A Case for Efficient Hardware/Software Cooperative Management of Storage and Memory

Justin Meza[*] Yixin Luo[*] Samira Khan[*‡] Jishen Zhao[†] Yuan Xie[†§] Onur Mutlu[*]
[*]Carnegie Mellon University [†]Pennsylvania State University [‡]Intel Labs [§]AMD Research

Abstract

Most applications manipulate persistent data, yet traditional systems decouple data manipulation from persistence in a two-level storage model. Programming languages and system software manipulate data in one set of formats in volatile main memory (DRAM) using a load/store interface, while storage systems maintain persistence in another set of formats in non-volatile memories, such as Flash and hard disk drives in traditional systems, using a file system interface. Unfortunately, such an approach suffers from the system performance and energy overheads of locating data, moving data, and translating data between the different formats of these two levels of storage that are accessed via two vastly different interfaces.

Yet today, new non-volatile memory (NVM) technologies show the promise of storage capacity and endurance similar to or better than Flash at latencies comparable to DRAM, making them prime candidates for providing applications a persistent single-level store with a single load/store interface to access all system data. Our key insight is that in future systems equipped with NVM, the energy consumed executing operating system and file system code to access persistent data in traditional systems becomes an increasingly large contributor to total energy. The goal of this work is to explore the design of a Persistent Memory Manager *that coordinates the management of memory and storage under a single hardware unit in a single address space. Our initial simulation-based exploration shows that such a system with a persistent memory can improve energy efficiency and performance by eliminating the instructions and data movement traditionally used to perform I/O operations.*

1. Introduction and Motivation

Applications—whether those running on a mobile platform or on a server in a warehouse-scale computer—create, modify, and process persistent data. Yet, though manipulating persistent data is a fundamental property of most applications, traditional systems decouple data manipulation from persistence, and provide different programming interfaces for each. For example, programming languages and system software manipulate data in one set of formats using load and store instructions issued to volatile memory (DRAM), while storage systems maintain persistence in another set of file system formats in non-volatile memories, such as hard disk drives and Flash. Unfortunately, such a decoupled memory/storage model suffers from large inefficiencies in locating data, moving data, and translating data between the different formats of these two levels of storage that are accessed via two vastly different interfaces—leading to potentially large amounts of wasted work and energy. Such wasted energy significantly affects modern devices. Cell phones need to be energy efficient to minimize battery usage, and servers must be energy efficient as the cost of energy in data centers is a substantial portion of the total cost [8]. With energy as a key constraint, and in light of modern high-density, non-volatile devices, we believe it is time to rethink the relationship between storage and memory to improve system performance and energy efficiency.

While such disparate memory and storage interfaces arose in systems due to the widely different access latencies of traditional memory and storage devices (i.e., fast but low capacity DRAM versus slow but high capacity hard disk drives), such stereotypical device characteristics are being challenged by new research in non-volatile memory (NVM) devices. New byte-addressable NVM technologies such as phase-change memory (PCM) [46], spin-transfer torque RAM (STT-RAM) [12, 25], and resistive RAM (RRAM), are expected to have storage capacity and endurance similar to Flash—at latencies comparable to DRAM. These characteristics make them prime candidates for use in new system designs that provide applications with a byte-addressable interface to both volatile memory and non-volatile storage devices.

In this way, emerging high-performance NVM technologies enable a renewed focus on the unification of storage and memory: a hardware-accelerated single-level store, or *persistent memory*, which exposes a large, persistent virtual address space supported by hardware-accelerated management of heterogeneous storage and memory devices. The implications of such an interface for system efficiency are immense: A persistent memory can provide a unified load/store-like interface to access all data in a system without the overhead of software-managed metadata storage and retrieval and with hardware-assisted data persistence guarantees. Previous single-level store proposals either focused on providing persistence through the software layer (e.g., [4, 16, 42, 45]), or were inefficient in providing hardware support for fast and cheap data access, migration, and protection (e.g., [9, 24, 40, 41, 43]). As we will show, with the use of emerging NVM technologies, the energy required to execute operating system and file system code to access persistent data will consume a large amount of total system energy, presenting an opportunity for coordinating the management of memory and storage between hardware and software in a single address space. Such a system can improve energy efficiency by reducing the amount of work performed by the operating system and file system when accessing persistent data, and also reduce execution time.

