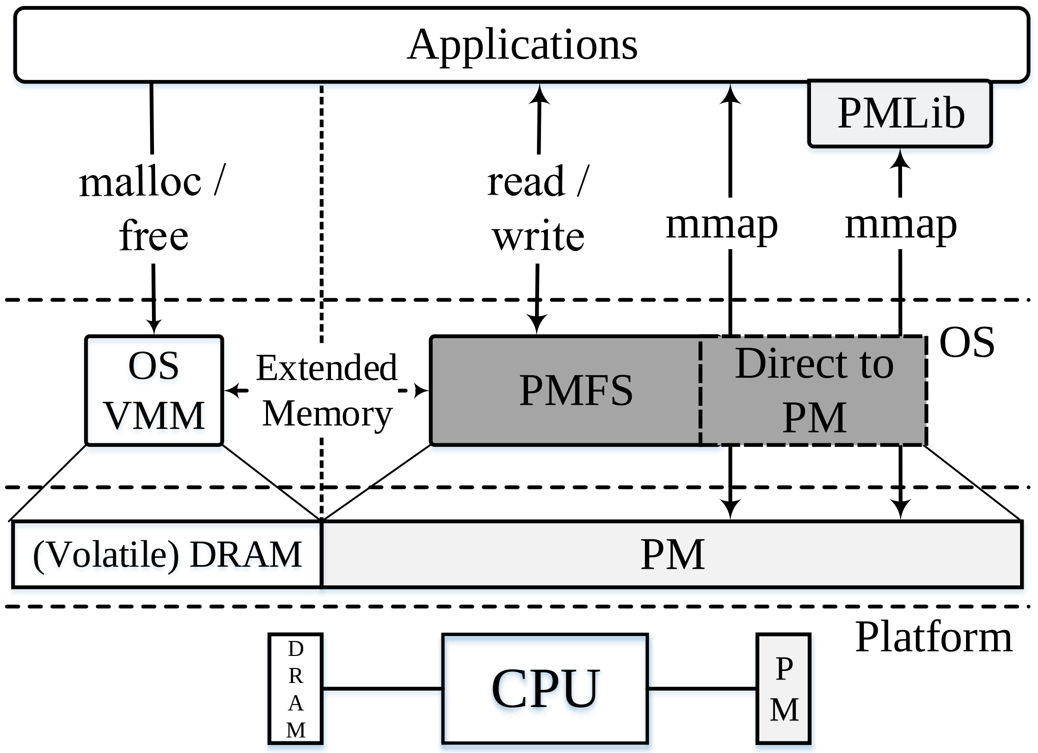


Figure 6: PM System Architecture

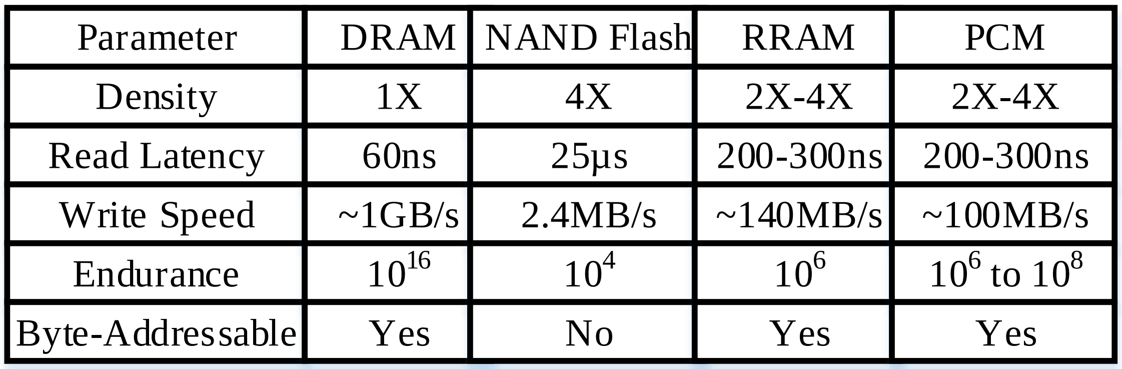
However, this appears likely to change in the near future due to emerging non-volatile memory technologies that are suited for use as large capacity, byte-addressable, storage class memory. Table 1 shows per-device characteristics of some of these technologies. We refer to such memory as *Persistent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

PM has implications on system architecture, system software, libraries, and applications [90, 52, 101, 51]. In this work, we address the challenge of system software support to enable efficient access of PM by applications.

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

(1) extending VMM to manage PM;

Table 1: Comparison of Memory Technologies [90, 18]



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

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

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

PMFS adopts the strategy of implementing a POSIX-compliant file system optimized for PM. Figure 6 shows a high-level overview of the proposed PM system architecture, with PMFS as the system software layer managing PM. PMFS has many advantages over the other two approaches:

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

(2) *Support for a light-weight file system.* Given the anticipated performance characteristics of PM, the overheads from maintaining a separate storage address space (e.g., operating at block granularity and copying data between storage and DRAM) become dominant [105, 65, 44]. By optimizing for PM and avoiding the block layer, PMFS eliminates copy overheads and provides substantial benefits (up to 22×) to legacy applications. Figure 7 shows a high-level comparison of the two approaches. PMFS exploits PM's byte-addressability to optimize consistency using a combination of atomic in-place updates, logging at cacheline granularity (fine-grained journaling), and copy-on-write (CoW).

(3) *Optimized memory-mapped I/O.* Synchronously accessing fast storage with memory semantics (e.g., using memory-mapped I/O) has documented advantages [105, 44]. However,

with traditional file system implementations, memory-mapped I/O would first copy accessed pages to DRAM, even when storage is load/store accessible and fast. PMFS avoids this overhead by mapping PM pages directly into an application's address space. PMFS also implements other features, such as transparent large page support [23], to further optimize memory-mapped I/O.

PMFS presented us with several interesting challenges. For one, PMFS accesses PM as write-back (WB) cacheable memory for performance reasons [45], but still requires a way to enforce both ordering and durability of stores to PM. To further complicate this situation, memory writes in most modern architectures are posted, with memory controllers scheduling writes asynchronously for performance. This problem is common to all PM software and not just PMFS. To address this issue, we propose a hardware primitive, which we call PM write barrier or *pm_wbarrier*, that guarantees durability of stores to PM that have been flushed from CPU caches (§3.2).

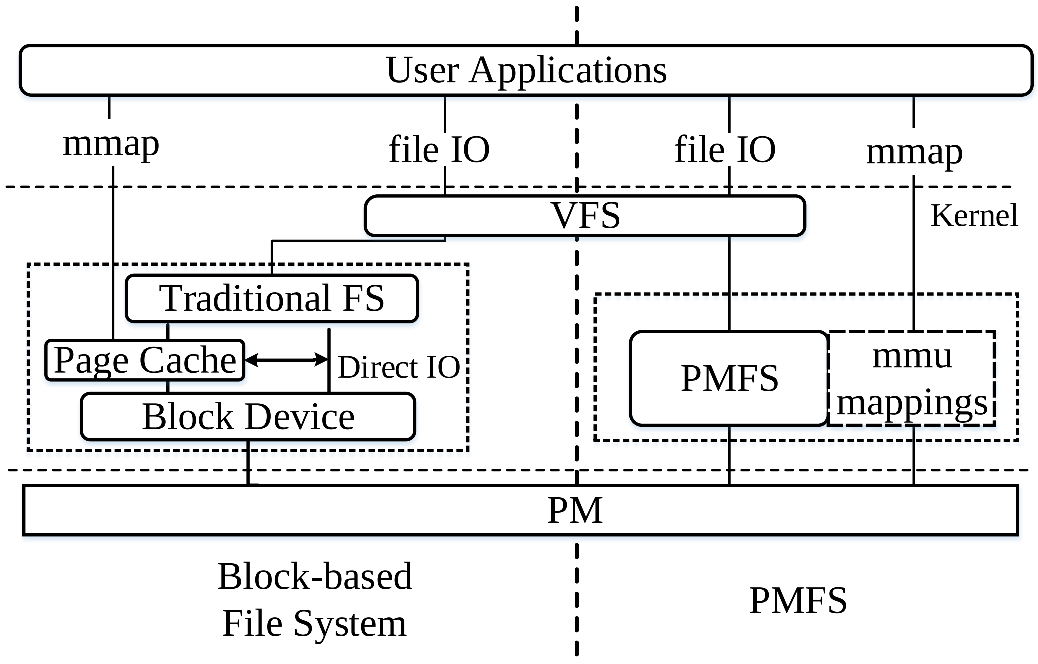


Figure 7: PMFS vs. Traditional File Systems

For performance and simplicity, PMFS maps the entire PM into kernel virtual address space at the time of mounting. As a result, PM is exposed to permanent corruption from stray writes due to bugs in the OS or drivers. One solution is to map PM pages as read-only in the CPU page tables, and temporarily upgrade specific PM pages as writable in code sections that need to write to them. However, this requires expensive global TLB shootdowns [23]. To avoid these overheads, we utilize processor write protection control to implement uninterruptible, temporal, *write windows* (§3.3.3).

Another challenge in PMFS is validation and correctness testing of consistency. Though a well-known problem for storage software [85], consistency in PMFS is further complicated by the need for careful consideration of processor memory ordering and use of *pm_wbarrier* for enforcing durability. For PMFS validation, we use a hypervisor-based validation tool that uses record-replay to simulate and test for ordering and durability failures in PM software (§3.4) [69].

Finally, while memory-mapped I/O (mmap) does provide memory-like access to storage, the interface is too low-level for many applications. Recently researchers have proposed new programming models to simplify direct use of PM by applications [101, 51, 99]. We envision such programming models and libraries, referred to as *PMLib* in Figure 6, building on PMFS using mmap for direct access to PM. We intend to explore PMLib, including integration with PMFS, in the future.

Contributions of this work are as follows:

- A high-level PM system architecture, including a simple new hardware primitive (*pm_wbarrier*) that provides software enforceable guarantees of durability and ordering of stores to PM.

- Design and implementation of PMFS, a light-weight POSIX file system with optimizations for PM and the processor architecture, such as fine-grained logging for consistency (§3.3.2), direct mapping of PM to applications with transparent large page support (§3.3.1), and a low-overhead scheme for protecting PM from stray writes by the kernel (§3.3.3).

- Detailed performance evaluation of PMFS with the PM hardware emulator, comparing PMFS with traditional file systems on a RAMDISK-like Persistent Memory Block

Device (*PMBD*).

In next section, we describe the proposed system architecture and *pm_wbarrier* primitive in detail (§3.2). We then present the design and implementation of PMFS (§3.3), followed by a detailed performance evaluation of PMFS (§3.5). Finally, we conclude the chapter with a brief survey of related research and thoughts on future work.

3.2 System Architecture

Figure 6 shows the high-level system architecture assumed in the dissertation. For illustration purposes, we assume standard high-volume server platforms and processors based on Intel® 64-bit architecture, but the concepts are applicable to other architectures as well.

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

A common flow for consistency in storage software such as PMFS requires a set of writes (A) to be durable before another set of writes (B) [52, 3]. Earlier research explored different approaches to managing ordering and durability of stores to PM, and implications on volatile CPU caches and store buffers. These include mapping PM as write-through (WT) [45], limiting PM writes to use non-temporal store instructions to bypass the CPU caches, or a new caching architecture for epoch based ordering [52].

In our evaluation, we encountered practical limitations with these approaches. While WT mapping offers the simplest solution to avoid caching related complications, it is not suited for use with PM due to both WT overheads [45] and limited PM write bandwidth (Table 1). Meanwhile, restricting PM stores to non-temporal instructions imposes programming challenges; for instance, in switching between cacheable loads and non-temporal stores. Non-temporal instructions also suffer from performance issues for partial cacheline writes, further discouraging general use. Finally, while an epoch-based caching architecture offers an elegant solution, it would require significant hardware modifications, such as tagged cachelines and complex write-back eviction policies. Such hardware mechanisms would involve non-trivial changes to cache and memory controllers, especially for micro-architectures with distributed cache hierarchies.

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

In most memory controller designs, for performance and scheduling reasons, writes to memory are treated as posted transactions and considered complete once accepted and queued. Also, for all memory write requests accepted, the memory controller enforces processor memory ordering (e.g., read-after-write ordering) [24] by servicing reads of in-flight writes from internal posted buffers. For existing volatile memory usage, such behavior is micro-architectural and transparent to software. But PM usage has additional implications, particularly since the durability point is beyond the memory controller's posted buffers.

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

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

PMFS requires only *pm_wbarrier* for correct operation. The performance of cache write back is vital to the proposed PM architecture. However, as reported by previous work [45, 101], current implementations of *clflush* are strongly ordered (with implicit fences), and therefore suffer from serious performance problems when used for batch flush operations. For this work, we assume and emulate an optimized *clflush* implementation that provides improved performance through weaker ordering on cache write back operations. Ordering is enforced in software with the use of memory fence operations (e.g., *sfence*). We used special write-combining stores to emulate the optimized *clflush* instruction, and observed up to 8× better performance compared to the strongly ordered *clflush* instruction (depending on the cacheline state).

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

One drawback of the proposed scheme is that software is required to keep track of dirty cachelines in PM. Our evaluation shows that the resulting performance gains justify the additional programming complexity. Moreover, for normal applications, most of the nuances of PM programming can be hidden behind programming models and libraries [101, 51, 99]. PMFS itself uses the proposed hardware primitives and the above mentioned software flow for most usage (for instance, in consistency and logging), and uses non-temporal stores sparingly for specific streaming write operations (e.g., in write system call).

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

3.3 PMFS Design and Implementation

Figure 7 shows a high-level software architecture of a system using PMFS, including a comparison with a traditional file system. PMFS design goals are:

(1) *Optimize for byte-addressable storage.* PMFS exploits PM's byte addressability to avoid the overheads of the block-based design of traditional file systems, such as copies to DRAM during file I/O and read-modify-write at block granularity (as opposed to cache-line granularity) for consistency. PMFS design, including layout (§3.3.1) and consistency (§3.3.2), is optimized for PM and the processor architecture.

(2) *Enable efficient access to PM by applications.* Because PM performance is comparable to DRAM, it is important to eliminate software overheads in accessing PM [105]. For this reason, PMFS optimizes file read, write, and mmap by avoiding unnecessary copies and software overheads (§3.3.1). File read/write in PMFS, for instance, requires only a single copy between PM and user buffers, while mmap avoids copying altogether.

(3) *Protect PM from stray writes.* PMFS maps the entire PM into kernel virtual address space for performance, which exposes PM to permanent corruption from stray writes due to bugs in the OS or drivers. To prevent this, PMFS implements a prototype low-overhead write protection scheme using a write protect control feature in the processor (§3.3.3).

3.3.1 Optimizations for memory-mapped I/O

PMFS Layout: PMFS data layout is shown in Figure 8. The superblock and its redundant copy are followed by a journal (*PMFS-Log*) and dynamically allocated pages. As in many other file systems, metadata in PMFS is organized using a B-tree, one of the best options for indexing large amounts of possibly sparse data. The B-tree is used to represent both the inode table and the data in inodes.

Allocator: Most modern file systems use *extent-based* allocations (e.g., ext4, btrfs), while some older ones are *indirect block-based* (e.g., ext2). Allocations in PMFS are *page-based*, with support for all processor page sizes (4KB, 2MB, 1GB), to enable transparent large page support [23]. By default, the allocator uses 4KB pages for metadata (internal) nodes of a data file's B-tree, but data (leaf) nodes can be 4KB, 2MB, or 1GB.

In many aspects, PMFS allocator is similar to the OS virtual memory allocator, except for consistency and durability guarantees (§3.3.2). Therefore, well-studied memory management issues, such as fragmentation due to support for multiple allocation sizes, apply to

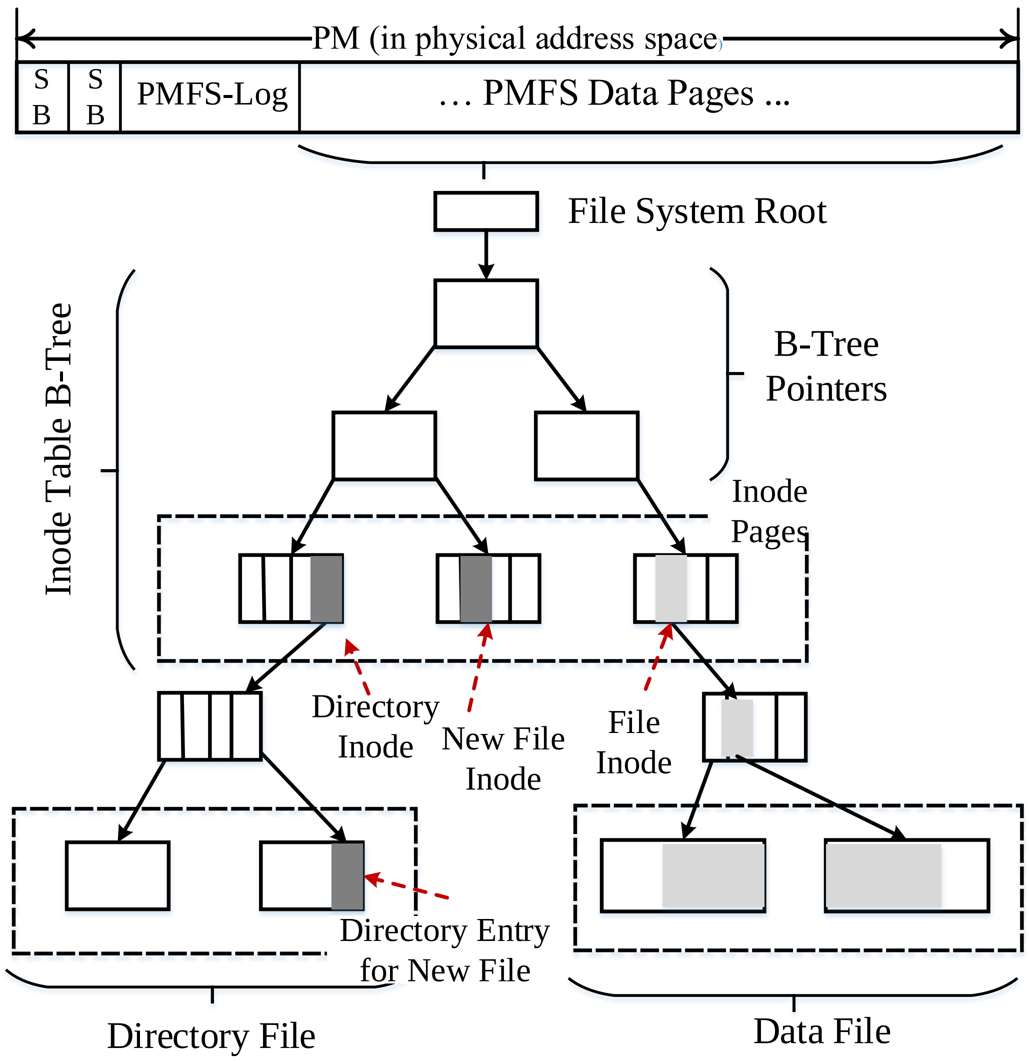
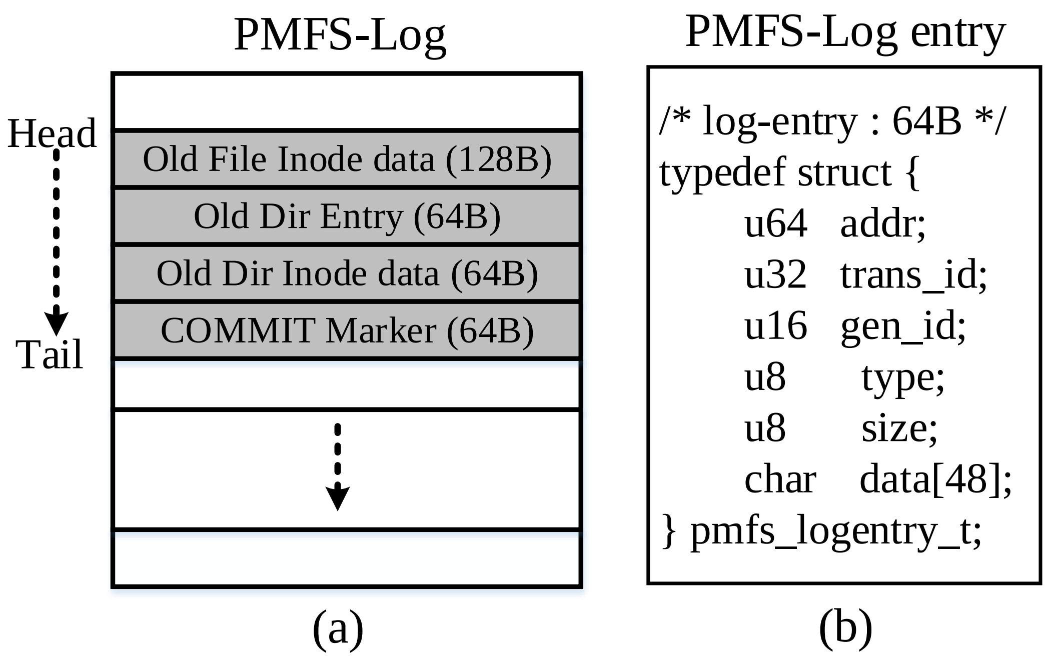


