Write-Aware Management of NVM-based Memory Extensions

Amro Awad
North Carolina State University
Raleigh, NC USA
ajawad@ncsu.edu

Sergey Blagodurov
Advanced Micro Devices, Inc.
Bellevue, WA USA
sergey.blagodurov@amd.com

Yan Solihin
North Carolina State University
Raleigh, NC USA
solihin@ncsu.edu

ABSTRACT

Emerging Non-Volatile Memory (NVM) technologies, such as 3D XPoint, are expected to be in production as early as 2016. Emerging NVMs are very attractive for several reasons. First, they are non-volatile and hence incur no refresh power. Second, they are dense and promising for scaling down further. Finally, they are fast and have latencies comparable to DRAM. On the other side, using emerging NVMs as direct replacement for DRAM as the main memory is challenging. Compared to DRAM, emerging NVMs can endure a very limited number of writes per cell. Furthermore, their write latency is typically much slower and more energy consuming than DRAM, e.g., Phase Change Memory (PCM) writes are multiple of times slower than that of DRAM. An important use case for emerging NVMs is using them as fast memory extensions. Memory extensions are hidden from programmers and managed by the Operating System (OS). Any access to pages held in the memory extension will cause a page fault. Later, the memory manager moves the faulting page to DRAM and maps the page. While similar in concept to the swap file, memory extensions bypass the file system. Furthermore, memory extensions are dedicated for being used as memory and hence avoid contention with the file system.

In this paper, we emulate an NVM-based memory extension and study its impact on performance on a real system. We also study how to improve its performance using OS-level prefetching. We show the importance of having the system software and the NVM controller work in concert for reducing the number of writes. Our best scheme where the system software and the NVM controller work in concert could reduce the number of writes to only 5% of the original baseline (increasing its lifetime by 20×).

CCS Concepts

•Computer systems organization → Heterogeneous (hybrid) systems;

ICS '16, June 01-June 03, 2016, Istanbul, Turkey

DOI: http://dx.doi.org/10.1145/2925426.2926284

Keywords

NVM Memory Extensions; PCM; System Software

1. INTRODUCTION

With the increasing pressure to deploy large main memories, the traditional use of DRAM as the only component of the main memory is increasingly becoming less attractive, for several reasons. First, the need to refresh volatile DRAM cells incurs large power consumption, which limits the amount of DRAM we can deploy in the same package. Second, scaling down DRAM cell size becomes difficult as the charge in DRAM cell capacitor needs to be kept constant to meet retention time requirements [13]. Considering these DRAM limitations, computer system designers are rightfully considering emerging Non-Volatile Memories (NVMs) as replacements for DRAM. NVMs are non-volatile and require no refresh power; some of them have a read latency comparable to DRAM, while at the same time they may scale better than DRAM [12, 2, 28].

On the other side, there are still serious challenges in using NVMs as the main memory. Writing to NVM cells is often slow, requires large power consumption, and has limited write endurance, e.g., 10-100 million writes in Phase Change Memory (PCM) [15, 32, 12, 2, 28]. Accordingly, future systems are expected to have heterogeneous memory systems that consist of both DRAM and dense emerging NVMs.

Emerging NVM products are expected to hit the market in the next few years. As an example, 3D XPoint technology has been announced by Intel and Micron with an expected arrival time of 2016 [4]. Emerging NVMs are expected to be offered in two major forms: Solid-State Drives (SSDs) and Dual-In-Line-Memory Modules (DIMMs). As shown in Figure 1, emerging NVM-based SSDs are expected to be connected through very fast I/O interconnects, such as the PCI Express or NVM Express [1]. The other form is similar to the typical DRAM-based DIMMs, except that the building block is NVM not DRAM.

Emerging NVM-based SSDs are expected to be optimized for density more than NVM-based DIMMs, which are expected to be optimized more for performance. A key difference between both forms is the write granularity; DIMMs are expected to be written in cache line granularity, which is typically 64B. In contrast, SSDs are written in block granularity, which is typically 4KB.