For example, Figure 1 shows a breakdown of how energy is consumed in three systems running an I/O-intensive program on Linux with: (left) a traditional HDD-based storage system, (middle) a system with the HDD replaced with NVM, and (right) a persistent memory system with NVM for all storage and memory (our experimental methodology is discussed in Section 3). It shows the number of cy-

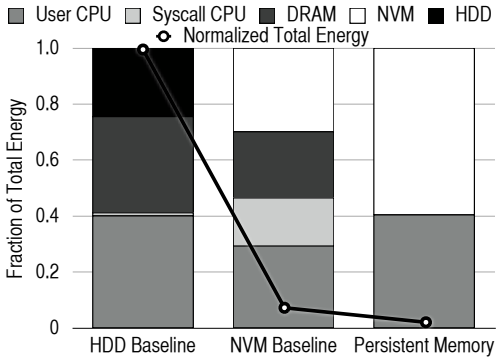

Figure 1: A breakdown of energy consumption for an I/O-intensive benchmark, PostMark.

cles spent executing user code (User CPU), stalling on user memory requests (User Memory), executing system call code (Syscall CPU), and performing device I/O during a system call (Syscall I/O).

While the HDD-based system's energy consumption is dominated by accessing the disk and the static energy of DRAM, in the NVM-based system, this fraction of energy is greatly reduced (as seen in the Normalized Total Energy), due to the energy-efficient nature of the device, causing the energy to execute OS and file system code (Syscall CPU), and buffer large amounts of data in DRAM (DRAM), to arise as large contributors to system energy. Using a persistent memory can eliminate the fraction of energy consumed by these operations by reducing the overhead of accessing persistent data, reducing the energy to access the same amount of data (which reduces User CPU). Note that we assume a scenario where the management of the persistent memory has a negligible overhead in terms of performance and energy in order to isolate this trend from a specific implementation.

However, integrating a persistent memory into systems poses several key challenges. How can data storage and retrieval be done efficiently using a unified, single address space? How can consistency, persistence, and security of data be maintained? How can such a system be backwards-compatible with applications written with traditional file operations, and be supported efficiently? How can applications be designed to efficiently leverage such a system? In this paper, we explore some of these questions and provide an initial evaluation of how an efficient hardware/software cooperative persistent memory can mitigate the energy overheads of traditional two-level stores.

2. Hardware/Software Cooperative Management of Storage and Memory

Our key idea is to coordinate the management of memory and storage under a single hardware unit in a single address space to leverage the high capacity and byte-addressability of new persistent memories, to eliminate the operating system and file system overheads of managing persistent data. Instead of executing operating system and file system code to manipulate persistent data, programs can issue loads and stores—just as if the files were mapped in memory (except their contents will remain persistent). This can improve performance and reduce energy by reducing the amount of work that must be performed (i.e., the number of instructions that must be executed, or the amount of data moved when performing a direct memory access) in order to access and manipulate persistent data.

Figure 2 shows one potential strategy for enabling efficient hardware support for persistent memory. A hardware-accelerated *Persistent Memory Manager (PMM)* is used to coordinate the storage and retrieval of data in a persistent memory. The PMM provides a load/store interface for accessing data, and is responsible for handling data layout and migration among devices, persistence management and journaling, metadata storage and retrieval, and encryption and security. In addition, it exposes hooks and interfaces for the software, the OS, and the language runtime to exploit the devices in the persistent memory.

As an example, in the code listing at the top of Figure 2, the `main` function creates a new persistent object (we use a similar notation to C's `FILE` structure) with the *handle* "file.dat"* and sets its contents to a newly-allocated array of 64 integers. Later (perhaps in between

*Note that the underlying hardware may choose to assign this handle a more machine-friendly form, such as a fixed n-bit value.

```
1 int main(void) {
2   // data in file.dat is persistent
3   FILE myData = "file.dat";
4   myData = new int[64];
5 }
6 void updateValue(int n, int value) {
7   FILE myData = "file.dat";
8   myData[n] = value; // value is persistent
9 }
```

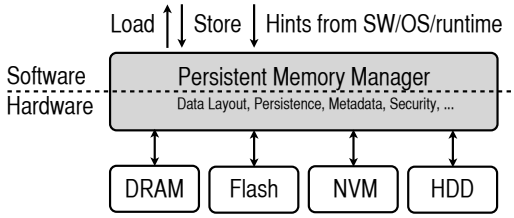

Figure 2: Hardware-based storage and memory management.

executions of the same program, or after restarting the system), in the `updateValue` function, the persistent object with the same handle ("file.dat") is accessed, and the value of one of its elements is updated, just as it would be if it were allocated in memory, except that now the PMM ensures that its data is mapped to persistent memory, can be located efficiently, and will remain persistent even if the machine were to crash after the program updates the value. This behavior should be satisfied even in the presence of a storage area network (SAN), or other storage in a distributed system, as well, and could extend existing software-based work in persistent main memories [33] with more efficient hardware support.