Figure 8: PMFS data layout

PMFS. In the current implementation, the PMFS allocator only coalesces adjacent pages to avoid major fragmentation. We plan to explore more strategies in the future.

Memory-mapped I/O (mmap): *Mmap* in PMFS maps file data directly into the application's virtual address space, so users can access PM directly. PMFS's *mmap* transparently chooses the largest hardware page table mappings, depending on *mmap* and file data node size. Using large page mappings has several benefits, such as efficient use of TLBs, fewer page faults, and shorter page table walks. Another benefit of using large pages is smaller page table structures, which is even more important at large PM capacities since page table pages are allocated from limited DRAM.



```
/* log-entry : 64B */
typedef struct {
      u64   addr;
      u32   trans_id;
      u16   gen_id;
      u8     type;
      u8     size;
      char   data[48];
} pmfs_logentry_t;
```

Figure 9: PMFS journaling data structures

However, using large pages without application hints could cause internal fragmentation. Therefore, by default, file data nodes in PMFS are 4KB. This default behavior can be overridden using one of the following strategies:

(1) Changing the default page size at mount time. This optimization works well for applications that use large files of fixed or similar sizes [7].

(2) Using existing storage interfaces to provide file size hints. If an application expects a file to grow to 10GB, for example, it can communicate this to PMFS using either *fallocate* (to allocate immediately) or *ftruncate* (to allocate lazily). These hints cause PMFS to use 1GB pages instead of 4KB pages for the file's data nodes. In our experience, modifying the applications to provide such file size hints is usually trivial. For instance, we were able to add basic *fallocate* support to Linux GNU coreutils [6] with just two lines of code, enabling large page allocations for file utilities such as *cp* and *mv*.

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

3.3.2 Consistency

A file system must be able to survive arbitrary crashes or power failures and still remain consistent. To provide this guarantee, file systems must implement some form of consistent and recoverable update mechanism for metadata and (optionally) data writes. From the file system's view, an operation is *atomic* if and only if updates made by the operation are committed in all or none fashion. In PMFS, every system call to the file system is an atomic operation. Note that applications directly accessing PM via *mmap* have to manage consistency of the file contents on their own, either within the application or using language, library, and runtime support [101, 51].

Modern file systems and databases use one of the following three techniques to support consistency: copy-on-write (CoW) [2, 52], journaling (or logging) [3], and log-structured updates [91, 65].

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

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

Journaling, however, has the drawback that all the updates have to be written twice; once to the journal, and then to the file system. For updates larger than a CoW file system's block size or a log-structured file system's segment size, journaling becomes less desirable due to this double copy overhead and the associated write amplification. Therefore, in PMFS, we use atomic in-place updates and fine-grained logging for the (usually small) metadata updates, and CoW for file data updates. We show (§3.5.2.3) that PMFS incurs much lower overhead for metadata consistency compared to BPFS, a PM-optimized file system that uses CoW and atomic in-place updates for metadata and data consistency [52].

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

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

As noted above, in PMFS, the original data is written and committed to the PMFS-Log (Figure 9) before the file system metadata is modified, thereby following undo semantics. If a failure occurs in the middle of a transaction, PMFS recovers on the next mount by reading the PMFS-Log and undoing changes to the file system from uncommitted transactions. To minimize journaling overhead, PMFS leverages processor features to use atomic in-place updates whenever possible, sometimes avoiding logging altogether. For operations where in-place updates to the metadata are not sufficient, PMFS falls back to use fine-grained logging.

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

Atomic in-place updates: As suggested by previous work, PM provides a unique opportunity to use atomic in-place updates to avoid much more expensive journaling or CoW [52]. However, compared to prior work, PMFS leverages additional processor features for 16-byte and 64-byte atomic updates, avoiding logging in more cases. PMFS uses the various atomic update options in the following ways:

- 8-byte atomic updates: The processor natively supports 8-byte atomic writes. PMFS uses 8-byte atomic writes to update an inode's access time on a file read.

- 16-byte atomic updates: The processor also supports 16-byte atomic writes using *cmpxchg16b* instruction (with LOCK prefix) [22]. PMFS uses 16-byte in-place atomic updates in several places, such as for atomic update of an inode's size and modification time when appending to a file.

- 64-byte (cacheline) atomic updates: The processor also supports atomic cacheline (64-byte) writes if *Restricted Transactional Memory (RTM)* is available [21]. To atomically write to a single cacheline, PMFS starts a RTM transaction with XBEGIN, modifies the cacheline, and ends the RTM transaction with XEND, at which point the cacheline is atomically visible to rest of the system. On a subsequent *clflush*, the modified cacheline is written back to PM atomically, since the processor caching and memory hardware move data at least at cacheline granularity. PMFS uses cacheline atomicity in system calls that modify a number of inode fields (e.g., in deleting an inode). Note that if RTM is not present, PMFS simply falls back to use fine-grained logging.

Journaling for Metadata Consistency: PMFS uses undo journaling and fine-grained logging. The main logging data structure is a fixed-size circular buffer called PMFS-Log (Figure 9(a)). The head and tail pointers mark the beginning and end, respectively, of the sliding window of *active* logged data. For every atomic file system operation that needs logging, PMFS initiates a new transaction with a unique id (*trans_id*).

PMFS-Log consists of an array of 64-byte *log entries*, where each log entry describes an update to the file system metadata. A log entry consists of a header and data portion, as shown in Figure 9(b). The 2-byte *gen_id* is a special field. For a log entry in PMFS-Log to be considered valid by recovery code, the gen_id field in the log entry must match a similar gen_id field in PMFS-Log metadata. PMFS-Log's gen_id field is incremented after every log wrap-around and after every PMFS recovery, thereby automatically invalidating all of the stale log entries.

To be able to identify valid entries in PMFS-Log, one of two requirements must hold: either PMFS must atomically append entries to PMFS-Log or the recovery code must be able to detect partially written log entries. One possible solution is to use two *pm_wbarrier* operations: append the log entry to PMFS-Log and make it durable, then atomically set a valid bit in the log entry before making the valid bit durable. Other approaches include using a checksum in the log entry header [86] or *tornbit RAWL* [101], which converts the logging data in to a stream of 64-bit words, with a reserved torn (valid) bit in each word. However, all these approaches have high overhead from either additional serializing operations (double barrier) or compute (checksum, tornbit RAWL). In PMFS, we fix the size of log entries to a single (aligned) cacheline (64 bytes) and exploit the architectural guarantee in the processor caching hierarchy that writes to the same cacheline are never reordered. For example, if A and B are two separate 8-byte writes to the same cacheline and in that order, then A will always complete no later than B. PMFS uses the gen_id field in log entry header as a valid field. When writing a log entry to PMFS-Log, gen_id is written last, before the log entry is made durable. To ensure this scheme works, we instruct the compiler not to reorder writes to a log entry.

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

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

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

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

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

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

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

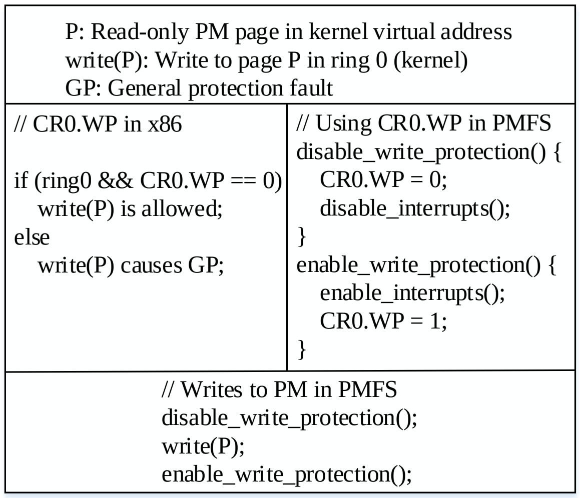
3.3.3 Write Protection

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

Table 2: Overview of PM Write Protection

	User	Kernel
User	Process Isolation	SMAP
Kernel	Privilege Levels	Write windows

Table 2 shows a brief overview of how PM write protection (from stray writes) works in the assumed processor architecture. The row name refers to the address space in which PM is mapped and the column name refers to the privilege level at which stray writes occur. Protection within a process (e.g., between multiple threads) is not covered here, but will be explored in the future. Protecting “kernel from user”, and “user from user” follow from existing isolation mechanisms based on privilege levels and paging, respectively. Also, “user from kernel” follows from use of Supervisor Mode Access Prevention (SMAP) feature in the processor [17]. When SMAP is enabled, supervisor-mode (i.e., ring 0 or kernel) accesses to the user address space are not allowed. This feature is important since memory-mapped I/O in PMFS provides the user-level applications with direct access to PM. Due to SMAP, the *mmap*'ed memory is protected from stray writes in kernel. Protecting "kernel from kernel", however, could be challenging, particularly if multiple OS components sharing the same kernel address space are allowed to write to PM. Since PMFS is the only system software that manages PM, the solution space is much simpler in our case.



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

```
// CR0.WP in x86

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

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

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

Figure 10: Write Protection in PMFS

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

Therefore, PMFS implements an alternate protection scheme, wherein it maps the entire PM as read-only during mount and upgrades it to writeable only for the sections of code that write to PM, called *write windows*. We could implement write windows by toggling write permission in the page table [23], but that would require an expensive global TLB shoot-down per write window. Instead, PMFS leverages the processor's write protect control (CR0.WP) [23] to implement the protection scheme described in Figure 10. PMFS exploits the following key attribute of CR0.WP: ring 0 or kernel writes to pages marked read-only in the kernel virtual address are allowed only if CR0.WP is not set. Therefore, by toggling CR0.WP, PMFS is able to open small temporal write windows without modifying the page tables, thereby avoiding expensive TLB shootdowns.

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

Architectures with more generic *protection key* mechanisms (e.g., *Itanium*® and *PowerPC*®) would have fewer such limitations with write protection in PMFS [61]. For instance, in Itanium, it is possible to group sets of pages, each assigned a protection key. The TLB entry for a page has a tag that associates the page with the assigned protection key that now resides in the processor's *protection key registers* (PKRs). For a given process, the access right for a page is determined by combining the access bits on the corresponding page table entry and the PKR. Since modifying a PKR does not necessitate a global TLB shootdown, protection keys can be used to implement write windows with very low overhead, as we do with CR0.WP. However, with protection keys, PMFS can control access to PM at a page granularity and without disabling interrupts.

3.3.4 Implementation

The protoype PMFS implementation for the Linux 3.9 kernel, available open source [12], is written as a Linux kernel module and has about 10K lines of code. PMFS uses and extends the eXecute In Place (XIP) interface in the Linux kernel. When the underlying storage is byte-addressable, XIP provides a set of VFS callback routines that offer a clean and efficient way to avoid the page cache and block device layer. In PMFS, the *read*, *write*, and *mmap* callbacks registered with VFS are modeled after *xip_file_read*, *xip_file_write*, and *xip_file_mmap*, respectively. PMFS also implements the callback routine (*get_xip_mem*) that is used by XIP to translate a virtual address to a physical address in PM.

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

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

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

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

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

Figure 8 shows the metadata updates for this operation (in dark shade). Figure 9(a) shows the corresponding log entries written during the transaction.

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

PMFS creates a new transaction for *write* only if it has to allocate new pages to complete the request. In this transaction, PMFS first allocates space for log entries in PMFS-Log, allocates the pages, and then writes data to PM using non-temporal instructions. Finally, PMFS logs and updates the inode's B-tree pointers and modification time, before writing a commit record to complete the transaction. Figure 8 shows the metadata updates for this operation (in light shade).

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

Freeing the inode involves atomically updating a set of fields in the inode, including the root of the inode's B-tree. By design, the inode fields for this operation are all in the same cacheline in PMFS. If RTM is available, PMFS uses a single 64-byte atomic write to free the inode. Otherwise, PMFS uses fine-grained logging. After successfully freeing the inode, all the inode data and metadata pages are returned to the allocator. Since the allocator structures are maintained in DRAM and reconstructed on failure, freeing the pages to the allocator doesn't require logging.

3.4 Testing and Validation

We have discussed earlier the need for extensions to the existing memory model to allow PM software to control *ordering* and *durability* of stores to write-back PM (§3.2).

However, such extensions to the memory model introduce the possibility of new types of programming errors. For instance, consider a PM software flow as shown in Figure 11. Starting at consistent state A, PM software performs two writes to PM (set $W_{A->B}$), dirtying two cachelines (L_1, L_2) in the process. For traditional block based storage, software

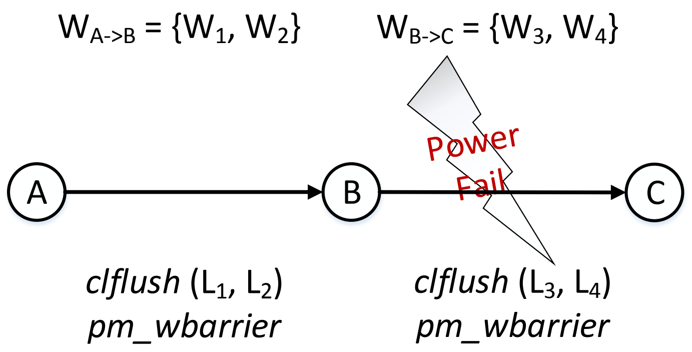


Figure 11: PM software flow

explicitly schedules IO to make these cachelines persistent at block granularity. However, for PM-based storage these cachelines can become persistent in arbitrary order by cacheline evictions outside of software control. Hence, extra care must be taken in enforcing ordering on updates to PM. Programmers today are not used to explicitly tracking and flushing modified cachelines in volatile memory. But, this is a critical requirement of PM software, failing which could cause serious consistency bugs and data corruption.

Testing for the correctness of PM software is challenging. One way is to simulate or induce failures (such as from power loss or crashes) and use an application-specific checker tool (similar to fsck for Linux file systems) to verify consistency of the data in PM. However, this method tests only the actual ordering of writes to PM in a single flow, even though many other orderings are possible outside the control of the PM software (e.g., due to arbitrary cacheline evictions).

To overcome this challenge, we built *Yat* [69], a hypervisor-based framework for testing the correctness of PM software. Yat uses a record and replay method to simulate architectural failure conditions that are specific to PM. In the first phase, Yat records a trace of all the writes, *clflush* instructions, ordering (*sfence*, *mfence*) instructions, and *pm_wbarrier* primitives executed while a test application is running. In the second phase, Yat replays the collected traces and tests for all the possible subsets and orderings of these operations, thereby simulating architectural failure conditions that are specific to PM. Yat includes an *fsck*-like tool to test the integrity of PM data in PMFS. For every simulated failure in PMFS testing, Yat automatically runs this tool, and provides diagnostic information if the

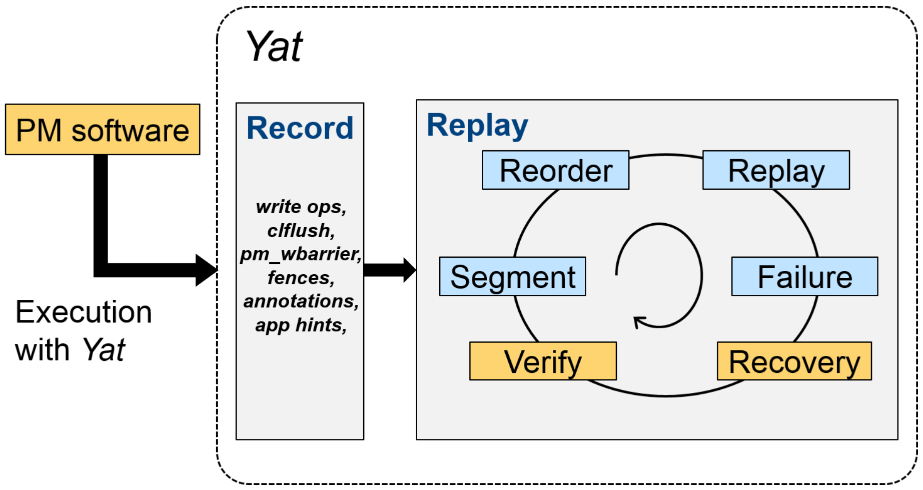


Figure 12: Overview of Yat's operation

verification fails. Figure 12 shows an overview of Yat's operational flow.

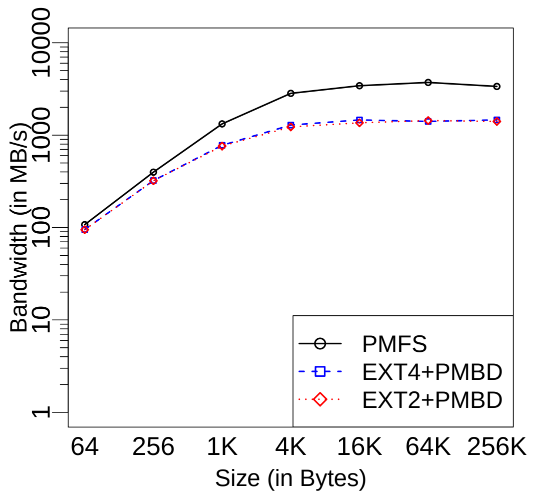
We used Yat in validation and correctness testing of PMFS. Maintaining consistency in PMFS is tricky due to the requirement that PM software track dirty cachelines and flush them explicitly before issuing *pm_wbarrier*. We found Yat to be extremely useful in catching several hard to find bugs in PMFS. Further discussion of Yat is beyond the scope of the work.

3.5 Evaluation

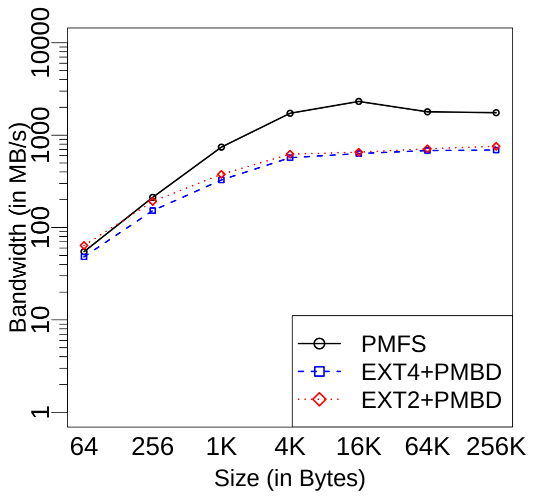
We now describe the experimental setup and baselines for the evaluation of PMFS (§3.5.1). Then we present results from a detailed evaluation with several micro-benchmarks (*fio*, file utilities) and application benchmarks (Filebench, Neo4j).

3.5.1 Experimental Setup

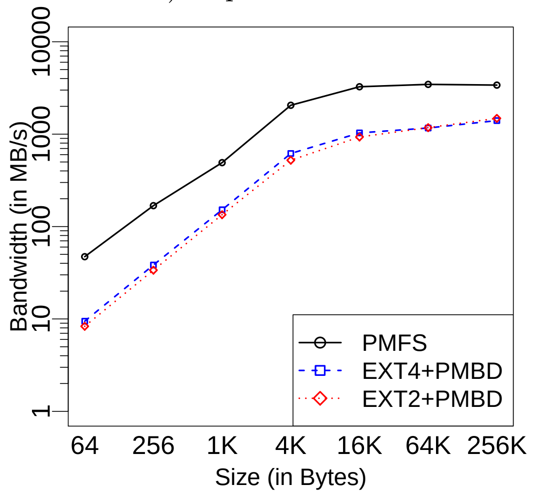
PM Emulator: System-level evaluation of PM software is challenging due to lack of real hardware. Publicly available simulators are either too slow and difficult to use with large workloads [90] or too simplistic and unable to model the effects of cache evictions, speculative execution, memory-level parallelism and prefetching in the CPU [11]. To enable the performance study of PM software for a range of latency and bandwidth points interesting to the emerging NVM technologies, we used the previously described PM performance emulator. For short, we call the emulator PM Emulation Platform (PMEP) in this chapter.



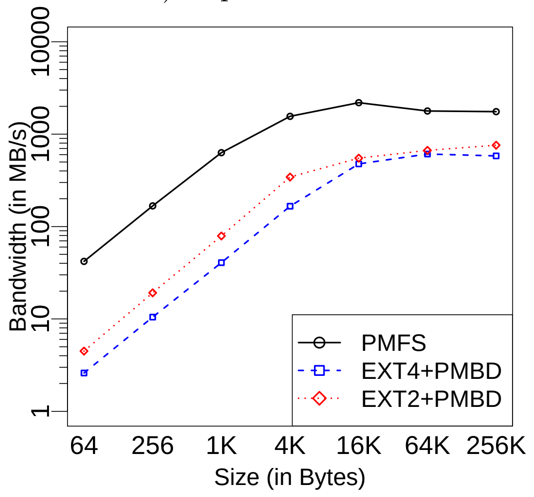
a) Sequential read



b) Sequential write



c) Random read



d) Random write

Figure 13: Evaluation of File I/O performance; X-axis is the size of the operation; Y-axis is the bandwidth in MB/s.

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

PMEP is implemented on a dual-socket Intel® Xeon® processor-based platform, using special CPU microcode and custom platform firmware. Each processor runs at 2.6GHz, has 8 cores, and supports up to 4 DDR3 Channels (with up to 2 DIMMs per Channel). The custom BIOS partitions available memory such that channels 2-3 of each processor are hidden from the OS and reserved for emulated PM. Channels 0-1 are used for regular

DRAM. NUMA is disabled for PM channels to ensure uniform access latencies. Unless specified otherwise, PMEP has 16GB DRAM and 256GB PM, for a 1:8 capacity ratio.

PMBD: For a fair comparison of PMFS with traditional file systems designed for block devices, it is important to not only use the same storage media (emulated PM in our case), but to also choose a representative and optimized implementation of block device. Previous work used a Linux RAMDISK-like block device that relies on OS VMM for memory management [52, 101, 51]. However, RAMDISK cannot partition between PM and DRAM and is subject to interference from OS paging. For this reason, we use Persistent Memory Block Driver (*PMBD*) [47], an open-source implementation that assumes partitioned PM and DRAM. For block writes, PMBD uses non-temporal stores followed by a *pm_wbarrier* operation (§3.3.2).

We compare PMFS with both ext4 and ext2 on PMBD. Ext2, a much simpler file system with no consistency mechanisms, provides an interesting baseline for comparison with PMFS. We first evaluate the performance of traditional file-based I/O (using *fio*) and file system operations (with file utilities and Filebench). We demonstrate the performance benefits of atomic in-place updates and fine-grained journaling in PMFS, by comparing the (raw) overhead of metadata consistency in PMFS with that in ext4 and BPFS [52]. Then we evaluate the performance of memory-mapped I/O with both micro-benchmarks (*fio*) and application workloads (Neo4j). Finally, we evaluate the performance of the PMFS write protection scheme.

3.5.2 File-based Access

3.5.2.1 File I/O

Many applications use traditional file I/O interfaces (*read* and *write* system calls) to access file data. In a block-based file system, this usually requires two copies, one between the block device and the page cache, and one between the page cache and the user buffers. PMFS requires only one copy, directly between the file system and the user buffers. In Figure 13, we show a comparison of raw *read* and *write* performance using *fio*, a highly tunable open source benchmarking tool for measuring the performance of file I/O (*read*, *write*, *mmap*) for various access patterns [5]. We measure the performance of a single *fio* thread reading from or writing to a single large file (64GB). In case of the *write* tests, we flush data explicitly (using *fdatasync*) after every 256KB bytes.

As seen in Figure 13, PMFS performs better than both ext2 and ext4 for all the tests, with improvements ranging from 1.1× (for 64B sequential reads) to 16× (for 64B random writes). The drop in performance of writes for sizes larger than 16KB is due to the use of non-temporal instructions. These instructions bypass the cache but still incur the cache-coherency overhead from snooping CPU caches to invalidate the target cacheline. At larger sizes, this overhead is large enough to congest the write-combining buffers and cause a drop in performance. We found that the performance using non-temporal instructions is still better than if normal cacheable writes were used along with (optimized) cache flushes.

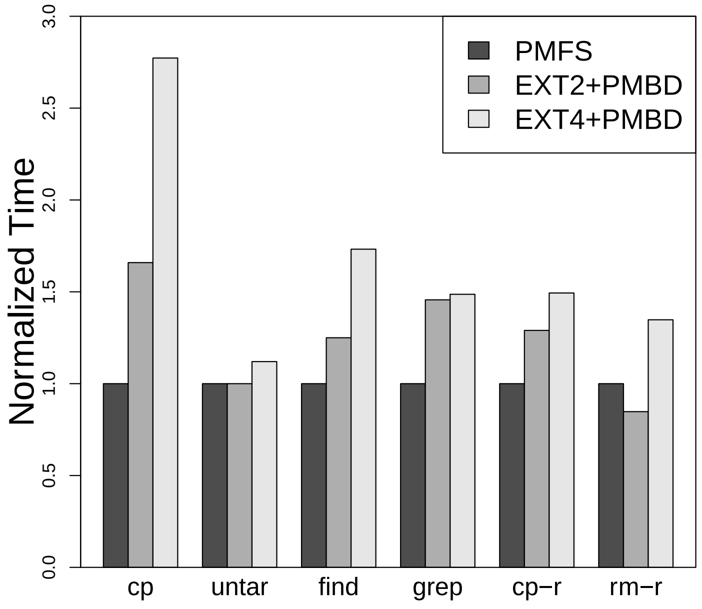
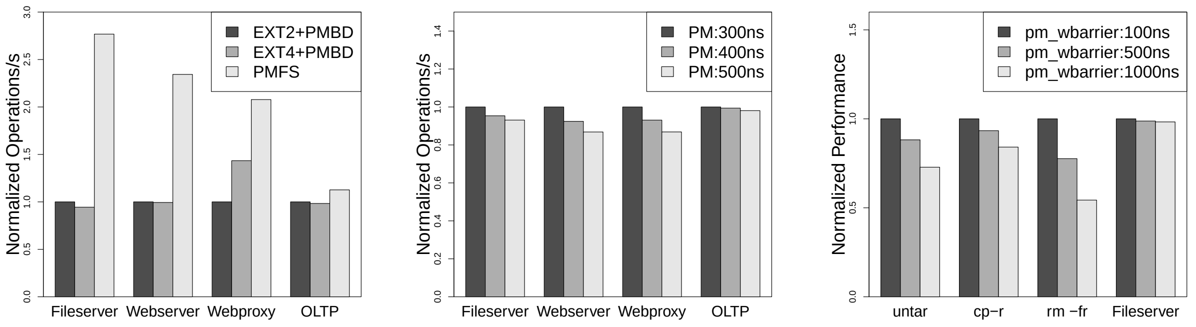


Figure 14: Evaluation of File utilities

3.5.2.2 File Utilities

Next we evaluate the performance of common file and directory operations. We perform these tests with a tarball of Linux Kernel sources in the file system under test. *cp* creates a copy of the tarball in the same directory. *untar* uncompresses and unpacks the tarball. *find* searches for an absent file in the uncompressed directory and *grep* searches for an absent pattern in the same directory. Finally, *cp-r* copies the entire directory and *rm-r* deletes the old copy. We measured the time taken for each of these operations. Figure 14 shows the relative performance. In all the tests other than *rm-r*, PMFS is faster than both ext2 and ext4, even while providing same consistency guarantees as ext4. PMFS performance gains, ranging from 1.01× (for *untar*) to 2.8× (for *cp*), are largely attributable to fewer copies. However, PMFS sees significant benefits from optimized consistency and atomic in-place updates as well, as seen with metadata intensive *rm-r* (in comparison with ext4). PMFS shows very little improvement with *untar* because *untar* involves only sequential writes from a single thread and is not very sensitive to latency.

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



a) Comparison with Ext2/Ext4 b) Effect of PM latency c) Effect of *pm_wbarrier* latency

Figure 15: Evaluation of Filebench performance

3.5.2.3 PMFS Consistency

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

Atomic in-place updates: PMFS exploits atomic updates to avoid logging whenever possible. To illustrate the benefits of 16-byte atomic updates, we measured the performance of a *write* system call that uses 16-byte atomic updates as described in §3.3.4. Using atomic updates, instead of fine-grained logging, speeds up the operation by 1.8×.

For the evaluation of 64-byte atomic updates, since *RTM* is not available in PMEP, we measured the RTM performance on Intel® Core™ i7-4770 processor [21]. Then we simulated the RTM performance in PMFS by replacing fine-grained logging with timed loops based on the above measurement. For deleting an inode (as described in §3.3.4), using 64-byte atomic updates is 18% faster than fine-grained logging.

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

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

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

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

3.5.2.4 Filebench

Filebench is a file system and storage benchmark that can simulate a variety of complex application workload I/O patterns [4]. In Figure 15, we present results from the evaluation of four multi-threaded application workloads from the Filebench suite.

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

In each case, we used the default number of threads — *Fileserver* has 50 threads;

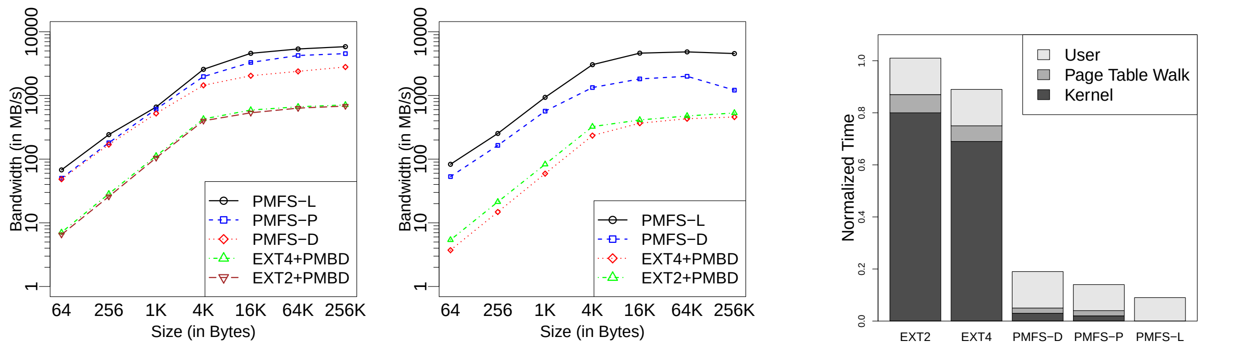
Webserver has 100 threads; *Webproxy* has 100 threads; and *OLTP* has a single thread doing log writes, 10 database writer threads, and 200 database reader threads. In each case, we scaled the dataset to at least 32GB to ensure that it does not fit in the page cache.

Figure 15(a) compares these workloads for PMFS, ext4, and ext2. PMFS performs better than both ext4 and ext2 for all the workloads, with gains ranging from 1.15$\times$ (for *OLTP*) to 2.9$\times$ (for *Fileserver*), confirming that PMFS does not have any obvious scaling issues for multi-threaded applications. As with the micro-benchmarks, the performance improvements with PMFS are due to fewer copies and optimized consistency.

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

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

Effect of *pm_wbarrier* latency: PMFS uses the *pm_wbarrier* primitive to enforce durability of stores evicted from the cache. Figure 15(c) shows the performance of several file utilites (described above) and one of the Filebench workloads (*Fileserver*), as the latency of *pm_wbarrier* primitive is varied from 100ns to 1000ns. The drop in performance directly corresponds to the rate of *pm_wbarrier* operations for the workload. For instance, we measured a *pm_wbarrier* rate of over 1M/sec for metadata intensive *rm-r*, which shows the largest drop in performance (85%). This observation justifies the optimizations described in §3.3.2 to reduce the number of *pm_wbarrier* operations. Without these optimizations, the drop in performance would have been double that observed. *Fileserver*, on the other hand, has a *pm_wbarrier* rate of only around 40K/sec and doesn't show any noticeable drop in performance. We observed similarly low rates and results for other Filebench workloads.