A hardware/software cooperative approach to providing persistent memory yields three major benefits. First, it can eliminate the execution of instructions in the operating system and file system that are typically used to manipulate persistent data (i.e., system calls such as `open`, `read`, and `write`, and file system management code such as that responsible for inode lookup and journaling). Second, employing persistent memory reduces the data movement associated with accessing persistent data from block-based devices such as HDD or Flash, which typically involves copying data from the device to a buffer in DRAM—even if it is then immediately written back to the persistent device. Third, it can help exploit the full parallelism and bandwidth of the underlying devices in the persistent memory. All of these benefits can lead to significant improvements in system performance and energy consumption for applications that manipulate persistent data.

2.1. Comparison to Related Work

To our knowledge, this is the first paper that explores the implications of efficient hardware-based coordinated management of storage and memory for improved system energy efficiency and performance.

Prior works proposed to take advantage of emerging non-volatile memory by integrating file systems with persistent memory (e.g., [6, 7, 13, 15, 17, 28]), providing software support for checkpointing distributed system states (e.g., [48]) and providing hardware support to manage Flash devices [47] or NVM devices for storage purposes [11]. To fully exploit the advantages of these technologies and reduce programmer burden, optimized hardware-software cooperative support needs to be designed for systems that are aware of the various trade-offs and limitations of their non-volatile devices.

Providing programming language support for persistent objects has been proposed previously (e.g., [2–5, 10, 14, 23, 34, 39, 45]). None of these works take into account hardware management of persistent objects. Persistence guarantees within the software layer

result in slower memory access latency due to the overhead of the extra indirection. Our work considers designs geared toward hardware with faster and cheaper persistent memory that can achieve high performance in the presence of volatile *and* non-volatile memory.

Previous works examined single-level stores where data was accessed through load and store instructions (e.g., [9, 16, 20, 24, 40–43]). Some of these works focused on a software interface for accessing data from a single-level store [16, 42], while others (e.g., [9, 20, 24, 40, 41, 43]) provided a unified address space through hardware (e.g., data indexed using hardware-based hierarchical tables [9, 20]). However, these works did not focus on how to provide efficient and fast hardware support for address translation, efficient file indexing, or fast reliability and protection guarantees. The goal of our work is to provide cheap and fast hardware support in persistent memories that enable high energy efficiency and performance. Oikawa [35] provides a good evaluation of one design point, and we quantify the potential benefits of research in this direction.

Others have shown that the OS contributes significantly to the latency required to service accesses to Flash devices [44]. Our study corroborates their findings, and shows even larger consequences for emerging NVM devices in terms of performance and energy-efficiency.

We discuss several opportunities, benefits, and challenges from using a hardware-based approach to managing storage and memory that affect performance, energy, security, and reliability next, and later complete our discussion with some open questions and potential downsides in Section 3.4. It is important to note that this initial work is primarily focused on an exploration of the potential for employing persistent memory in future systems; we plan to explore more detailed implementation strategies in future works.

Eliminating system calls for file operations. A persistent memory allows all data to reside in a linear, byte-addressable virtual address space. This allows data elements to be directly manipulated using traditional load/store instructions, while still remaining persistent. This eliminates the need for using layers of operating system code (associated with system calls such as `open`, `read`, and `write`) for managing metadata associated with files, such as file descriptors (metadata about open files), file buffers (space allocated in volatile memory for manipulating small portions of files), and so on.

Eliminating file system operations. Locating data in a persistent memory can be done quickly with efficient hardware support. Just as processors use Memory Management Units (MMUs) to efficiently traverse operating system managed page tables, and Translation Lookaside Buffers (TLBs) to cache virtual-to-physical page address translations to efficiently access memory, we envision future systems that employ hardware techniques to efficiently store and retrieve persistent data from memory. This can eliminate the code required by traditional file systems that, for example, locates a file's inode, or performs journaling to improve reliability.

Efficient data mapping. A persistent memory gives the illusion of a single, persistent store for manipulating data. Yet, the underlying hardware that comprises the persistent memory may actually consist of a diverse array of devices: from fast, but low-capacity volatile devices such as DRAM; to slower, but high-capacity non-volatile devices such as non-volatile memories, Flash, and hard disk drives. With such an assortment of device characteristics, it will become increasingly challenging for the programmer to decide how to partition their data among them, and we believe the software and hardware together can help here by observing access patterns and tailoring data placement on devices based on its access characteristics.

Figure 3 shows two axes we may consider when making decisions about mapping data in a persistent memory—locality and persistence—and suggests examples of data in applications that might be classified in each category[†]. By monitoring how the data in a program is accessed, with hints from the operating system and programming language runtime (e.g., by tagging or classifying I/O requests [19, 29]), we think that a hardware structure such as the Persistent Memory Manager can be used to make intelligent data placement decisions, at runtime. For example, a frequently-referenced and frequently-updated index may be placed on a DRAM cache, but the persistent data it indexes may reside on non-volatile memory.