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

Figure 16: Evaluation of memory-mapped I/O performance

3.5.3 Memory-mapped I/O

Memory-mapped I/O (*mmap*) in PMFS eliminates copy overheads by mapping PM directly to application's address space. In Figure 16(a)(b), we use *fio* to compare the performance of random reads and writes with *mmap*'ed data. We use *fallocate* to create a single large (64GB) file for the *mmap* test, thereby allowing the use of large (1GB) pages with *mmap* (§3.3.1). PMFS-D refers to the default case where *mmap* is done using regular 4KB pages. PMFS-P is exactly the same as PMFS-D, except for the use of MAP_POPULATE option with *mmap*. MAP_POPULATE causes the *mmap* system call to pre-populate the page tables at the time of *mmap*, thereby eliminating the runtime overhead of page faults. Finally, PMFS-L refers to the configuration where *mmap* is done using large (1GB) pages.

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

Compared to PMFS-D, PMFS-L is 1.3× to 2.3× better for random reads and 1.6× to 3.8× better for random writes. Performance gains with PMFS-L are due to several reasons: fewer page faults, better TLB efficiency, and shorter page table walks. In Figure 16(a), we quantify the overhead of page faults alone with PMFS-P. By eliminating the page faults, PMFS-P is able to improve the performance by 1.0× to 1.8× over PMFS-D. However, PMFS-L is still faster than PMFS-P by 1.1× to 1.4× due to better TLB efficiency and shorter page table walks. Figure 16(c) shows a more detailed breakdown of *mmap* performance, using 4KB random reads as an example. “Kernel” refers to the time spent in ring 0, either servicing page faults or copying data between DRAM and PMBD. “Page Table Walk” is the time that hardware spends walking the page table structures due to a TLB miss. “User” represents the time spent doing actual work in the application. Ext2 and ext4 spend only about 10% of the time doing useful work. As expected, PMFS-L has negligible overhead from page faults and page table walks, and therefore spends most of the time in “User”. Counter-intuitively, PMFS-L also achieves a reduction in the actual application work. This behavior is attributable to the fact that PMFS-L has fewer instruction and LLC misses, because the fewer and shorter page walks in PMFS-L do not pollute the cache.

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

3.5.3.1 Neo4j Graph Database

We use Neo4j to evaluate the benefits of PMFS to an unmodified application that uses memory-mapped I/O to access storage. Neo4j is a fully ACID (NoSQL) graph database aimed at connected data operations [9], with the database itself comprised of multiple store files. By default, Neo4j accesses these files by *mmap*’ing them, using Java’s NIO API [10].

Neo4j supports both embedded (stand-alone) and client-server applications. For the evaluation in this chapter, we wrote a sample Neo4j embedded application to measure the performance of common graph operations. We created a Neo4j graph database from the Wikipedia dataset [14]. Titles in the dataset are the nodes in the graph and links to the titles are the edges or relationships. The resulting graph has about 10M nodes and 100M edges. Both the nodes and edges have no other properties. We ran the following tests on this graph database :

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

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

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

Since the size of the entire Neo4j database is less than 8GB, we limit the amount of DRAM in the system to 4GB for these tests, thereby ensuring that the database does not completely fit in the page cache. We ran these tests three times and measured their average run times. There was very little variation in the run times across the different runs. Figure 17 shows a comparison of PMFS with ext2 and ext4 over PMBD. In all the tests, PMFS is faster than both ext2 and ext4, with performance gains ranging from 1.1× (for *Insert*) to 2.4× (for *Query*). This improvement is attributable to the fact that memory-mapped I/O in PMFS has no copy overhead.

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

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

Although the performance gains, particularly for *Query*, are impressive for an unmodified application, they could be further improved by re-designing applications as suggested

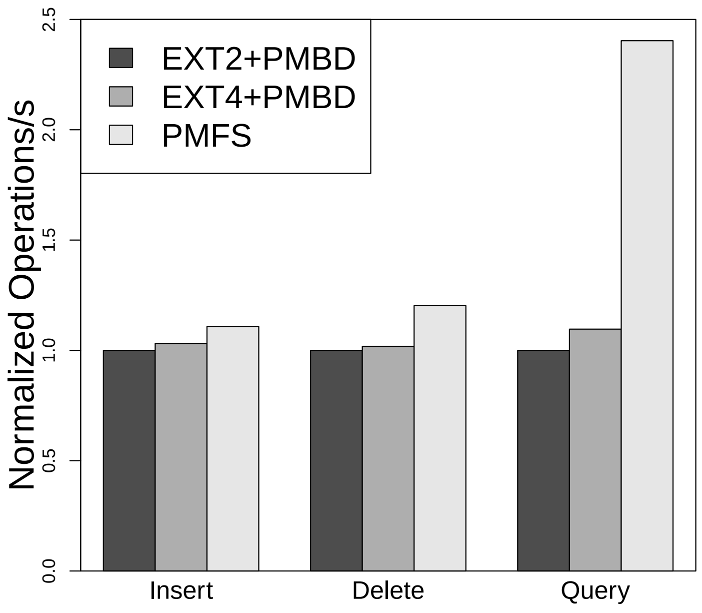
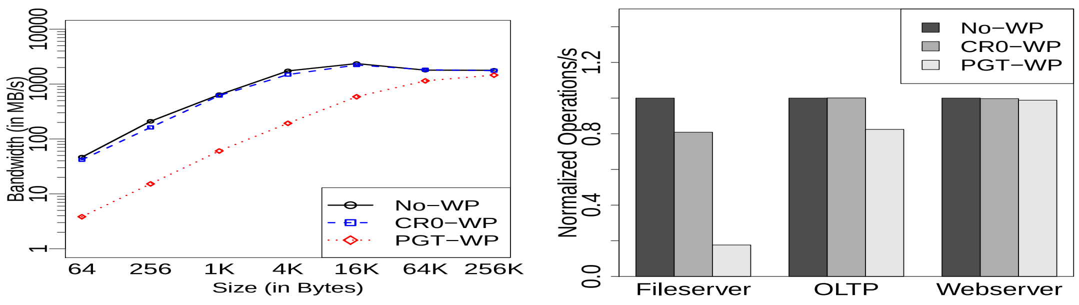


Figure 17: Evaluation of Neo4j performance

by prior work [101, 51] and using PMFS's memory-mapped I/O to access PM.

3.5.4 Write Protection

As described in §3.3.3, PMFS protects PM from stray writes in the OS using an existing processor write control feature (CR0.WP). In this section, we evaluate the performance overhead of this write protection scheme.



a) *fio* file write b) Filebench

Figure 18: Evaluation of PMFS write protection overhead

In Figure 18, we compare the write protection scheme in PMFS (CR0-WP) with the baseline case of no write protection at all (No-WP), and an alternate implementation that uses page table permissions (PGT-WP) (§3.3.3). As shown in Figure 18(a), CR0-WP is close to No-WP in performance and 1.2× to 11× faster than PGT-WP for file writes.

Figure 18(b) shows a similar comparison for select Filebench workloads. For *Fileserver*, a multi-threaded workload with writes from several threads, CR0-WP is 23% slower than No-WP, but still 5× faster than the alternate PGT-WP scheme. The slowdown with CR0-WP is because writes to a processor control register (to toggle CR0.WP) are serializing operations with latency close to an uncacheable (UC) write.

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

3.6 Summary

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

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

CHAPTER IV

DATA TIERING IN HYBRID MEMORY SYSTEMS

In the previous chapter, we discussed a light-weight file system (PMFS) that provides the system software layer to efficiently manage PM. PMFS not only improves the performance of legacy applications but also engenders the development of new runtimes and applications [101, 51, 99]. With PMFS, the focus is on the use of PM as fast storage and on applications that rely on PM's persistency property, with special emphasis on the challenges associated with achieving consistency in PM software. In this chapter, we explore the other key usage model intended for PM – as cheaper, high capacity volatile memory. To accentuate the fact that persistence is irrelevant to this usage, we refer to PM as NVM throughout the chapter.

The primary motivators for using NVM as main memory are the memory-based data center applications such as key-value stores, in-memory databases, and analytics frameworks. These applications require increasingly large memory capacities, but face the challenges posed by the inherent difficulties in scaling DRAM and also the cost of DRAM. Future systems will address these demands with heterogeneous memory architectures coupling DRAM with high capacity, low cost, but also lower performance, NVM. Data center operators are bound to ask whether this model for the usage of NVM to replace the majority of DRAM memory leads to a large slowdown in their applications? It is crucial to answer this question because a large performance impact will be an impediment to the adoption of such systems.

We present a thorough study of representative applications – including a key-value store (MemC3), an in-memory database (VoltDB), and a graph analytics framework (GraphMat) – on a platform that is capable of emulating a mix of memory technologies. Our conclusions are that it is indeed possible to use a mix of a small amount of fast DRAM and large amounts of slower NVM without a proportional impact to an application's performance. The caveat is that this result can only be achieved through careful placement of data structures. The

Table 4: Comparison of new memory technologies [90, 18]. NVM technologies include PCM and RRAM [90, 18]. Cost is derived from the estimates for PCM based SSDs in [67]. Since writes to write-back cacheable memory are posted, the effect of NVM's slower writes is lower bandwidth to NVM. For reads, latency is the critical metric.

Parameter	**DDR-DRAM**	**NVM**
Capacity per CPU	100s of GBs	Terabytes
Read Latency $\times$	$1\times$	$2\times$ to $4\times$
Write bandwidth	$1\times$	$\frac{1}{8}\times$ to $\frac{1}{4}\times$
Estimated cost	$5\times$	$1\times$
Endurance	10^{16}	10^{6} to 10^{8}

contribution of this work is the design and implementation of a set of libraries and automatic tools that enables programmers to achieve optimal data placement with minimal effort on their part.

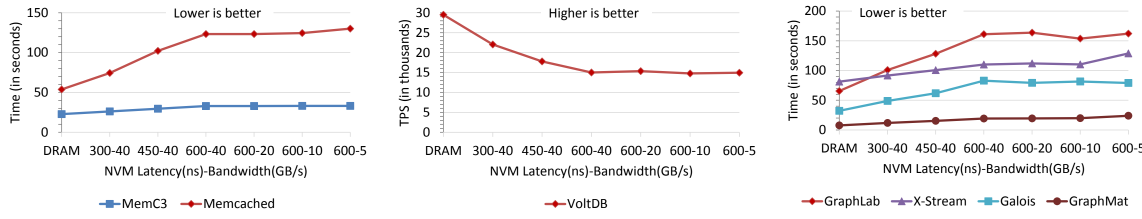
With such guided placement and with DRAM constituting only 6% of the total memory footprint for GraphMat and 25% for VoltDB and MemC3 (remaining memory is NVM with $4\times$ higher latency and $8\times$ lower bandwidth than DRAM), we show that our target applications demonstrate only a 13% to 40% slowdown. Without guided placement, these applications see $1.5\times$ to $5.9\times$ slowdown on the same configuration. Based on a realistic assumption that NVM will be $5\times$ cheaper (per bit) than DRAM, this hybrid solution also results in $2\times$ to $2.8\times$ better performance/\$ than a DRAM-only system.

4.1 Background

Datacenter applications like key-value stores [43, 83], in-memory databases [37], and data analytics [98, 60] are being used to handle exponentially growing datasets but cannot tolerate the performance degradation caused by spilling their workloads to disk. On the other hand, DRAM density (and cost) is not scaling due to physical limitations [73, 90], meaning that continuing to fit growing datasets in DRAM will be unviable in the future.

To address this challenge, industry is exploring new non-volatile memory technologies (or NVM) [63, 90, 35]. These memory technologies are positioned between DRAM and secondary storage (such as NAND flash), both in terms of performance and cost. Table 4 quantifies this tradeoff. NVM provides approximately $5\times$ the capacity at the same cost as DRAM for less than an order of magnitude reduction in performance (up to $4\times$ higher latency and 8× lower bandwidth).

The likely deployment of NVM is in systems that have a mix of DRAM and NVM. Most of the memory in such systems will be NVM to exploit their cost and scaling benefits, with a small fraction of the total memory being composed of DRAM. Application developers and data center operators looking at deploying DRAM and NVM based hybrid memory systems are justifiably concerned about the performance overheads due to the vast majority of the memory being several times slower than traditional DRAM.



a) Key-value stores b) In-memory databases c) Graph analytics

Figure 19: Effect of memory latency and bandwidth on the performance of large in-memory applications.

To illustrate the effect of NVM's characteristics on an application's performance, consider three different applications: (i) a Facebook like trace [43] running on the popular Memcached and MemC3 key value stores, (ii) standard TPC-C benchmark on the VoltDB in-memory database, and (iii) a set of graph analytics frameworks running the Pagerank algorithm [46] on the Twitter dataset [68]. DRAM's latency and bandwidth on this system are 150 ns and 40 GB/s, respectively. NVM technology is evolving and will likely exhibit a range of performance characteristics, depending on the details of the technology and the controller design. We therefore vary the parameters of our NVM emulator to reflect this scenario by first varying the latency from 300ns to 600ns, while keeping the bandwidth the same as DRAM and then varying the the bandwidth from 40 GB/s (same as DRAM) down to 5 GB/s (12.5% of DRAM), while keeping latency fixed at the worst case of 600ns. Figure 19 underlines that the performance impact across *all* application categories can be as much as 6× for NVM - that has 4× the latency and $\frac{1}{8}\times$ the bandwidth of DRAM.

The first contribution of this work is a quantitiative study demonstrating that a large portion of this slowdown can be removed by judiciously placing some application data structures in the small amount of DRAM available in a hybrid memory system.

The second contribution of this work is showing that the performance impact of placing a data structure in a particular type of memory depends on the access pattern to it. This dependence is due to different memory access patterns – sequential, random and pointer chasing – having very different effective latencies on modern superscalar out-of-order processors.

The third contribution of this work is the design and construction of a memory management infrastructure (called X-Mem) whose API allows programmers to easily map objects to data structures that they belong to at the time of object allocation. The implementation of X-Mem then places objects of different data structures in disjoint regions of memory.

The fourth and final contribution of this work is a profiling tool that automatically identifies the access pattern to a data structure and automates the task of mapping memory regions of that data structure to DRAM or NVM in order to optimize performance.

The final result is optimal performance without programmers needing to reason about their data structures, processor behavior or memory technologies. Using a small amount of DRAM (only 6% of the total memory footprint for GraphMat, and 25% for VoltDB and MemC3) the performance of these applications is only 13% to 40% worse than their DRAM based counterparts, but at much reduced cost – i.e., with 2× to 2.8× better performance/$. Intelligent data structure placement is the key to achieving this result, without which these applications perform 1.5× to 5.9× worse on the same configuration, depending on the actual data placement.

Based on these results, we contend that a *hybrid* architecture offering both DRAM and NVM, where software has direct access to its different types of memories is indeed a realistic approach to building systems that can tackle the exponentially growing datasets at reasonable performance and compelling cost.

4.2 The case for X-Mem

Methods for memory tiering can be broadly classified as application-driven or -agnostic, as shown in Table 5. Application-driven techniques offer explicit information, provided either

Table 5: An overview of memory tiering techniques.

	Static	Adaptive
App-driven	NUMA API	X-Mem API
App-agnostic	NA	Migration

as hints or via strict interfaces (e.g., NUMA API [31]), to enable runtimes or system software to make appropriate decisions regarding data placement. Agnostic methods require no application involvement, relying on system software to transparently monitor data access patterns and perform informed data placement (e.g, AutoNUMA [16] or paging [39]). Unfortunately agnostic methods also suffer from a number of pitfalls that motivated us to proceed in a different direction with X-Mem: (i) monitoring and placement at page granularity can lose semantic information about application data, an example being the allocation of both important database indices and less frequently used database tuples from the same page. Use of large hardware pages only worsens the situation; (ii) agnostic methods reacting to online monitoring depend in accuracy on monitoring frequency, but current architectures have limited support for doing so with small overhead and high effectiveness (e.g., hot page detection via access and dirty bits [8, 70]).

Another key design choice for X-Mem was whether to make placement decisions *statically* or *adaptively*. In static tiering, applications dictate data placement using strict interfaces like the NUMA API [31], but this requires applications to know the types and sizes of available physical memories and fully understand their properties. Applications must also implement mechanisms like migration to react to runtime change, thus adding complexity and reducing portability.

X-Mem, on the other hand, determines the data placement adaptively at runtime. Applications use the X-Mem API to semantically "tag" their data at the time of allocation, and to specify their relative importance (or "priority") to the application's performance (§4.3.1). The X-Mem runtime, then, manages the allocation of virtual memory while preserving the semantic (tag) information and transparently manages the actual physical mappings based on the priority of data and the sizes of memories (§4.3.2). Additionally, X-Mem provides tools that can assist in automatically determining the data priorities based on their access

```
void* xmalloc(int tag, size_t size);
void xfree(size);
void xmem_priority(int tag, int prio);
```

Figure 20: X-Mem API

characteristics, the properties of the physical memories, and the placement policy (§4.5). While the principles described here are applicable to a system with any number and types of physical memories, we focus on a system with only two types – DRAM and slower NVM.

4.3 X-Mem Allocator

4.3.1 Allocator API

X-Mem can improve application performance by automatically placing data-structures in appropriate memory types. However, in order to do that it must distinguish between objects belonging to different data structures. To solve this problem, X-Mem provides an expanded memory allocation interface that allows programmers to specify a tag during memory allocation. By using a unique tag for each data structure programmers provide the necessary information for the X-Mem runtime to map each object to its data structure.

Figure 20 shows the X-Mem API. The allocation routine (*xmalloc*) is the same as the standard *malloc* memory allocator API, except for the addition of a data classifier (or *tag*). In our current implementation, if an application wishes to change the tag of data after allocation, it has to explicitly allocate new data with the required tag and copy the old contents.

xmem_priority enables applications to explicitly assign priority values to the tags. Memory belonging to higher priority tags is preferred for placement in faster memory (DRAM). X-Mem applies automatic placement to break ties between data structures of the same priority. By default, all data structures have the same priority leaving X-Mem as the sole authority that determines placement.

X-Mem is designed to tier in-memory application data that is being scaled up over time. Such data is usually dynamically allocated on the heap, and therefore the X-Mem API is targeted at dynamic memory management. Statically allocated data structures and the program stack continue to be allocated in DRAM by the default runtime mechanisms.

4.3.2 Allocator Internals

Since the purpose of this work is placement in hybrid memory systems rather than optimizing memory allocators for NVM, X-Mem internally uses a standard memory allocator (*jemalloc* [?]) to manage memory. Unlike normal usage however, X-Mem instantiates a new jemalloc allocator for each tag that it sees. Each tag is therefore mapped to a unique allocator and all memory allocation and deallocation for objects with a particular tag are redirected to the memory allocator associated with the tag.

X-Mem assumes ownership of a large contiguous range in the application's virtual address space (using *mmap*). This virtual address space is managed by a special "root" allocator. An allocator used to manage memory for a particular tag (data structure) grows or shrinks its memory pool in units of a region that is allocated (deallocated) from the root allocator. Each region has a fixed size that is a power of two - for this work we use regions that are 64 MB in size. At any given point, a region in use is assigned to exactly one jemalloc instance. *A*ll objects in a region therefore have the same tag. Unused regions belong to the root allocator.

Some metadata (§4.5) is maintained for each memory region to guide placement decisions. A hash table of regions indexed by the start address of the region allows easy access to region metadata and the allocator owning the region. It is therefore easy to map an address to the start address of its containing region by masking the appropriate number of bits - the lower 26 bits for 64 MB regions. The start address of the region is used to index into the hash table and allows locating the appropriate allocator for `xfree` operations and region metadata. The metadata overheads are negligible compared to the 64 MB region size.

4.3.3 Mapping Regions to Memory Types

X-Mem maps regions to memory types based on its automatic placement algorithm that we describe later (§4.5). We expose NVM as a memory-only NUMA node separate from DRAM and use the *mbind* system call to restrict physical memory allocations for a region to a particular NUMA node – i.e., to DRAM or NVM.

At runtime, as new memory regions are added to the program, the newly added regions can displace memory regions from DRAM. This displacement causes X-Mem to trigger migration to change the type of physical memory backing an existing virtual region from DRAM to NVM. We simply reissue an `mbind` call with the desired mapping for the region, leaving the operating system (Linux) to do the heavy lifting of actually migrating pages and updating page tables synchronously. We note that this process leaves the application visible virtual addresses unchanged, allowing the application to continue running unmodified. Also, this design results in X-Mem being entirely restricted to user space making it portable and easy to deploy as a library.

Figure 21 shows an example of X-Mem's data placement and migration policies. The application in this case allocates three data structures (tree, hash, and buffer) using the X-Mem API. Memory allocations for each of these data structures is managed by a separate jemalloc instance. The tags used by the application (T_1, T_2, and T_3) are assigned internal priorities (§4.5) used to decide placement.

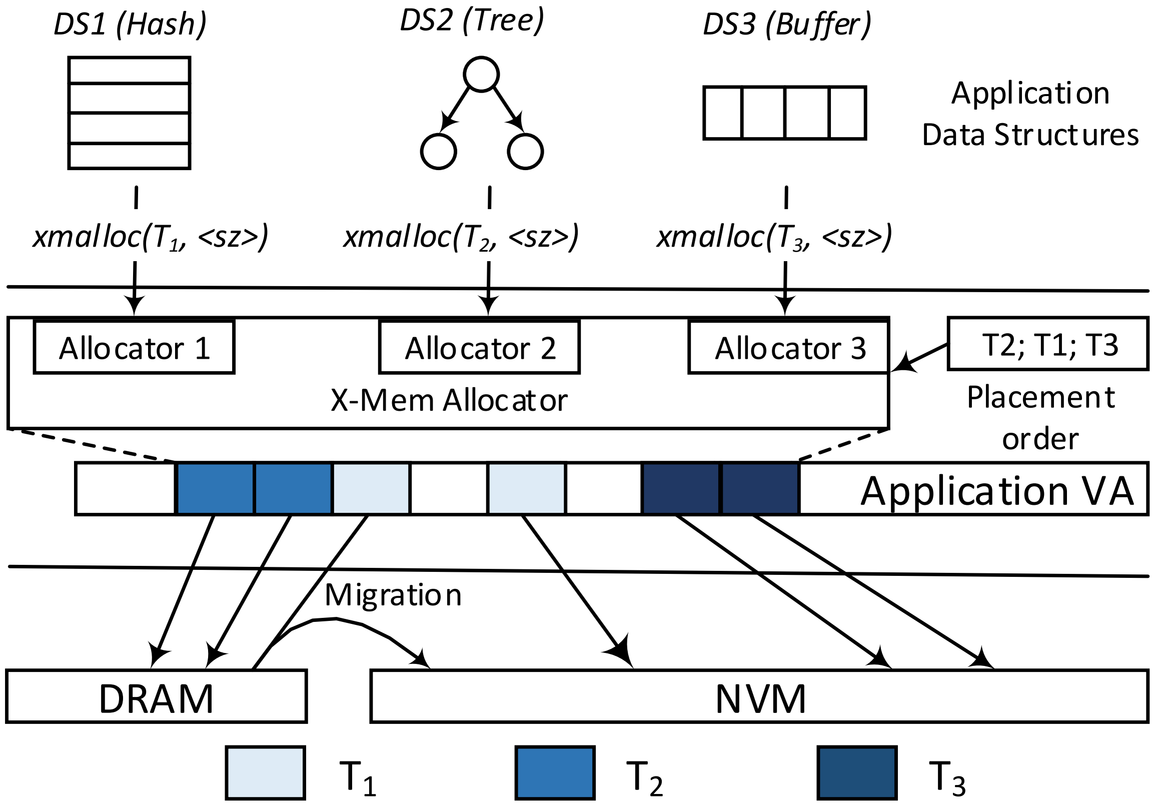


Figure 21: X-Mem design

4.3.4 Implementation

X-Mem is implemented as a user space library on standard Linux. It leverages operating system APIs for:

- Direct load/store access to each of the physical memories.
- Control over the type of physical memory backing its virtual addresses.
- Migrating virtual memory from one type of physical memory to the other.

With modern operating systems (such as Linux), there are several ways to realize these needs. One option is to extend an in-memory file system to manage multiple physical memories and use memory-mapped files that bypass the operating system page cache for direct access to the physical memory [57]. However this model requires complex additions to the operating system to support migration of pages from one type of memory to the other.

We instead chose to expose NVM as a memory-only NUMA node separate from DRAM. In this hybrid system, X-Mem uses System V shared memory segments to allocate memory and the *mbind* system call to restrict memory allocations for a given range of virtual addresses to a particular NUMA node – i.e., to DRAM or NVM.

Each instance of X-Mem creates a large shared memory segment equal in size to the maximum amount of memory that can be allocated through the X-Mem APIs. The virtual memory from this shared memory segment is managed by X-Mem's root jemalloc allocator. Based on application usage, the instances of jemalloc supporting different tags grow or shrink at run time by allocating and deallocating regions from the root X-Mem allocator.

4.4 Memory Access Time Model

X-Mem supports automatic assignment of virtual memory regions to physical memory types in order to optimize performance. Optimum placement is based on a *model* of memory access time in superscalar processors that we now describe.

For working sets that do not fit in the last level cache, every accessed cacheline is equally likely to result in a miss. For brevity in notation, we consider the case of N data structures in a program. Let S_i be the stall time for each access to data structure i and let C_i be the count of the number of accesses made to the data structure from a single thread. The average time for a memory access:

$$A = \frac{\sum_{i=1}^{N} C_i * S_i}{\sum_{i=1}^{N} C_i} \tag{1}$$

The stall for each access to the data structure clearly depends on the latency to memory, which in turn is determined by the type of memory that the data structure is placed in. However that piece of information alone is insufficient.

The stall latency of each access in a modern superscalar processor is *not* the same as the time to access memory. The processor can hide the latency to access memory by locating multiple memory requests in the instruction stream and then using out-of-order execution to issue them in parallel to memory via non-blocking caches [76]. In addition, microprocessors incorporate prefetchers that locate striding accesses in the stream of addresses originating from execution and prefetch ahead. As a result the *effective* latency to access memory can be much smaller than the actual physical latency for certain access patterns.

We illustrate these effects using a set of microbenchmarks that consider three possible access patterns:

- Random access: The instruction stream consists of independent random accesses. The processor can use its execution window to detect and issue multiple requests simultaneously.

- Pointer chasing: The processor issues random accesses but the address for each access depends on the loaded value of a previous one, forcing the processor to stall until the previous load is complete.

- Streaming: The instruction stream consists of a striding access pattern that can be detected by the prefetcher to issue the requests in advance.

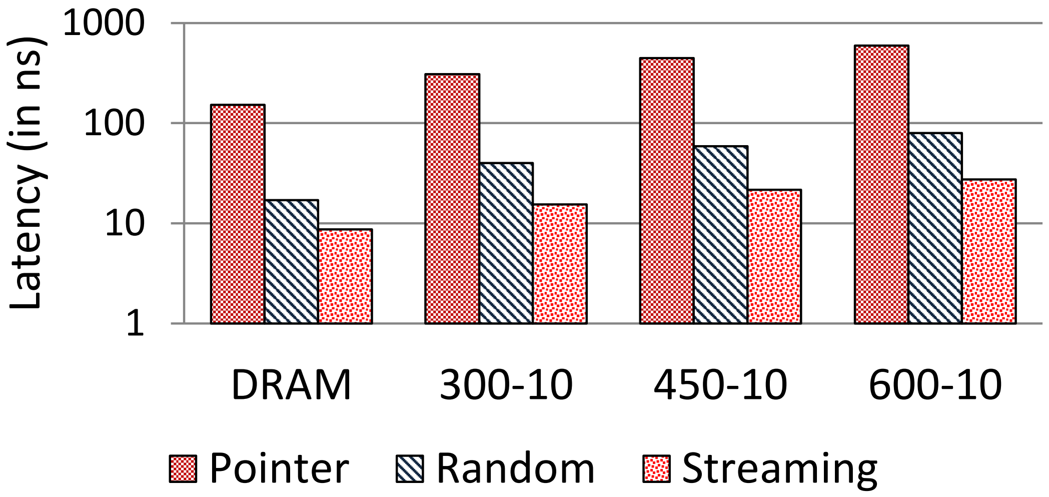


Figure 22: Memory read latency for various access patterns.

In Figure 22 we vary the latency and bandwidth characteristics of memory in the emulator. Pointer chasing experiences the maximum amount of stall latency, equal to the actual physical latency to memory. In the case of random access, out-of-order execution is effective at hiding some of the latency to memory. In the case of streaming access, the prefetcher is even more effective at hiding latency to memory.

The microbenchmarks give us an effective way to measure the average stall latency for different types of accesses and data structures. We use $L(p, m)$ to indicate the stall latency for accesses of pattern p to memory of type m.

4.5 Profile Guided Placement

Optimizing placement in X-Mem works as follows. First, we use a profiler to determine the relative frequency of different types of access to each data structure (tag). Second, during production runs the profiling data is used to maintain a list of memory regions sorted by preference for their placement in faster memory (DRAM). As many regions as possible from the head of the list are mapped to DRAM leaving the rest in slower NVM. We first describe our profiler and then describe how the profile data is used by X-Mem at runtime.

4.5.1 Profiler

We use a profiling tool that operates in conjunction with the X-Mem memory allocator described in §4.3. For each data structure tag, the tool classifies accesses to its regions as one of random, pointer chasing and streaming, and counts the number of accesses of each type.

The profiler uses PIN [74] to intercept memory accesses and record (for each access) the address that was accessed. This information is stored in a buffer and processed periodically using the post-processing algorithm shown in Figure 23. We first sort the accesses, effectively grouping accesses to the same region together. We iterate through the buffer searching for access pairs where the value at the location of the first access equals the address of the location of the second access. We interpret this pattern as pointer chasing to the second location and attribute the stall latency to the second access. We also maintain a sliding window (of size 8 by default) over elements of the sorted buffer to detect streaming accesses where consecutive elements of the window differ by the same amount. We also count the total number of accesses to a memory region. Subtracting the number of streaming and pointer chasing accesses from this total number gives us the the number of random accesses.

In the post-processing algorithm we eliminate accesses to the same cacheline assuming that the set of accesses in the buffer would fit in cache and therefore repeated accesses to the same cacheline do not go to memory. The cost of post-processing the buffer grows slowly with the size n of the buffer as $O(n\log(n))$ and therefore we are able to use a large buffer of size 4KB to amortize the cost of switching the context from running the application to running the analysis routine.

As described in §4.3 we maintain metadata for each memory region and are able to access the metadata starting from an address via a hash index. This metadata includes the counters we use in Figure 23.

After the program terminates, we roll up the counters from all memory regions for a particular data structure into a per-memory region average that is attached to the parent data structure (tag). This average consists of three counters – one each for pointer chasing, random and streaming accesses. Since data structures can have different kinds of accesses (such as traversing a linked list and then accessing fields of a particular node), we do not attempt any further classification of the data structure. We use $F_i(p)$ to denote the fraction of accesses of pattern p to data structure i as read from these counters.