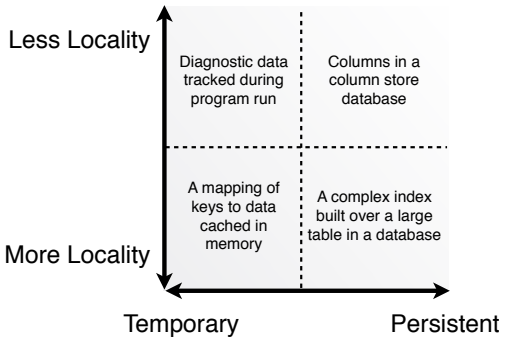

Figure 3: The locality and persistence traits of data affect placement among devices in a persistent memory. This figure gives a potential example categorization.

2.2. Opportunities, Benefits, and Challenges

Providing security and reliability. Using a persistent memory can provide opportunities for maintaining data security at a granularity smaller than files, and potentially even smaller than existing pages in virtual memory. Finding the right balance between security metadata storage and security guarantees will be a key challenge. Integrating volatile storage into the persistent memory will improve system performance, but will also require careful coordination among the devices to ensure that any hardware-managed volatile copies of persistent data obey the ACID properties (atomicity, consistency, isolation, and durability). In systems where a persistent memory is distributed across multiple machines, ensuring data security and reliability will also be an important challenge.

Hardware/software cooperative data management. By providing the appropriate hooks and interfaces to applications, the OS, and the runtime, persistent memories can enable improved system robustness and efficiency. As a simple example, persistent memories can enable fast checkpointing and fast reboots raising the question of how system availability mechanisms can be designed to take advantage of this behavior. It will be important to examine application-transparent (hardware/software cooperative) mechanisms for checkpointing and fast restart as well as application, OS, and runtime hooks and interfaces to take advantage of fast checkpointing and bootup features. Such techniques can enhance distributed system availability and reliability by making use of persistent locks and other persistent synchronization constructs that can survive system power events.

[†]Note that we have highlighted two salient axes, though others also exist (such as reliability, latency sensitivity, or bandwidth requirement).

3. Initial Exploration

We have begun examining the effects of eliminating the code and data movement traditionally required to manage persistent data. Our early experimentation along these lines has focused on studying the effects of eliminating the overhead of performing system calls in the operating system and file system code used to manage persistent data. We use a hybrid real system/simulation-based approach for performing our initial persistent memory evaluation because it makes future studies involving hardware modifications feasible.

In our study, we simulate the execution of several applications using the Multi2Sim simulator [1]. We have modified the simulator so that when a file I/O system call is performed in the program (our programs use `open`, `close`, `read`, `write`, `pread`, `pwrite`, and `mmap`), we track the number of cycles spent servicing the system call on the host machine. On our baseline system that does not employ a persistent memory, these cycles spent in the OS and file system code are then added to the simulation after the system call is performed, and execution continues. Detailed simulation and system parameters are listed in Table 1.

To estimate the effects of using an emerging non-volatile memory as a storage device, we separated the time spent performing file I/O system calls into two categories: (1) executing the system call and (2) accessing the I/O device. The cycles needed to execute the system call, as well as the effect of a buffer cache, are measured on a real system that has the same configuration as the simulator. The benchmarks we used are shown in Table 2[‡], and include a variety of workloads of varying I/O and compute intensity. In addition to examining microbenchmarks composed of common Unix utilities, we also examined PostMark, a file system benchmark from NetApp [22], and the MySQL database server. We show the average across five runs of each experiment and the standard deviation is reported above each bar. We cleared the disk buffer cache before running each experiment and we warmed up the system's state through the initialization of MySQL server before collecting results. For MySQL (simple) and MySQL (complex), we cleared the disk cache in between each request to simulate the effects of querying a much larger database in the random way that that our workload, Sysbench, does. For this reason, our MySQL results do not have a standard deviation associated with them.

Processor	16 cores, 4-wide issue, 128-entry instruction window, 1.6 GHz.
L1 cache	Private 32 KB per core, 2-way, 64 B blocks, 3-cycle latency.
L2 cache	Shared 4 MB, 8-way, 128 B blocks, 16 MSHRs, 25-cycle latency.
Memory	DRAM: 4 GB, 4 KB pages, 100-cycle latency. NVM: 4 GB, 4 KB pages, 160-/480-cycle (read/write) latency. [26] HDD: 4 ms seek, 6 Gbps bus rate.
Power	Processor and caches: 1.41 W average dynamic power per core / 149 W peak power [27]. DRAM: 100 mW/chip static power [32], row buffer read (write): 0.93 (1.02) pJ/bit, array read (write): 1.17 (0.39) pJ/bit. [26] NVM: 45 mW/chip static power [18], row buffer read (write): 0.93 (1.02) pJ/bit, array read (write): 2.47 (16.82) pJ/bit. [26] HDD: 1W average power

Table 1: Major simulation parameters.

cp	Copy an 80 MB file.
cp -r	Copy a directory that contains 700 folders and 700 1-byte files.
grep	Read through a 10 MB file and search for a keyword.
grep -r	Read through a directory that contains 700 folders and 700 1-byte files and search for a keyword.
PostMark [22]	Perform 512 B random reads/writes 500 times on 500 files that are between 500 B and 10 KB in size.
MySQL [36]	MySQL server; OLTP queries generated by SysBench. 1 M-entry database; 200 B per entry. simple: Repeatedly reads random table entries. complex: Reads/writes 1 to 100 table entries.