```
Input: A buffer of addresses

Sort the buffer by address
sliding_window = ()
for each access i in sorted buffer
  if exists access j s.t. j.address == *i.address
    and Cacheline(j.address) != Cacheline(i.address)
    region[j.address].pointer_chasing++
  if Cacheline(i.address) !=
    Cacheline(sliding_window.end().address)
    Advance sliding_window to include i
    if sliding_window is a strided access pattern
      Region[i.address].streaming++
    Region[i.address].accesses++
    if i is a read
      Region[i.address].reads++
    else
      Region[i.address].writes++
```

Figure 23: Classifying Access Type

4.5.2 Runtime

The runtime estimates Equation 1 using the profiling information. Let $T(r)$ be the the type of memory in which memory region r is placed and let $D(r)$ be the parent data structure of r. We estimate the memory access time for a given configuration as follows:

$$\hat{A} = \sum_{r \in \text{Regions}} \sum_{p \in \text{Patterns}} F_{D(r)}(p) L(p, T(r)) \tag{2}$$

We use the profiling run estimate of the average number of accesses of a particular pattern to a memory region, and then multiply it by the appropriate latency of that access pattern.

The placement algorithm aims to minimize the quantity $\hat{A}$ by choosing the right assignment of memory regions to memory types, while respecting the capacity constraints of each type of memory. Since this dissertation deals with only two different types of memory (standard DRAM and slower NVM), we cast this problem as follows.

We start by assuming that all regions are in NVM. Given a fixed amount of faster DRAM, moving some memory regions to DRAM would reduce the average access time. Hence the problem is to choose a set of memory regions moving which maximizes the gain.

For any memory region r the benefit of moving to DRAM is:

$$B(r) = \sum_{p \in \text{Patterns}} F_{D(r)}(p)[L(p, NVM) - L(p, DRAM)] \tag{3}$$

The estimated average access time in a hybrid memory system can therefore be rewritten as follows:

$$\hat{A} = \sum_{r \in \text{Regions}} \sum_{p \in \text{Patterns}} F_{D(r)}(p) L(p, NVM) - \sum_{r \in DRAM} B(r) \tag{4}$$

To minimize $\hat{A}$ we need to maximize the second term. We have a fixed number of slots in DRAM into which we can place memory regions and a fixed real quantity $B(r)$ that quantifies the benefit of putting memory region r in DRAM. We use a greedy algorithm to maximize the total benefit by first sorting the regions into a list ordered by decreasing benefit, and then placing regions starting from the head of the list into DRAM until no more space is left in DRAM. This algorithm is optimal because any placement that does not include one of the chosen regions can be further improved by replacing a region not in our chosen set with the missing region.

We note that the benefit $B(r)$ is the same for all memory regions of a particular data structure. Therefore we compute this quantity as a property of the data structure itself and use it for each memory region of the data structure.

4.5.3 Loaded Latency

Our model ignores queuing delays caused by traffic from other cores. In Equation 3, we use the difference in latencies of access to NVM and DRAM. Requests to all types of memory (for all access patterns) follow the same path from the core to the memory controller before being dispatched to the appropriate memory channel. The queuing delays are therefore the same for both types of memory and cancel out.

4.6 Evaluation

We evaluate X-Mem using a set of representative applications: a high-performance graph analytics framework (*GraphMat*), an in-memory database (*VoltDB*), and a key-value store

(*MemC3*). All the three applications work on data sets that can be scaled up and therefore have potential to benefit from the larger capacity and lower cost of NVM in a hybrid memory system. The evaluation has two objectives. The first is to show that the performance of all three applications depends significantly on the placement choice for the data structures in them. Second, we aim to show that our placement technique (§4.5) correctly identifies access patterns to each data structure and enables optimal data placement with X-Mem. To demonstrate the results - in all cases - we provide two baselines for comparison. The first is *NVM-only* – in which an application's memory is allocated entirely from NVM. NVM-only depicts the worst case performance of unmodified applications in a hybrid memory system. The second is *DRAM-only*, which represents the best-case performance of an application when all accessed data is allocated in fast DRAM.

We perform all experiments on a hardware-based hybrid memory emulator (§4.6.1). For the network-based applications (Memcached and VoltDB), we use a dedicated machine to run clients to drive the application under test. The client machine is conencted directly to the server (with no intermediate switch) via multiple 10 GigE NICs, therefore ensuring that neither the network nor the clients are limiting the performance and we can focus entirely on the interaction with main memory.

Although NVM technologies have been announced by some manufacturers [1], the industry is still actively developing the first generation of NVM as well as the controllers to which they will be attached. As a result, till the technology matures, one can expect a spectrum of performance characteristics from NVM, rather than a single point. For proper sensitivity analysis, we therefore use the hybrid memory emulator to study a range of NVM latency and bandwidth characteristics.

4.6.1 Hybrid Memory Emulator

The *hybrid memory emulation platform (HMEP)* was built by Intel and made available to researchers to support the exploration of software stacks for hybrid memory systems. HMEP enables study of hybrid memory with real-world applications by implementing – (i) separate physical memory ranges for DRAM and emulated NVM (using custom BIOS), and

(ii) fine-grained emulation of configurable latency and bandwidth for NVM (using custom CPU microcode).

HMEP is based on Intel Xeon E5-4620 platform, with each of its two processors containing eight 2.6 GHz cores. Hyperthreading is disabled. Each CPU supports four DDR3 channels and memory is interleaved across the CPUs. The measured DRAM latency and bandwidth on the system are 150ns and 37GB/s, respectively. HMEP has been used before in other research [57, 42, 108, 81, 82, 54], and described in detail elsewhere [76].

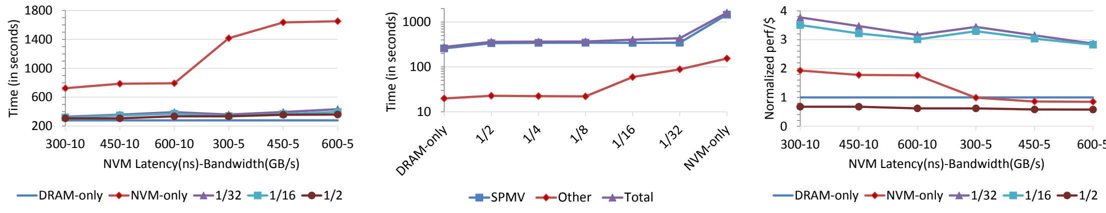
We present results for six HMEP configurations that represent the performance spectrum for NVM (Table 4). In these configurations, NVM's latency ranges from 2× to 4× of DRAM latency (300ns, 450ns, and 600ns), and NVM's bandwidth varies from $\frac{1}{4}\times$ to $\frac{1}{8}\times$ of DRAM bandwidth (10GB/s to 5GB/s). In addition, we also evaluate different ratios of NVM to DRAM in the system. We use *1/T* to refer to a setup where the DRAM size is *1/T* of the total available memory in the system. Corresponding to these setups, we also consider the cost of the total memory. For generality, NVM cost is derived from prior research that expects PCM based SSDs to be 4× more expensive than enterprise MLC NAND flash SSDs [67]. Based on the prevalent prices of DDR-based DRAM [41] and a conservative estimate that directly addressable NVM devices will be more expensive than NVM-based SSDs (at least initially), we assume that DRAM is 5× more expensive than NVM on cost-per-bit basis.

4.6.2 Modifications to use the X-Mem API

Modifying applications to use the X-Mem API involves very few (10 to 50) lines of code in the applications. For each application, we first idenfify data structures that grow with the size of the application data and occupy significant portions of the application's memory footprint. Only such large data structures are allocated using the X-Mem API; other data continue to be allocated in DRAM using the default system allocator. Each data structure is given a unique tag. We are exploring the options to automate the process of attaching a unique tag to each data structure, perhaps as compiler extensions.

Table 6: Memory tiering in GraphMat with the X-Mem API.

Data structure	%Size	%Accesses	Access pattern				Benefit per region
			%pointer	%seq	%random	%writes	
Sparse Vectors	3.5	92.64	0.0	0.34	99.64	32.75	3948.06
Vertex Data	2.4	1.53	0.0	13.98	86.02	56.02	67.05
Adjacency Matrix	94.1	5.83	0.0	0.0	100.0	0.0	9.30



a) Tiering performance b) Time taken (at 600-5) c) Tiering performance/$

Figure 24: Evaluation of memory tiering in GraphMat.

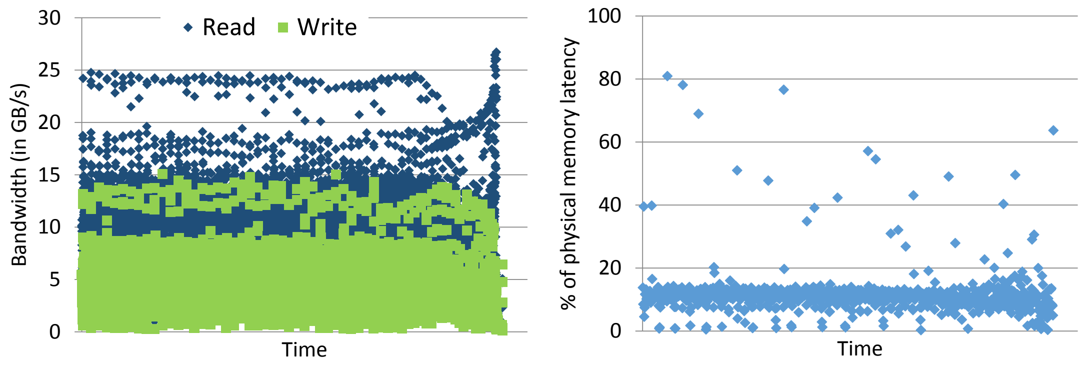
4.7 *GraphMat*

Exploding data sizes increase the importance of large-scale graph-structured computation to various commercial tasks [75]. Large graph processing, however, lacks access locality and therefore many graph analytics frameworks require that the entire data fit in memory [75, 60, 98].

We study a high performance graph analytics framework called GraphMat [98], which implements a vertex-centric scatter-gather programming model. The performance of graph analytics frameworks (like GraphMat) depends both on the topology of the graph and the specific algorithm [93, 98, 76].

Test details: Since the topic of our research is not graph processing, we present results only from execution of the Pagerank algorithm on a large graph representing the Twitter follower network [68]. The graph has 61.5 million vertices and 1.4 billion edges. The Pagerank algorithm on this graph ranks users by the number of influential followers to identify the most influential individuals using Twitter [46].

Figure 25 shows the bandwidth requirements and the *effective latency* of GraphMat. Effective latency, shown as percentage of the actual physical memory latency, approximates the average memory read latency in an application by measuring the core stalls due to pending reads per LLC miss. GraphMat can achieve very high bandwidth usage (of up to 25GB/s read and 15GB/s write bandwidth) but its effective latency is low (mostly



a) Memory bandwidth   b) Effective memory latency

Figure 25: Memory usage in GraphMat (Pagerank algorithm).

under 20%) due to GraphMat's ability to exploit CPU's MLP. GraphMat's performance is therefore highly sensitive to lower bandwidth, but only to some extent to higher latency. This explains the degradation in the NVM-only performance of GraphMat with the decrease in bandwidth rather than increased latency (Figure 24a).

4.7.1 Data Structures and Placement

GraphMat takes vertex programs and maps them to highly optimized, scalable sparse matrix operations in the backend. GraphMat implements graph algorithms as iterative generalized *sparse matrix-sparse vector multiplication (or SPMV)* that updates some data associated with the vertices (e.g., Pagerank).

Graphmat has three main data structures: a sparse vector, a dense vector of vertex data and finally the adjacency matrix of the graph in compressed sparse row format. The sparse vector of vertex data is built on every iteration and consists of the data from active vertices in the computation. By reading from and writing to the sparse vector rather than the full dense vector of vertices, Graphmat achieves better cache efficiency. We modify GraphMat to use xmalloc for these three data structures.

Table 6 illustrates the inner workings of X-Mem's automatic placement algorithm. We provide the sizes of the above mentioned GraphMat data structures (as percentage of the total memory footprint) in the second column, and the relative access frequencies of those data structures in the third column. The most frequently accessed data structure is the sparse vector, followed by vertex data and finally by the adjacency matrix. The next group of columns break down the accesses to each data structure in to pointer chasing, sequential scans and random accesses. For the sake of completion, we also show the percentage of accesses to the data structures that are writes. The access to the sparse vector and adjacency matrix is largely random. The access to the vertex data is somewhat sequential - depending on the active vertices accessed at every pass to build and tear down the sparse vector. The product of the fraction of accesses and the fraction of a particular type gives us the $F_i(t)$ quantity for that particular data structure (Equation 3). In the last column of the table, we show the benefit computed by the placement algorithm for the 450-10 configuration of NVM. The placement algorithm prioritized the sparse vector for placement in DRAM followed by the vertex data and finally the adjacency matrix.

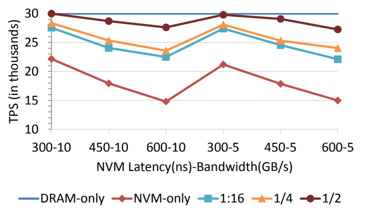
Figure 24a shows the performance of Graphmat with varying memory configurations and X-Mem's adaptive placement enabled. GraphMat's performance for NVM-only is worse than that for DRAM-only by 2.6× to 5.9×, with the drop in performance particularly steep at lower peak NVM bandwidth. Most of the performance gap can be attributed to sparse vectors, which account for only 3.5% of GraphMat's footprint (of 32GB). Figure 24b depicts this overwhelming contribution of SPMV operations, which use sparse vectors, to the overall run time of GraphMat. Performance improves dramatically with 1/32 tiering – only 1.17× to 1.55× worse than DRAM-only – because X-Mem is able to place sparse vectors in available DRAM. With 1/16 tiering, X-Mem can allocate vertex data (less than 2.5% of the total footprint) in DRAM and that improves tiered GraphMat's performance to within 1.13× to 1.4× of DRAM-only. Beyond 1/16, performance is limited by uniform (bandwidth-bound) accesses to the large adjacency matrix. Hence, 1/2's performance is 1.09× to 1.29× of DRAM-only, a minor improvement over 1/16 but at much higher cost.

From the cost viewpoint, 1/32's performance/\$ is 2.9× to 3.8× that of DRAM-only and 2× to 3.4× compared to NVM-only. 1/16 is only slightly behind 1/32 – 1.83× to 2.5× compared to DRAM-only. Interestingly, 1/2's performance/\$ is worse than that of DRAM-only and NVM-only across all HMEP configurations.

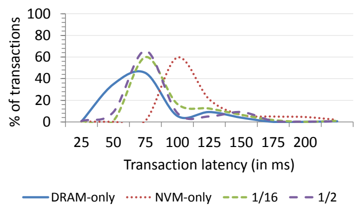
Table 7: Memory tiering in VoltDB with the X-Mem API.

Data structure	%Size	%Accesses	Access pattern				Benefit per region
			%pointer	%seq	%random	%writes	
IDX_ORDER_LINE_TREE	4.73	43.32	48.97	0.12	50.91	27.07	190.17
IDX_S_PK_HASH	1.16	2.7	49.55	0.0	50.45	0.0	56.14
CUSTOMER	0.66	3.28	0.06	0.0	99.94	7.19	27.92
IDX_O_U_HASH	0.52	0.17	3.01	0.0	96.99	69.33	5.99
STOCK	20.36	16.57	0.0	0.0	100.0	5.55	5.31
ORDER_LINE	11.61	5.98	0.01	0.33	99.66	28.59	5.13
IDX_OL_PK_HASH	5.73	1.69	8.68	0.95	90.37	55.89	4.81
HISTORY	0.94	0.25	2.03	0.0	97.97	56.51	4.27
IDX_CUSTOMER_NAME_TREE	16.43	1.65	32.76	0.0	67.24	0.0	1.93
CUSTOMER_NAME	35.90	0.59	0.0	0.0	100.0	0.0	0.11

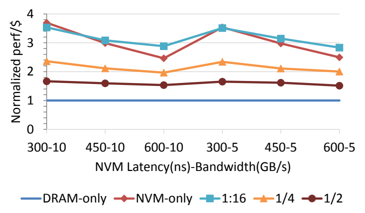
4.8 VoltDB



a) Tiering performance



b) Transaction latency (at 600-5)



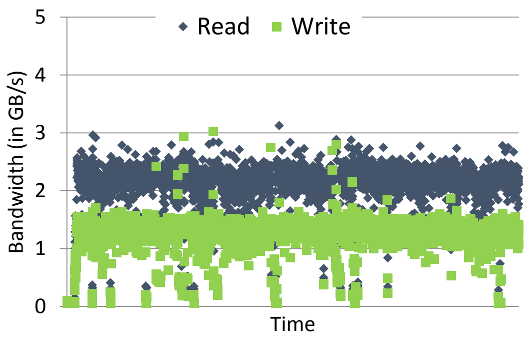
c) Tiering performance/$

Figure 26: Evaluation of memory tiering in VoltDB.

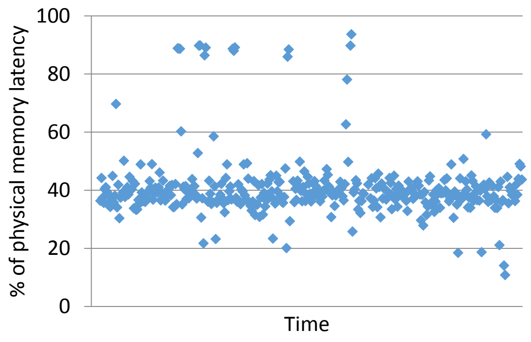
In-memory databases exploit architectural and application trends to avoid many overheads commonly associated with disk-based OLTP databases [34, 33, 97, 37]. VoltDB, one such in-memory database, is an ACID-compliant, distributed main memory relational database that implements the shared-nothing architecture of H-Store [97].

Test details: For the results in this case, we run the industry standard TPC-C benchmark, which models the operations of a wholesale product supplier [36] . We set the number of warehouses to 512, the number of sites in VoltDB to 8, and report TPC-C throughput in total transactions per second.

Figure 27a shows that VoltDB's performance is sensitive only to memory latency, which can be explained by the high effective latency – of over 50% at times (Figure 27c) – caused by random accesses in VoltDB (particularly to the indices). VoltDB consumes an average read bandwidth of 2.3GB/s and write bandwidth of only 1.3GB/s for the TPC-C workload (Figure 27b), and is therefore insensitive to lower bandwidth.



a) Memory bandwidth



b) Effective memory latency

Figure 27: Memory usage in VoltDB (TPC-C workload).

4.8.1 Data Structures and Placement

VoltDB scales by horizontally partitioning the tables across multiple single-threaded engines (called sites) and replicating the partitions. We modify VoltDB to allocate indices and tables using xmalloc. Overall, TPC-C in VoltDB contains 28 such data structures, of which the top 10 data structures (size-wise) account for approximately 99% of the total memory footprint. Table 7 illustrates the application of X-Mem's placement algorithm to these top 10 data structures. Intermediate results and other data structures are allocated in DRAM using default system allocator.

Figure 26a shows that NVM-only's performance is 26% to 51% worse than that of DRAM-only, which is relatively low compared to GraphMat because TPC-C is comparatively less memory-intensive. 1/16 performs better than NVM-only by 24% to 48%, just by enabling X-Mem to place temporary data and three frequently accessed data structures – tree-based secondary index for the ORDER_LINE table, hash-based primary index for the STOCK table, and the small CUSTOMER table – in DRAM. 1/8 (not shown here) and 1/4 do not perform much better than 1/16, mainly because the accesses to the remaining data structures are uniformly random, resulting in no obvious contender for placement in DRAM in X-Mem. Only at 1/2 is X-Mem able to place most of the data structures in VoltDB, barring the very large, but infrequently used, CUSOMTER_NAME table and some portions of the tree-based primary index for the same table. As a result, 1/2's performance is 35% to 82% better than NVM-only and only up to 9% worse than DRAM-only.

Figure 26b shows the latency distribution of TPC-C transactions, since it is an important metric for OLTP workloads. While 80% of the transactions complete in 75ms with DRAM-only, the average transaction latency increases only moderately with 1/2, with 66% of the transactions completing within 75ms and 75% of them in 100ms. Average transaction latency with 1/16 is slightly worse but still 75% of the transactions complete within 100ms.

1/16's performance/$ is better than DRAM-only by 2.5× to 3.7×, and best across most HMEP configurations. The only exception is 300-10, where NVM-only provides better performance/$ than 1/16. More importantly, all of the tiering options offer better performance/$ than the DRAM-only option. 1/2's performance/$ is, for instance, 1.5× to 1.7× better than DRAM-only, while providing within 9% of DRAM-only's performance.

Note that our cost analysis does not take into account the wearing of the NVM devices and the fact that NVM devices have limited write endurance (Table 4). We explore this aspect briefly in a later section (§4.10).

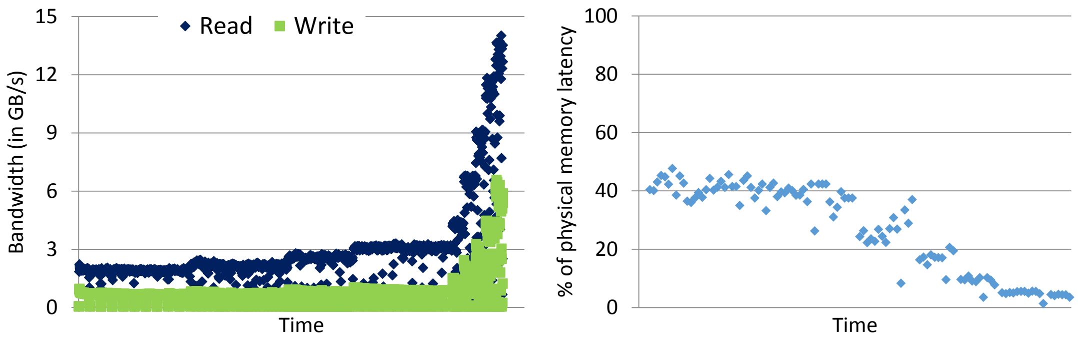
4.9 MemC3

Key-value stores are becoming increasingly important as caches in large data centers [43]. We study a variation of Memcached [30] called MemC3 – a major rewrite of open source Memcached that improves both memory efficiency and throughput over the stock implementation [58]. MemC3 implements a variant of cuckoo hashing to avoid expensive random dependent accesses that are common in Memcached. In addition, MemC3 replaces Memcached's exclusive, global locking with an optimistic locking scheme.

Test details: Key-value stores (such as Memcached and MemC3) are used in widely different scenarios. Unlike with OLTP databases, there is a lack of standard benchmarks for key-value stores. We base our test workload on data recently published by Facebook [43, 80], specifically on the ETC trace.

To test MemC3 in a deterministic manner, we modified *memaslap* [29] to implement separate load and execution phases (similar to YCSB [38] but without requiring the traces to be saved to a file). During the load phase, the MemC3 server is populated with a dataset of approximately 64GB. The key size is fixed at 16 bytes and the size of the values ranges from 16B to 8K, roughly following the distribution of values in the ETC trace in Facebook's published data [43]. More than 90% of the values are 512 bytes or less in size. However, values greater than 512 bytes occupy almost 50% of the cache space.

During the execution phase, the client generates 50 million random queries based again on the value distribution in the ETC request trace. The workload is read heavy (95% GET and 5% SET) and currently does not contain any DELETE requests. We report the total time to complete the test.



a) Memory bandwidth b) Effective memory latency

Figure 28: Memory usage in MemC3 (Facebook-like workload).

Figures 28 shows the performance characterization of *MemC3* with the client requesting values of one size at a time (from 16B to 8K). Figure 28a shows that MemC3's read bandwidth ranges from 1.8 GB/s to 14 GB/s and write bandwidth ranges from 1 GB/s to 6.5 GB/s, depending on the size of the value. Effective latency decreases (from 48% to 4%) at larger values, due to improved sequential performance (Figure 28b).

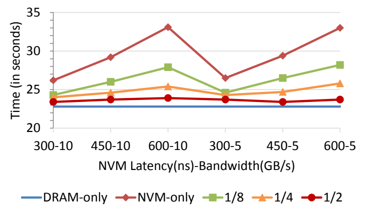
MemC3's sensitivity to higher latency rather than lower bandwidth, as depicted by NVM-only's performance in Figure 29a, is explained by the large skew towards the smaller values in our workload. Figure 29b further illustrates this fact by breaking down NVM-only's overheads at various value sizes.

4.9.1 Data Structures and Placement

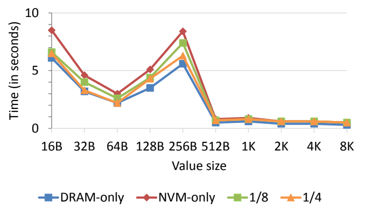
MemC3 allocates memory for the cuckoo hash table at start-up and employs a dynamic slab allocator to allocate values, with the slab type determined by the size of allocation from the slab. The hash table and the slabs are all allocated using xmalloc, resulting in

Table 8: Memory tiering in MemC3 with the X-Mem API.

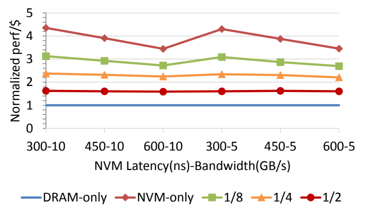
Data structure	%Size	%Requests	%Accesses	Access pattern				Benefit per region
				%pointer	%seq	%random	%writes	
Cuckoo hash	12.35	NA	17.65	21.15	0.0	78.85	3.44	83.76
16B, 32B, 64B	8.64	55.0	23.51	0.0	0.0	100.0	18.63	70.53
256B	18.52	16.0	16.29	0.0	0.0	100.0	13.37	22.77
128B	16.05	24.0	8.48	0.0	0.0	100.0	15.41	13.7
4096B	6.17	2.0	10.47	0.0	91.95	8.05	4.98	13.3
512B	4.32	1.0	2.01	0.0	21.02	78.98	10.98	10.35
1024B	4.94	1.0	3.41	0.0	72.21	26.79	6.78	9.01
2048B	10.49	0.7	7.67	0.0	86.5	13.5	5.49	7.7
8192B	18.52	0.3	10.5	0.0	94.32	5.68	4.49	5.11



a) Tiering performance



b) Time taken (at 600-5)



c) Tiering performance/$

Figure 29: Evaluation of memory tiering in MemC3.

a total of nine data structures in X-Mem (Table 8) since each slab type is treated as a separate data structure. Table 8 shows the priorities of these data structures as determined by the X-Mem placement algorithm. Note that the priorities determined by X-Mem for the various slab types fully follow the value-wise breakdown of NVM-only's overhead in Figure 29b, demonstrating the accuracy of X-Mem placement model.

Figure 29a shows that, depending on the HMEP configuration, NVM-only's performance is 1.15× to 1.45× worse than that of DRAM-only. With 1/8 tiering, where only 12.5% of the application data is in DRAM, performance improves by 8% to 17% over NVM-only because X-Mem allocates MemC3's cuckoo hash DRAM. 1/4 fares even better and improves the performance to within 6% to 13% of DRAM-only, mainly because X-Mem is now able to place a large number of small values in DRAM and reduce the number of random accesses to NVM.

Increasing the amount of available DRAM beyond 1/4 results in incremental improvements, and eventually at 1/2, performance overhead over DRAM-only is within 5%. However, this improvement comes at much higher cost as depicted in Figure 29c. NVM-only provides best performance/$ across all HMEP configurations – 3.5× to 4.5× compared to

DRAM-only, mainly because the NVM-only overhead is relatively low. As mentioned before, this cost analysis only accounts only for the initial cost of memory. The effect of tiering on the wearing of NVM devices is discussed later (§4.10). Finally, all tiering options provide significantly better performance/$ (1.6× to 3.1×) than DRAM-only, resulting in a range of interesting performance/cost tradeoffs.

4.10 Practical Deployment

X-Mem is intended for deployment with soon to be introduced NVM based systems [63]. Therefore, design choices in X-Mem were made with following practical considerations.

4.10.1 Human Overhead

X-Mem offers a powerful yet simple API for applications to express important semantic information about their data. While replacing malloc and free with xmalloc and xfree is straight forward, assigning unique tags to individual data structures might require more effort depending on the implementation. In our case, modifying VoltDB required the most effort, but still involved changes to under 50 lines of code. The effort is considerably lower for object oriented programs where a class-specific allocator can be defined for calling `xmalloc`, rather than scattering those calls across the code.

No further effort or reasoning is necessary on the part of the programmer since the automatic placement algorithm does a job *at least as good* as a human programmer in placing data structures. To verify this claim, we repeated the experiments with other possible orderings of data structures for placement in DRAM. In case of VoltDB and MemC3, because of the large number of data structures, we selected a subset of the possible permutations that we deemed most optimal based our understanding of the applications. For Graphmat we evaluated all possible permutations. In no case were we able to outperform our automatic placement algorithm. For example, with GraphMat, we were able to run all possible permutations for placement, with one instance (1/16 tiering at 600-5), showing that the placement model's suggestion was 1.0× to 5.8× better than the alternatives. Detailed results on these experiments are omitted due to space constraints.

4.10.2 Profiling Overhead

Memory tracing (with Pin) during the profiling step can significantly slow down the application (up to 40× in our tests), and is therefore suited only for offline use and not for production. We are currently exploring extensions to the x86 ISA that would enable necessary profiling but without the expensive process of tracing each memory access. Such hardware extensions would enable real-time measurement and placement in X-Mem. The profiler however remains the solution of choice for initial deployment.

We also found that it is useful to downsize the workload while preserving the relation between its various data structures, in order to reduce the time spent in profiling. For instance, in the case of MemC3, we resize the number of values and the number of requests in the workload proportionately to the original distribution.

4.10.3 Device Wear

Device wear is a significant concern for the initial generation of NVM devices. With X-Mem based tiering, the applications and OS continue to use DRAM from the default system allocator for small temporary data and program stack, while X-Mem prioritizes the use of the remaining DRAM for frequently used important application data. In that regard, X-Mem acts as a natural guard against wear by moving the most frequently accessed (and written) data structures to DRAM. Figure 30 shows that, depending on the mix of DRAM and NVM, X-Mem reduces writes to NVM by 48% to 98% (over NVM-only) in the applications evaluated in this work.

This work prioritizes performance over reducing wear, but it is possible to configure X-Mem to instead prioritize write-intensive data structures for placement in DRAM. We do not explore that avenue in this dissertation.

4.10.4 Migration Overhead

X-Mem performs "synchronous" migration by waiting for the `mbind` call to return when a memory region is migrated from DRAM to NVM. The overhead of this migration is negligible in our experiments due to the relatively small amount of time spent in allocations

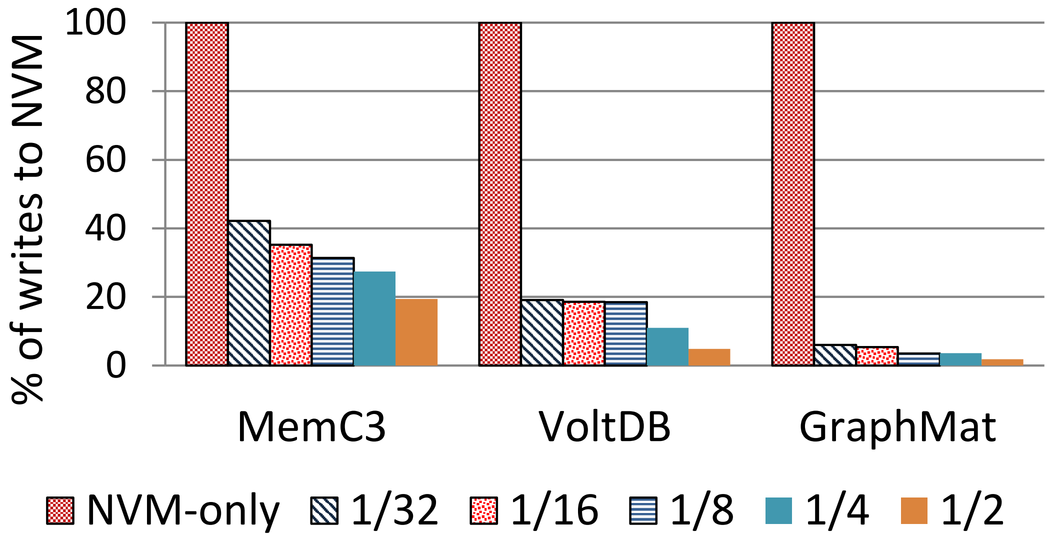


Figure 30: Writes to NVM as a percentage of total writes

Table 9: Time taken to migrate one gigabyte of application data from DRAM to NVM (at 600-5). *S* and *L* denote the use of 4K pages and 1G pages respectively.

Region Size	64M-S	512M-S	1G-S	1G-L
Time (in ms)	1015	972	924	378

compared to the overall run time of the application.

Table 9 shows the raw overhead of migration for region sizes ranging from 64M to 1G. 1G regions can amortize the cost of migration by 5% to 8% over smaller regions. The real benefit of using 1G regions however is that it allows the use of large (one gigabyte) hardware pages on most 64-bit servers; thereby further reducing the migration overhead by 60% compared to the baseline with 4K pages (1G-S). While we understand the benefits of using 1G regions with large pages in applications with large memory footprint [57], the current implementation of X-Mem uses 64 MB regions (by default) to reduce internal fragmentation. Also, X-Mem uses 4K hardware pages due to limitations in the Linux kernel w.r.t. enforcement of memory policies on large pages. We plan to address this situation in the future.

4.10.5 Choosing the right implementation

The ease of tiering in an application and the resulting benefits depend heavily on the application's implementation. Figure 19 shows DRAM-only and NVM-only results for popular implementations of key-value stores, in-memory databases, and graph analytics frameworks. MemC3 performs significantly better than Memcached [30] in all tests, mainly because MemC3 eliminates dependent accesses to memory and avoids exclusive reader-writer locks which can be particularly expensive if it involves random accesses to NVM while the locks are held (as in Memcached).

Figure 19c shows the time taken by these frameworks to run the Pagerank algorithm on a RMAT scale 24 graph [15]. GraphMat performs much better than other specialized graph analytics frameworks, including GraphLab [60], Galois [79], and X-Stream [92]. GraphMat's better performance (across all tests) is due to its highly optimized core operations (SPMV), which are memory efficient and cache conscious. In comparison, GraphLab's performance degrades rapidly at higher latency due to random accesses and lock based concurrency. Galois, which is based on a fine-grained task abstraction and scheduling, is again sensitive to latency. And X-Stream's model of streaming in graph data for edge centric processing is very bandwidth intensive.

4.11 Summary

Future systems are likely to address scalability and cost limitations of DRAM with hybrid memory systems containing DRAM and Non-Volatile Memories (NVM) such as PCM and RRAM. This work proposes data classification and tiering techniques for best matching application data with underlying memory types in hybrid systems. We consider three large in-memory server workloads – MemC3, VoltDB, and GraphMat. In a hybrid system with only a small amount of DRAM (6% of the memory footprint for GraphMat, 25% for MemC3 and VoltDB), applying data tiering to these workloads improves their performance by as much as 22% to 76% over the corresponding unmodified versions. At the same time, these tiered applications perform merely 13% to 40% worse than their counterparts running entirely in DRAM, thus achieving 2× to 2.8× better performance/$. Service providers can use the data tiering techniques in this chapter to enable a range of differentiated services in hybrid memory systems. In addition, NVM enables applications to host much larger datasets on a single system than possible with DRAM alone. We plan to explore the impact of this fact on the design choices of large scale in-memory distributed systems [107, 83, 93, 80].

CHAPTER V

RELATED WORK

PM in the hybrid memory systems has been explored in several contexts. Prior work has examined the use of PM for both capacity and persistence, with emphasis on the necessary system software and libraries to provide applications with efficient access to PM [66, 52]. Others have proposed programming models for applications that seek consistent, low-overhead access to PM as fast byte-addressable storage [101, 51, 99]. Some of this work, particularly the persistent programming models, is complementary to ours. The key contribution of this dissertation, however, is a holistic view of the software stack necessary to harness the full potential of PM, both as fast storage and as scalable but slower memory.

5.1 PM as fast storage

File systems have always been optimized for the storage media [91, 102, 65]. As an example, recently several file systems have either been completely designed for flash or optimized for flash [13]. DirectFS, a file system optimized for (vendor-specific) flash-based storage, is one such example [65]. PMFS is optimized for PM and the processor architecture.

Researchers have long explored attaching storage to the memory interface. eNVy is one such system that proposed attaching flash to the memory bus, using controller logic and SRAM to achieve high throughput at fairly low latency [103]. However, it is only recently that NVDIMM solutions are entering the mainstream, with a number of vendors providing either small capacity, byte-addressable NVDIMMs [19] or large capacity, block-addressable NVDIMMs [94]. Large capacity, byte-addressable PM is a natural evolution made possible by emerging NVM technologies.

Systems such as Rio File Cache [48] and, more recently, ConquestFS [102] suggested using a hybrid NVM-Disk or NVM-Flash for better performance, with the file system responsible for managing NVM efficiently. PMFS is designed for large capacity PM. We intend to explore tiering with other, possibly cheaper, storage technologies in the future.

Some researchers have proposed using a single-level store for managing PM, obviating the need to translate between memory and storage formats [78]. However, this requires significant changes to applications and does not benefit legacy storage applications. PMFS allows for a smooth transition from file-based access to memory-like semantics with PM by implementing a light-weight POSIX file interface and optimizing memory-mapped I/O.

PMFS is more directly comparable to file systems optimized for PM-only storage, such as BPFS [52]. BPFS uses copy-on-write (CoW) and 8-byte in-place atomic updates to provide metadata and data consistency. PMFS, in contrast, uses larger in-place atomic updates with fine-grained logging for metadata consistency, and CoW for data consistency. PMFS optimizes memory-mapped I/O, through direct mapping of PM to the application's address space and use of transparent large page mappings with *mmap*. BPFS doesn't support *mmap*. In PMFS, we also provide low overhead protection against stray kernel writes. BPFS doesn't address this issue. Finally, BPFS is designed assuming a very elegant hardware extension (*epochs*) for software enforceable guarantees of store durability and ordering. However, support for *epochs* requires complex hardware modifications. We assume only a simple *pm_wbarrier* primitive to flush PM stores to a power fail-safe destination. The decoupled ordering and durability primitives proposed in this chapter are similar to the file system primitives, *osync()* and *dsync()*, proposed for OptFS [49].

SCMFS, like PMFS and BPFS, is a file system that is optimized for PM [104]. SCMFS leverages the OS VMM and the fact that the virtual address space is much larger than physical memory to layout the files as large contiguous virtual address ranges. PMFS, on the other hand, manages PM completely independent of the OS VMM, by implementing a page allocator that is optimized for the goals of PMFS. While the authors note that SCMFS uses only *clflush/mfence* for ordering, the details of consistency (in the presence of failures) are unclear. On the other hand, in PMFS, the design and implementation of consistency mechanisms are among the key contributions. Furthermore, consistency in PMFS has undergone careful and thorough validation (§3.4).

Programming for PM is tricky, particularly when PM accesses are cached write-back. We faced many challenges with PMFS ourselves, not least of which was validating and testing

PMFS for correctness. Applications that want to operate directly on PM (for performance) encounter similar issues too. While *mmap* in PMFS does provide (efficient) direct access to PM, it is too low-level for many application programmers. *Failure-atomic msync* [84] addresses this by enabling atomic commit of changes to *mmap*'ed files. Implementing failure-atomic *msync* in PMFS is not difficult, but it is unclear if it is the right approach for PM. We will explore that in the future. Other researchers have proposed interesting programming models [101, 51] and library solutions [99] to simplify PM programming. These solutions, referred to as PMLib in Figure 6, complement our work in this chapter, and can be built on the system-level PMFS services, such as naming, access control, and direct access to PM with *mmap*.

5.2 PM as scalable memory

There have been many efforts, both in the past and ongoing, that explored the use of PM as scalable (but slower) memory, with special emphasis on the necessary architectural support. To underline the fact that persistence is irrelevant to this usage, we refer to PM as NVM throughout the section.

Qureshi et al. [90] discuss the use of NVM as main memory and evaluate several main memory organizations with DRAM and PCM, including NVM-only and multi-level memory (or MLM), where DRAM is employed as transparent hardware-managed cached to the large capacity NVM. Their evaluation is based on simulation of a simple in-order processor model and memory that models only higher latency of PCM and not lower bandwidth. Further, their evaluation is limited to simple medium-sized application kernels. Our goal is to evaluate the performance of NVM on modern CPUs with out-of-order execution and prefetch capabilities, and with large-scale applications that are both latency-sensitive and bandwidth-intensive. The hybrid memory emulator enables such realistic evaluation.

Previous research has demonstrated that MLM can achieve close to DRAM performance with NVM, depending on the implementation of DRAM cache and the workload [90]. Others present techniques to implement efficient stacked DRAM caches [72, 88], but concedes that large hardware-managed caches are stop gap measures before software stacks (both OS and applications) develop to exploit heterogeneous memories [72]. Our study suggests that such the hybrid memory architecture provides applications with required flexibility, and is particularly interesting for large in-memory data center applications. We focus only on software managed hybrid memory systems and techniques to achieve optimal performance with them.

Lim et al. [71] study the use of slow memory in the context of shared, network-based (disaggregated) memory and conclude that a fast paging-based approach performs better than direct access to slow memory. While their choice of target applications is key to their findings, their work also relies on a simple processor model that does not account for CPU's MLP and prefetch features (unlike our work). In our experiments, we found that paging to NVM is several times slower than even the corresponding NVM-only solutions. Others have made similar observations regarding performance of paging in Linux [39].

Ferdman et al. [59] conduct a thorough study of many scale-out workloads using hardware performance counters and conclude that these workloads are unable to exploit the CPU's memory-level parallelism (MLP), leading to poor power efficiency. Our work is different from theirs in several ways – (i) Since our goal is to study the use of NVM, our workloads are all large in-memory applications, and (ii) depending on the implementation, our workloads are able to achieve high MLP and therefore amortize cost of accessing slower NVM, and (iii) we conclude that, for future hybrid memory architectures with NVM, it is imperative (and hugely beneficial) for the application's performance to exploit MLP and hardware prefetching when accessing NVM, even if it requires re-designing these applications.

Qureshi et al. [89] study the impact of MLP on the effective cost of LLC misses in an application, and categorize those misses as costly isolated/dependent misses and cheaper parallel/independent misses. Their proposal to expose this information to cache replacement algorithms to reduce the number of isolated misses is even more relevant to the NVM architectures in this dissertation, owing to NVM's higher latencies. We employ similar techniques in the classification of data structures in X-Mem, by attributing different costs to the various memory access patterns in the placement model.

AutoNUMA monitors memory usage of applications in NUMA platforms and co-locates their memory with compute (via scheduling and migration) for optimal performance [16]. Data co-location in NUMA platforms, however, is very different from data placement in hybrid memory systems, where NVM is slow to access from all processors. AutoNUMA does not solve the problem of matching application data with physical memory.

NVML is an open source implementation of allocators and abstractions optimized for NVM, both for persistent and volatile usage [32]. Like X-Mem, one of the goals of NVML is to exploit NVM's capacity and cost attributes. But, X-Mem's objectives go beyond that of NVML. For one, X-Mem alleviates applications from the responsibility of understanding system's topology, properties of memories, data migration at runtime, etc. Secondly, application-driven memory tiering with X-Mem has much broader scope and the X-Mem runtime can be extended to support any future hybrid system irrespective of the memory technology [26, 35].

SSDAlloc is a hybrid memory manager with a custom API that enables applications to use flash-based SSDs as an extension of DRAM [44]. In comparison, X-Mem is designed for hybrid memory systems where all memories are directly accessible to software and there are no overheads related to block-based accesses. X-Mem is concerned with optimal placement and tiering of classified application data whereas SSDAlloc is an optimized paging solution that implements techniques such as an *object-per-page* model and on-demand page materialization to extract maximum performance out of SSDs. For the same reason, X-Mem API is richer than SSDAlloc API.

Data classification is a well-studied problem in the context of traditional block-based storage [95, 77, 25]. Mesnier et al. [77] propose the classification of I/O data at block granularity to enable differentiated storage services (DSS) for traditional file and storage systems. A DSS classifier is internally translated to a relative priority based on a predefined policy that was assigned to the file by software. Data classification in X-Mem is finer grained and preserves semantic information at the granularity of a single allocation. As a result, X-Mem enables optimal data placement between memories whose performance is not necessarily orders of magnitude apart.

An alternative to dynamic tracing of memory accesses is static analysis to determine the access pattern to various data structures. However, since we target unmanaged languages, static analysis would require some form of “points-to-analysis” [96] to map memory accesses through pointers to the actual types that they refer to. Since points-to-analysis is necessarily conservative it leads to situations where an access maps to more than one type or data structure, therefore resulting in inaccurate attribution of accesses. In addition static analysis does not give dynamic access counts; for example, the number of accesses within loops whose iteration count is input dependent. Dynamic access counts are critical to the model we use for placement decisions.