Table 2: Benchmark characteristics.

We considered four system configurations in our study:

HDD Baseline (HB): This system is similar to modern systems that employ volatile DRAM memory and hard disk drives for storage. When a file operation like `fread` or `fwrite` is performed, file system code is used to locate the file on the disk, and operating system code acts on behalf of the program to perform a direct memory access to transfer data from the disk to a DRAM buffer.

HDD without OS/FS (HW): This system is similar to the HDD Baseline, but represents a situation where all operating system and file system code executes instantly. File operations still work on buffered copies of data in DRAM and device access latencies are the same as the HDD Baseline.

NVM Baseline (NB): In this system, the HDD is replaced with NVM. All other operations are similar to the HDD Baseline case (i.e., OS and file system code are still executed).

Persistent Memory (PM): In this system, to achieve full-memory persistence, *only* NVM is used (i.e., there is no DRAM) and applications access data using loads and stores directly from and to NVM (i.e., there is no operating system and file system code, and data is manipulated directly where it is located on the NVM device). In our simulator, whenever a system call is performed in the program, we executed the system call on the host machine (so that our simulation would behave functionally the same as before), but *did not add* the time spent performing the system call to the simulation, and only included the time to access the NVM device. This design would use a hardware-based Persistent Memory Manager, like we are proposing, to coordinate accesses to the NVM device and has the potential for much optimization. We will suggest potential directions for improving this system in Section 3.4.

3.1. Performance

Figure 4 shows how these configurations fare in terms of execution time, normalized to the HDD Baseline for our benchmarks. For each of the systems on the x axis, we show the number of cycles spent executing user code (User CPU), stalling on user memory requests (User Memory), executing system call code (Syscall CPU), and performing device I/O during a system call (Syscall I/O).

Looking at the HDD Baseline (HB) and the HDD without OS/FS (HW) data points, we see that eliminating operating system and file

[‡] We are actively exploring a larger-scale evaluation of popular databases and key-value stores, where we expect our findings to hold due to the scale at which they manipulate persistent data.

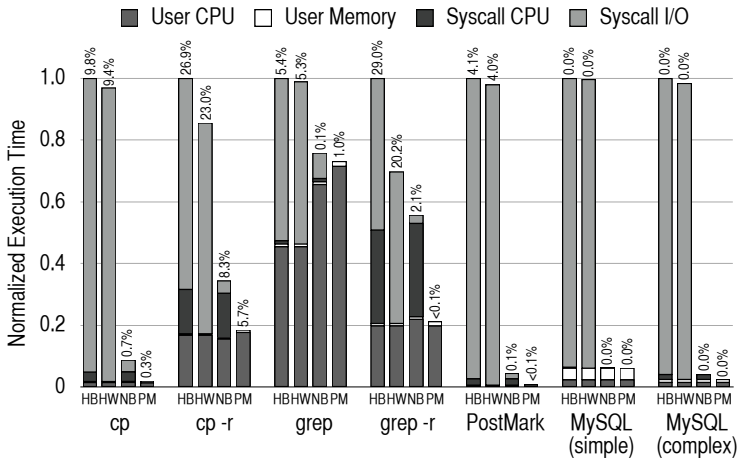

Figure 4: Normalized execution time (standard deviation shown above bars).

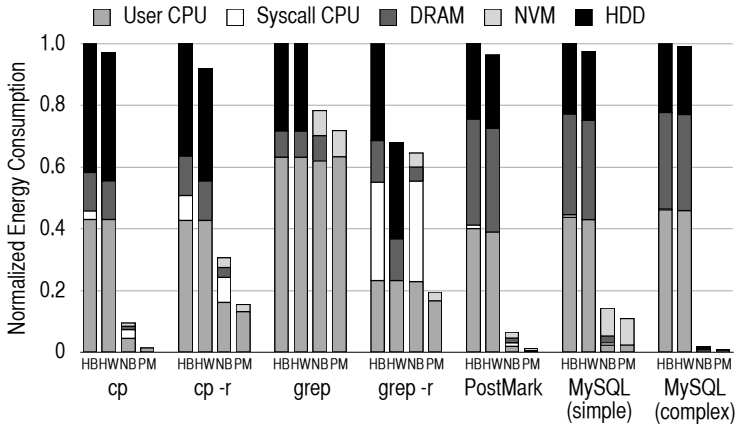

Figure 5: Normalized energy consumption.

system code—in the ideal case—leads to performance improvement by reducing the amount of work performed to manipulate the same amount of data, though not by a very large amount, because the execution time on these systems is dominated by the HDD access time. Notice that grep hardly benefits from eliminating the work performed in OS/FS code, because its execution is dominated by its string-matching computation.