Linux allows applications to provide hints about their use of virtual memory using *madvise* system call [28]. OS is free to act on or ignore those hints. Jantz et al. [64] propose a *page coloring* framework for applications to dynamically attach an *intent* to a group of pages. The applications and OS collaborate in a feedback loop to execute on (or fine tune) the intents. Others have demonstrated the benefits of profile guided page placement in high performance GPU kernels with a small number of (usually large) statically allocated data structures [40]. Since the target applications in this work are mostly bandwidth bound, the focus is on hybrid memory systems with high-bandwidth memory [73] and regular DDR-DRAM, and exposing the bandwidth asymmetry between these memories. Our work differs from these efforts in many ways. For one, all of these existing mechanisms operate at the granularity of (at least) a page in virtual memory, but applications allocate memory for one object (often smaller than a page) at a time. X-Mem preserves this semantic information by operating at the allocator level and by using tags for application data structures. Secondly, the profiling techniques in the previous efforts assume homogeneous (bandwidth optimized) accesses to all application data, and therefore use the frequency of accesses to data as the proxy for their relative importance. X-Mem makes no such assumptions, and is aware of the fact that both the frequency and the actual access pattern matter. Therefore, X-Mem is more general purpose. Finally, unlike the previous work, X-Mem takes a holistic approach towards data placement and runtime management (e.g., dynamic migration), all the while exploiting the additional information provided by the applications.

Prior work has examined the use of NVM in the hybrid architecture as CPU addressable persistent memory. For instance, PMFS is a light weight file system optimized for NVM. In addition to supporting regular file system interface, PMFS provides applications with direct access to NVM with the *mmap* interface [57]. Some researchers have proposed persistent programming models for applications that require consistent, low-overhead access to NVM [101, 51, 99]. Others are exploring systems (particularly databases) that are partially or completely redesigned for NVM [81, 108, 42]. X-Mem is complementary to these efforts.

CHAPTER VI

SUMMARY AND FUTURE DIRECTIONS

Byte-addressable non-volatile or persistent memory (NVM or PM) technologies have been on the horizon for a long time [90]. As industry finally gears towards mass production and adoption of PM [63], its potential to bridge the gap between DRAM and NAND flash is one of the most exciting technological developments in the recent years. The promise of PM comes at the right time for the computing industry, as the big data trends and server consolidation in cloud environments continue to push the demand for faster storage and higher capacity memory at lower cost. PM however has profound implications on system design, necessitating rethinking of the hardware architecture and (more importantly) the software stack.

This dissertation addresses the problem of efficiently and consistently managing PM. The key contribution of our research is a set of technologies for using PM as both fast storage and scalable (although slower) memory. Towards this end, we have developed a light-weight POSIX compliant file system (called *PMFS*) to enable low overhead access to PM by both legacy applications (using file based access) and new applications that access PM directly (using the memory mapped interface). For consistency, we propose a software flow (plus minimal hardware extensions) that can guarantee ordering and durability of stores to PM. PMFS employs several architectural techniques to optimize this software flow, and provides a reference implementation that is generic enough to be useful to other PM software (such as databases). Finally, we explore the use of PM as scalable memory that can be coupled with DRAM for capacity and cost reasons. This use case is particularly compelling for large in-memory applications commonly deployed in data center environments. The challenge of course is to use PM in a manner that is conscious of the design of the application data structures and their access patterns. For this purpose, we developed *X-Mem*, a profile guided runtime system that enables optimal placement of application data between different tiers of memory. X-Mem requires only minor changes to the application source code, and provides automatic tools to further minimize the programmer effort required for data classification.

In summary, this dissertation provides hard evidence for the thesis statement, specifically that *new system software and application-level mechanisms are required to harness the full potential of future hybrid memory systems (coupling DRAM with PM) for the increasingly performance hungry data center applications.*

As an ongoing effort, we are exploring several extensions to our work in this dissertation. On the architectural front, we are looking into hardware primitives that can further reduce the overhead of consistency, and perhaps completely avoid logging in the data path. Another important architectural extension would be a low overhead mechanism to monitor the access pattern (or at least the number of accesses) to specified virtual address ranges. In conjunction with X-Mem, such hardware support could be used to implement online profiling with acceptable performance; an important feature if we want to improve dynamic migration (particularly for regions with the same tag) in X-Mem.

From the software viewpoint, we are investigating extensions to PMFS and X-Mem. For one, we are exploring ways to integrate the two systems in a seamless manner, while supporting multiple tiers of memory and storage (including block oriented NAND based SSDs). One option is to extend PMFS to manage both the storage and memory tiers, and explore richer interfaces (e.g., additional hints to madvise) to the applications. The main idea is to enable applications/runtime to convey the characteristics of both volatile and persistent application data to the control layer in OS. Such a hybrid memory/storage management layer can support additional research into the control and allocation of resources in multi-tenant cloud environments.

Finally, we would like to the explore the role of PM in high performance distributed systems, particularly in conjunction with the ultra low latency Ethernet and RDMA networks. The fact that PM enables applications to host much larger datasets on a single system than previously possible, and the fact that such systems will be connected together by next generation networks with sub-microsecond latency are bound to have major repercussions on future distributed systems.

REFERENCES

[1] https://en.wikipedia.org/wiki/3D_XPoint.

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

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

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

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

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

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

[8] "Intel64 Software Developer's Manual (Vol 3A, Ch 4.8)." http://www.intel.com/Assets/en_US/PDF/manual/253668.pdf.

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

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

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

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

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

[14] "Wikimedia Downloads." http://dumps.wikimedia.org.

[15] "Introducing the Graph 500 - Cray User Group." https://cug.org/5-publications/proceedings_attendee_lists/CUG10CD/pages/1-program/final_program/CUG10_Proceedings/pages/authors/11-15Wednesday/14C-Murphy-paper.pdf, 2010.

[16] "AutoNuma." https://www.kernel.org/pub/linux/kernel/people/andrea/autonuma/autonuma_bench-20120530.pdf, 2012.

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

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

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

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

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

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

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

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

[25] "fadvise - Linux man page." http://linux.die.net/man/2/fadvise, 2014.

[26] "Intel Xeon Phi (Knights Landing) Architectural Overview." http://www8.hp.com/hpnext/posts/discover-day-two-future-now-machine-hp#.U9MZNPldWSo, 2014.

[27] "Intel64 and IA-32 Architectures Optimization Reference Manual." http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, 2014.

[28] "madvise - Linux man page." http://linux.die.net/man/2/madvise, 2014.

[29] "Memaslap." http://docs.libmemcached.org/bin/memaslap.html, 2014.

[30] "Memcached - a distributed memory object caching system." http://memcached.org, 2014.

[31] "Numa - Linux man page." http://linux.die.net/man/3/numa, 2014.

[32] "NVM Library." http://pmem.io/nvml, 2014.

[33] "Oracle Database In-Memory." http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html, 2014.

[34] "SAP HANA for Next-Generation Business Applications and Real-Time Analytics." http://www.saphana.com/servlet/JiveServlet/previewBody/1507-102-3-2096/SAP\%20HANA\%20Whitepaper.pdf, 2014.

[35] "The Machine from HP." http://www8.hp.com/hpnext/posts/discover-day-two-future-now-machine-hp#.U9MZNPldWSo, 2014.

[36] "TPC-C." http://www.tpc.org/tpcc, 2014.

[37] "VoltDB." http://voltdb.com/downloads/datasheets_collateral/technical_overview.pdf, 2014.

[38] "Yahoo Cloud Serving Benchmark (YCSB)." http://labs.yahoo.com/news/yahoo-cloud-serving-benchmark, 2014.

[39] "Improving page reclaim." https://lwn.net/Articles/636972, 2015.

[40] Agarwal, N., Nellans, D., Stephenson, M., O'Connor, M., and Keckler, S. W., "Page Placement Strategies for GPUs Within Heterogeneous Memory Systems," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, 2015.

[41] Amazon, "Supermicro Certified MEM-DR432L-SL01-LR21 Samsung 32GB DDR4-2133 4Rx4 LP ECC LRDIMM Memory," 2015.

[42] Arulraj, J., Pavlo, A., and Dulloor, S. R., "Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, 2015.

[43] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M., "Workload Analysis of a Large-scale Key-value Store," in *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE Joint International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '12, 2012.

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

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

[46] Brin, S. and Page, L., "The Anatomy of a Large-scale Hypertextual Web Search Engine," in *Proceedings of the Seventh International Conference on World Wide Web 7*, WWW7, 1998.

[47] Chen, F., Mesnier, M., and Hahn, S., "A Protected Block Device for Non-volatile Memory. LSU/CSC Technical Report (TR-14-01)," 2014.

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

[49] Chidambaram, V., Pillai, T. S., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Optimistic Crash Consistency," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[50] Chou, Y., Fahs, B., and Abraham, S., "Microarchitecture Optimizations for Exploiting Memory-Level Parallelism," in *Proceedings of the 31st Annual International Symposium on Computer Architecture*, ISCA '04, 2004.

[51] Coburn, J., Caulfield, A. M., Akel, A., Grupp, L. M., Gupta, R. K., Jhala, R., and Swanson, S., "NV-Heaps: Making Persistent Objects Fast and Safe with Next-generation, Non-volatile Memories," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[52] Condit, J., Nightingale, E. B., Frost, C., Ipek, E., Lee, B., Burger, D., and Coetzee, D., "Better I/O Through Byte-addressable, Persistent Memory," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, 2009.

[53] Dean, J. and Ghemawat, S., "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, 2004.

[54] DeBrabant, J., Arulraj, J., Pavlo, A., Stonebraker, M., Zdonik, S., and Dulloor, S., "A prolegomenon on OLTP database systems for non-volatile memory," in *ADMS@VLDB*, 2014.

[55] Dragojević, A., Narayanan, D., Castro, M., and Hodson, O., "FaRM: Fast Remote Memory," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, 2014.

[56] Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Pratt, I., Warfield, A., Barham, P., and Neugebauer, R., "Xen and the art of virtualization," in *Proceedings of the ACM Symposium on Operating Systems Principles*, 2003.

[57] Dulloor, S. R., Kumar, S., Keshavamurthy, A., Lantz, P., Reddy, D., Sankaran, R., and Jackson, J., "System Software for Persistent Memory," in *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, 2014.

[58] Fan, B., Andersen, D. G., and Kaminsky, M., "MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, 2013.

[59] Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafaee, M., Jevdjic, D., Kaynak, C., Popescu, A. D., Ailamaki, A., and Falsafi, B., "Clearing the Clouds: A Study of Emerging Scale-out Workloads on Modern Hardware," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[60] Gonzalez, J. E., Low, Y., Gu, H., Bickson, D., and Guestrin, C., "PowerGraph: Distributed Graph-parallel Computation on Natural Graphs," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, OSDI'12, 2012.

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

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

[63] Intel, "3d x-point press announcement." `http://newsroom.intel.com/community/intel_newsroom/blog/2015/07/28/intel-and-micron-produce-breakthrough-memory-technology`, 2015.

[64] Jantz, M. R., Strickland, C., Kumar, K., Dimitrov, M., and Doshi, K. A., "A Framework for Application Guidance in Virtual Memory Systems," in *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '13, 2013.

[65] Josephson, W. K., Bongo, L. A., Li, K., and Flynn, D., "DFS: A File System for Virtualized Flash Storage," *ACM Trans. Storage*, vol. 6, Sept. 2010.

[66] Kannan, S., Gavrilovska, A., and Schwan, K., "Reducing the cost of persistence for nonvolatile heaps in end user devices," in *High Performance Computer Architecture (HPCA), 2014 IEEE 20th International Symposium on*, 2014.

[67] Kim, H., Seshadri, S., Dickey, C. L., and Chiu, L., "Evaluating Phase Change Memory for Enterprise Storage Systems: A Study of Caching and Tiering Approaches," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, FAST'14, 2014.

[68] Kwak, H., Lee, C., Park, H., and Moon, S., "What is Twitter, a Social Network or a News Media?," in *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, 2010.

[69] Lantz, P., Dulloor, S., Kumar, S., Sankaran, R., and Jackson, J., "Yat: A validation framework for persistent memory software," in *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, 2014.

[70] Lee, M. and Schwan, K., "Region Scheduling: Efficiently Using the Cache Architectures via Page-level Affinity," in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, 2012.

[71] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F., "Disaggregated Memory for Expansion and Sharing in Blade Servers," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[72] Loh, G. and Hill, M. D., "Supporting Very Large DRAM Caches with Compound-Access Scheduling and MissMap," *IEEE Micro*, vol. 32, May 2012.

[73] Loh, G. H., "3D-Stacked Memory Architectures for Multi-core Processors," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, 2008.

[74] Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K., "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the Conference on Programming Language Design and Implementation*, pp. 190–200, ACM, 2005.

[75] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G., "Pregel: A System for Large-scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, 2010.

[76] Malicevic, J., Dulloor, S. R., Sundaram, N., Satish, N., Jackson, J., and Zwaenepoel, W., "Exploiting nvm in large-scale graph analytics," in *Proceedings of the 3rd Workshop on Interactions of NVM/FLASH with Operating Systems and Workloads*, INFLOW '15, 2015.

[77] Mesnier, M. P. and Akers, J. B., "Differentiated Storage Services," *SIGOPS Oper. Syst. Rev.*, vol. 45, Feb. 2011.

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

[79] Nguyen, D., Lenharth, A., and Pingali, K., "A Lightweight Infrastructure for Graph Analytics," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[80] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T., and Venkataramani, V., "Scaling Memcache at Facebook," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, 2013.

[81] Oukid, I., Booss, D., Lehner, W., Bumbulis, P., and Willhalm, T., "SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery," in *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, 2014.

[82] Oukid, I., Lehner, W., Thomas, K., Willhalm, T., and Bumbulis, P., "Instant Recovery for Main-Memory Databases," in *Proceedings of the Seventh Biennial Conference on Innovative Data Systems Research*, CIDR '15, 2015.

[83] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R., "The Case for RAMClouds: Scalable High-performance Storage Entirely in DRAM," *SIGOPS Oper. Syst. Rev.*, vol. 43, Jan. 2010.

[84] Park, S., Kelly, T., and Shen, K., "Failure-atomic Msync(): A Simple and Efficient Mechanism for Preserving the Integrity of Durable Data," in *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, 2013.

[85] Prabhakaran, V., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Model-Based Failure Analysis of Journaling File Systems," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, DSN '05, 2005.

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

[87] Qureshi, M. K., Franceschini, M. M., Jagmohan, A., and Lastras, L. A., "PreSET: Improving Performance of Phase Change Memories by Exploiting Asymmetry in Write Times," *SIGARCH Comput. Archit. News*, vol. 40, June 2012.

[88] Qureshi, M. K. and Loh, G. H., "Fundamental Latency Trade-off in Architecting DRAM Caches: Outperforming Impractical SRAM-Tags with a Simple and Practical Design," in *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-45, 2012.

[89] Qureshi, M. K., Lynch, D. N., Mutlu, O., and Patt, Y. N., "A Case for MLP-Aware Cache Replacement," in *Proceedings of the 33rd Annual International Symposium on Computer Architecture*, ISCA '06, 2006.

[90] Qureshi, M. K., Srinivasan, V., and Rivers, J. A., "Scalable High Performance Main Memory System Using Phase-change Memory Technology," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, 2009.

[91] Rosenblum, M. and Ousterhout, J. K., "The Design and Implementation of a Log-structured File System," *ACM Trans. Comput. Syst.*, vol. 10, Feb. 1992.

[92] Roy, A., Mihailovic, I., and Zwaenepoel, W., "X-Stream: Edge-centric Graph Processing Using Streaming Partitions," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 2013.

[93] Satish, N., Sundaram, N., Patwary, M. M. A., Seo, J., Park, J., Hassaan, M. A., Sengupta, S., Yin, Z., and Dubey, P., "Navigating the Maze of Graph Analytics Frameworks Using Massive Graph Datasets," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, 2014.

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

[95] Sivathanu, M., Prabhakaran, V., Popovici, F. I., Denehy, T. E., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H., "Semantically-Smart Disk Systems," in *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, FAST '03, 2003.

[96] Steensgaard, B., "Points-to analysis in almost linear time," in *Proceedings of the Symposium on Principles of Programming Languages*, pp. 32–41, ACM, 1996.

[97] Stonebraker, M., Madden, S., Abadi, D. J., Harizopoulos, S., Hachem, N., and Helland, P., "The End of an Architectural Era: (It's Time for a Complete Rewrite)," in *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, 2007.

[98] Sundaram, N., Satish, N., Patwary, M. M. A., Dulloor, S. R., Vadlamudi, S. G., Das, D., and Dubey, P., "GraphMat: High performance graph analytics made productive." `http://arxiv.org/abs/1503.07241`, 2015.

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

[100] "The VMWare ESX Server." `http://www.vmware.com/products/esx/`.

[101] Volos, H., Tack, A. J., and Swift, M. M., “Mnemosyne: Lightweight Persistent Memory,” in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVI, 2011.

[102] Wang, A.-I. A., Kuenning, G., Reiher, P., and Popek, G., “The Conquest File System: Better performance Through A Disk/Persistent-RAM Hybrid Design,” *ACM Trans. Storage*, vol. 2, Aug. 2006.

[103] Wu, M. and Zwaenepoel, W., “eNVy: A Non-volatile, Main Memory Storage System,” in *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS VI, 1994.

[104] Wu, X. and Reddy, A. L. N., “SCMFS: A File System for Storage Class Memory,” in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC ’11, 2011.

[105] Yang, J., Minturn, D. B., and Hady, F., “When Poll is Better than Interrupt,” in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, FAST’12, 2012.

[106] Yoon, J. H., Hunter, H. C., and Tressler, G. A. I. C., “Flash and dram si scaling challenges, emerging non-volatile memory technology enablement - implications to enterprise storage and server compute systems,” in *Flash Memory Summit 2013 Proceedings*, Flash Memory Summit ’13, 2013.

[107] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M. J., Shenker, S., and Stoica, I., “Resilient Distributed Datasets: A Fault-tolerant Abstraction for In-memory Cluster Computing,” in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI’12, 2012.

[108] Zhang, Y., Yang, J., Memaripour, A., and Swanson, S., “Mojim: A Reliable and Highly-Available Non-Volatile Memory System,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS ’15, 2015.

VITA

Subramanya Dulloor is a PhD candidate at the College of Computing, Georgia Institute of Technology. He is also a member of the NSF-sponsored Center for Experimental Research in Computer Systems (CERCS) which conducts research in the domains of Enterprise, High Performance, and Embedded Systems. His research interests include emerging persistent memories, file systems, virtualization, and large-scale data center systems. He holds a Master of Science in Computer Science from Georgia Institute of Technology and a Bachelor of Technology in Computer Science from B.M.S College of Engineering, Bangalore, India.