Comparing the HDD Baseline (HB) to the NVM Baseline (NB), we see that replacing the hard disk drive with NVM can greatly reduce execution time for I/O-bound workloads (those besides grep). Yet such an approach still inherits the inefficiency from HDD-based systems of executing system call and file system code, allocating buffers in DRAM for managing file data, and moving data between NVM and DRAM, as seen in the Syscall CPU and Syscall I/O time. Using a Persistent Memory (PM) causes some of the Syscall I/O time to instead be spent by User Memory accesses, but in general performance is greatly improved for I/O-intensive workloads, such as PostMark, compared to the NVM Baseline (NB) by eliminating the traditional two-level storage interface.

Though both MySQL simple and complex benefit from using a Persistent Memory, the benefits are more pronounced with MySQL complex, due to its more I/O-intensive nature. For the CPU-bound benchmark grep, the PM system hardly improves performance because it does not employ DRAM, and grep makes good use of locality in DRAM. This negative effect can be mitigated with intelligent management of a DRAM cache, as others have shown in the context of heterogeneous main memories [21, 30, 31, 37, 38, 49]. Developing policies for data placement in heterogeneous memories employing some amount of fast DRAM is a promising direction.

From this preliminary analysis, we conclude that a system with a persistent memory can significantly improve performance by eliminating the instructions and data movement traditionally used to perform I/O operations and also reducing average storage access latency.

3.2. Energy

Energy is an increasingly important constraint in devices from cell phones (where battery life is essential), to data centers (where energy costs can be a substantial portion of total cost of ownership [8]). By improving performance and reducing the number of instructions executed by the processor (i.e., those that would have been executed for system call or for file system code) programs executing on a persistent memory can finish more quickly and perform less work, consuming less static *and* dynamic device energy.

We modeled energy using the McPAT [27] framework and the values assumed for device power are shown in Table 1. Figure 5 breaks down the energy consumed by the system configurations we evaluated for each of the benchmarks, normalized to the energy of

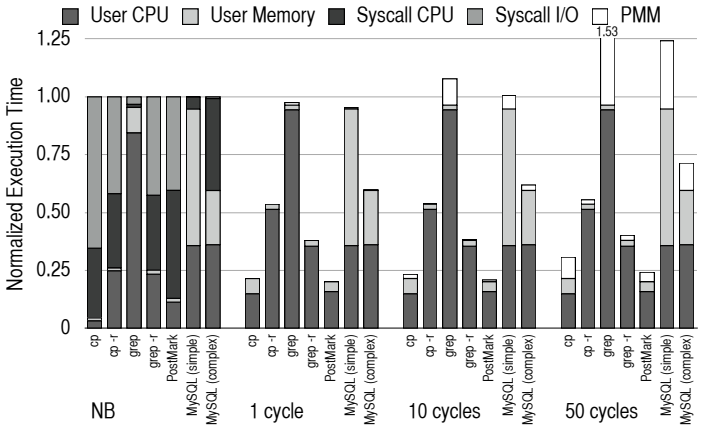

Figure 6: Execution time compared to the NVM Baseline (NB) system with different PMM latencies.

the HDD Baseline configuration. For each of the configurations labeled on the x axis, we show the energy consumed by the processor on user code (User CPU), system call code (Syscall CPU), DRAM, NVM, and HDD (when these devices are used in a system). Note that storage device (HDD or NVM) energy consumption is dominated by static device energy, even under I/O-intensive operation. For the CPU, neither static nor dynamic energy consumption dominates total energy.

In the HDD-based systems (HB and HW), energy consumption is dominated by the energy of the HDD device. While eliminating system call code can reduce static energy consumption by improving performance, it does not greatly reduce total system energy. For I/O-bound benchmarks, using NVM instead of disk (NB) helps performance and energy by using a much faster storage device (in this case, PCM). Interestingly, in these systems, the overheads for energy of the CPU and the code it has to execute to perform I/O operations becomes a significant portion of total system energy. Switching from a traditional storage model (NB) to a persistent memory (PM) helps tackle this overhead and reduces energy significantly for two main reasons:

1. Reduced code footprint. Eliminating system call code reduces CPU dynamic energy consumption because there are fewer instructions for the CPU to execute to perform the same amount of work. This leads to energy benefits (even for grep, despite its small improvement in execution time).

2. Reduced data movement. Eliminating device access though the OS reduces the amount of data movement in the system (i.e., accessing persistent data does not require a direct memory access operation to transfer data into a buffer in memory), reducing the execution time of the benchmark, and improving static device energy consumption. As NVM energy is dominated by static energy consumption, this can be seen on most of the benchmarks in the reduction of NVM energy.

We believe that the adoption of persistent memory can help significantly reduce system energy consumption, while providing full-memory persistence in future devices.

3.3. Scalability

So far, we have shown results for a PMM with no additional latency, in order to quantify the potential benefits of a persistent memory. We now look at the impact of various fixed-cycle PMM latencies on program performance.

Figure 6 shows the normalized execution time (compared to the NB system) broken down in the same way as Figure 4, but with an added category for PMM latency (PMM), for our benchmarks on the Persistent Memory system assuming a PMM latency of 1, 10, and 50 cycles. We account for the PMM latency on every memory reference, and our estimates are pessimistic in the sense that they do not consider the possibility of overlapping the PMM latency of multiple outstanding requests. Even so, with a 1- or 10-cycle latency per reference, performance is only slightly degraded on most benchmarks. At a 50-cycle latency per reference, however, the more I/O-intensive benchmarks (grep and MySQL (simple)), spend a considerable amount of their time in the PMM. Still, in most cases, this does not lead to performance degradation compared to the NVM Baseline system. As we will discuss next, designing an efficient PMM will be a key concern, though our results suggest that there exists a wide range of PMM latencies that lead to overall system efficiency improvements.

3.4. Open Research Questions and Challenges

There are several open research questions that arise and downsides that must be mitigated when providing efficient hardware support for persistent memory, which we discuss next.

Q1. How to tailor applications for systems with persistent memory? To fully achieve the potential energy and performance benefits of a persistent memory, we would like full support from applications. This leads to several questions, such as: (1) How should the runtime system (and the programmer/compiler) partition data into persistent versus temporary data and marshal the communication among different storage, memory, and computation components in a system? (2) How can data be more efficiently transmitted between storage and memory? (3) How can software be designed to improve the reliability and fault-tolerance of systems with persistent memories? (4) How does persistent memory change the way high availability is provided in modern applications and systems, especially distributed systems?

Q2. How can hardware and software cooperate to support a scalable, persistent single-level address space? Like virtual memory in modern systems, we envision future persistent memory systems will tightly integrate system hardware and software, leading to several key design questions, such as: (1) What is the appropriate interface between a persistent memory manager and system and application software? (2) What application-, OS-, and runtime-level hooks and interfaces will be most useful for enabling efficient persistent memo-

ries? (3) How can software provide hints to the hardware regarding the characteristics of data (e.g., locality, persistence, security, latency, and bandwidth requirements)? (4) How can system software provide consistency, reliability, and protection for data in a persistent memory system when the memory hierarchy consists of both volatile and non-volatile storage?

Q3. How to provide efficient backward compatibility on persistent memory systems? (1) How do we efficiently support legacy page table and file system functionality in a persistent memory with hardware support to reduce system software overhead? (2) How can the devices that comprise a persistent memory best be used to store the data and metadata associated with existing memory and storage interfaces?

Q4. How to mitigate potential hardware performance and energy overheads? By supporting persistent memory in hardware, we may introduce performance and energy overheads due to the addition of new hardware structures. A key question is how to mitigate these overheads. We plan to examine how these overheads can be kept low, making persistent memory a desirable option.

We believe at least these questions are important to investigate in enabling efficient hardware support for persistent memories, and we intend to explore these questions in our future work.

4. Summary

Energy is a key constraint in future devices from cell phones to servers, and efficient storage organizations can drastically improve the performance and reduce the energy consumption of such systems. We have shown that in future systems employing NVM devices for persistence, the energy required to execute OS and file system code, as well as the energy required to transfer data to and from DRAM and persistent memory, constitutes a large portion of total energy. Based on this observation, we explored the design of a hardware/software cooperative Persistent Memory Manager that coordinates the management of memory and storage in a single address space, taking into account main memory data layout, migration, persistence management, metadata storage and retrieves, encryption/security, and so on, for improved system energy efficiency. We identified several research questions along these lines, which we intend to explore in our future work.

References

[1] Multi2Sim. http://www.multi2sim.org/.
[2] A. Albano, L. Cardelli, and R. Orsini. GALILEO: a strongly-typed, interactive conceptual language. TODS 1985.
[3] M. Atkinson, K. Chisholm, and P. Cockshott. PS-Algol: an Algol with a persistent heap. *ACM SIGPLAN Notices*, 1982.
[4] M. Atkinson et al. An approach to persistent programming. *The Computer Journal*, 1983.
[5] M. Atkinson et al. An orthogonally persistent Java. *ACM SIGMOD Record*, 1996.
[6] K. Bailey et al. Operating system implications of fast, cheap, non-volatile memory. HotOS, 2011.
[7] M. Baker et al. Non-volatile memory for fast, reliable file systems. ASPLOS 1992.
[8] L. A. Barroso and U. Hölze. *The Datacenter as a Computer*. Morgan & Claypool, 2009.
[9] A. Bensoussan, C. T. Clingen, and R. C. Daley. The Multics virtual memory: concepts and design. *Communications of the ACM*, 1972.
[10] L. Cardelli. Amber. *Combinators and Functional Programming Languages*, 1986.
[11] A. M. Caulfield et al. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. MICRO 2010.
[12] E. Chen et al. Advances and future prospects of spin-transfer torque random access memory. *IEEE Transactions on Magnetics*, 46(6), 2010.
[13] P. M. Chen et al. The Rio file cache: surviving operating system crashes. ASPLOS 1996.
[14] J. Coburn et al. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. ASPLOS 2011.
[15] J. Condit et al. Better I/O through byte-addressable, persistent memory. SOSP 2009.
[16] G. Copeland et al. Uniform object management. EDBT 1990.
[17] G. Copeland et al. The case for safe RAM. VLDB 1989.
[18] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid PRAM and DRAM main memory system. DAC 2009.
[19] S. Faibish, P. Armangau, and C. Seibel. Application aware intelligent storage system. MSST 2013.
[20] C. Hunter et al. *Introduction to the Intel IAPX 432*. 1984.
[21] J.-Y. Jung and S. Cho. Memorage: emerging persistent ram based malleable main memory and storage architecture. ICS 2013.
[22] J. Katcher. PostMark: A new file system benchmark. Technical Report 3022, NetApp, 1997.
[23] S. Khoshafian and G. Copeland. *Object identity*. OOPSLA 1986.
[24] T. Kilburn et al. One-level storage system. *IEEE Transactions on Electronic Computers*, 1962.
[25] E. Kultursay et al. Evaluating STT-RAM as an energy-efficient main memory alternative. ISPASS 2013.
[26] B. C. Lee et al. Architecting phase change memory as a scalable DRAM alternative. ISCA 2009.
[27] S. Li et al. McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures. MICRO 2009.
[28] D. E. Lowell and P. M. Chen. Free transactions with Rio Vista. SOSP 1997.
[29] M. Mesnier et al. Differentiated storage services. SOSP 2011.
[30] J. Meza et al. Enabling efficient and scalable hybrid memories using fine-granularity DRAM cache management. *IEEE Computer Architecture Letters*, 2012.
[31] J. Meza, J. Li, and O. Mutlu. A case for small row buffers in non-volatile main memories. ICCD, Poster Session, 2012.
[32] Micron Technology. 4Gb: x4, x8, x16 DDR3 SDRAM, 2011.
[33] I. Moraru et al. Persistent, protected and cached: Building blocks for main memory data stores. Technical Report PDL-11-114, CMU, 2012.
[34] J. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 1992.
[35] S. Oikawa. Integrating memory management with a file system on a non-volatile main memory system. SAC 2013.
[36] Oracle. *MySQL Manual*. http://dev.mysql.com/doc/refman/5.6/en/.
[37] M. K. Qureshi and G. H. Loh. Fundamental latency trade-off in architecting DRAM caches: Outperforming impractical SRAM-tags with a simple and practical design. MICRO 2012.
[38] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. ISCA, 2009.
[39] J. E. Richardson et al. The design of the E programming language. *ACM Transactions on Programming Languages and Systems*, 1993.
[40] J. S. Shapiro and J. Adams. Design evolution of the EROS single-level store. ATEC 2002.
[41] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. SOSP 1999.
[42] E. Shekita and M. Zwilling. Cricket: A mapped, persistent object store. POS 1990.
[43] F. G. Soltis. *Inside the AS/400*. Twenty Ninth Street Press, 1996.
[44] V. R. Vasudevan. *Energy-efficient Data-intensive Computing with a Fast Array of Wimpy Nodes*. PhD thesis, CMU, 2011.
[45] S. J. White and D. J. Dewitt. QuickStore: A high performance mapped object store. SIGMOD 1994.
[46] H.-S. Wong et al. Phase change memory. *Proceedings of the IEEE*, 98(12), 2010.
[47] M. Wu and W. Zwaenepoel. eNVy: A non-volatile, main memory storage system. ASPLOS 1994.
[48] S. Yoo et al. Composable reliability for asynchronous systems. USENIX ATC 2012.
[49] H. Yoon et al. Row buffer locality-aware data placement in hybrid memories. ICCD, 2012.