Understanding how using these technologies can affect performance and lifetime is very important. NVM technologies can be adopted in several ways. One way is to use the

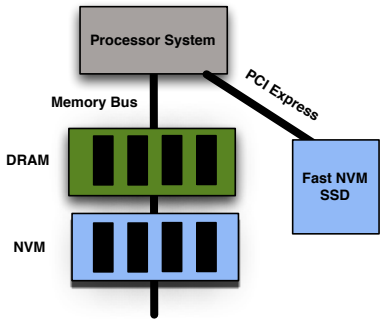


Figure 1: Emerging NVMs as SSD vs. DIMM.

emerging NVMs as a direct replacement for DRAM main memory. While this use case is attractive for low-frequency processors, such as sensor processors, it can be challenging for high-performance systems for several reasons. First, the long write latency of NVM devices can degrade the whole system performance, even for applications that do not require large memory capacities. Second, the lifetime of the system can be intolerably shortened; NVMs can wear out millions of times faster than DRAM [15, 12, 29]. Another use case would be as a direct replacement for slower technologies in I/O storage systems. However, previous studies showed that the I/O software path, such as the block layer and filesystem, can highly devalue the impact of such technologies [6]. While both of the previous use cases are expected to take place in future systems, in this paper we look into a different use case. Specifically, we look into using the emerging NVMs as **fast memory extensions**. Memory extensions can be thought of as a second-level memory, where the OS ultimately chooses which pages are held in DRAM and which are moved to the memory extension. Unlike traditional swap systems, memory extensions bypass the file system and are dedicated for extending the capacity of the main memory system. Using emerging NVMs as memory extensions is attractive for several reasons. First, it is simple to integrate efficiently in current systems; no modifications are required at the hardware-level. Second, memory extensions are managed by the OS and hence their usage is transparent to applications. Finally, unlike swap systems, no interference with file system is required. In this paper, we aim to answer the following questions. How can memory extensions affect performance? How to maximize their performance? Most importantly, how fast can they wear out, and can we improve their lifetime?

Previous work explored extending the virtual memory using NAND-based Flash SSD drives [18]. Their work, FlashVM, showed how using a dedicated SSD drives to extend memory is efficient and cheap. FlashVM isolates the SSD flash drives being used for extending memory from the SSD flash drives containing the file system. In this work, we take similar direction but with much faster technology. We aim to understand how emerging NVM technologies can impact the performance.

In this paper, we emulate using NVMs as fast memory extensions and study their impact on the performance on a real system. We discuss and evaluate several write-aware management policies, including a simple frequent value-based compression. We show the importance of having the system software and NVM controller work in concert for reducing the number of writes. Our best scheme where the system software and the NVM controller work in concert could reduce the number of writes to only 5% of the original baseline. Finally, we study the impact of OS-level prefetching and page replacement policy in reducing the number of page faults.

- Our work is the first to study using emerging NVMs as fast memory extensions through emulation on a real system.
- We investigate the impact of several write-aware system software management techniques and their relation with hardware techniques.
- We investigate OS-level page prefetching and its impact on performance.
- We evaluate the impact of page replacement policy on the number of page faults.

The rest of the paper is organized as follows. In Section 2, we discuss how memory extensions work and their impact on lifetime. We present related work in Section 3. We discuss our emulation methodology and assumptions in Section 4. In Section 5, we discuss and study our memory extension emulation. We also propose and study several performance optimizations and write reduction techniques. In Section 6, we do a parameter sensitivity study. Finally, we conclude our work in Section 7.

2. MEMORY EXTENSIONS

Memory extensions are memories dedicated for extending memory capacity. At any instance of time, a memory page can be either in the DRAM or the NVM. However, any access to an NVM page will be handled by the OS and this results in moving the page to DRAM.

The OS kernel can keep track of physical pages through a linked list of pages' metadata structures (e.g., `struct page`). When a physical page is moved to the NVM, all the virtual addresses that map to that physical page get unmapped, so that any later access to that page will raise a page fault. Note that there must be a mechanism to know the actual NVM physical address the unmapped virtual address used to map to, so the kernel can copy its content to a DRAM page and then map the virtual address to the new page. Since DRAM size is fixed, copying or moving a page from the NVM to the DRAM also requires evicting a page from the DRAM to the NVM. The actual page eviction to the NVM does not need to occur in the critical path, but can be simply buffered in a pre-allocated fixed-size eviction buffer in DRAM. The eviction buffer is periodically flushed through a kernel thread without affecting the critical path.

NVM technologies have different endurance levels. As an example, the NAND-based Flash cells can wear out after a few tens of thousands of write cycles. In contrast, PCM cells can endure few millions to hundreds of millions of writes per cell. Furthermore, PCM is orders of magnitude faster than NAND-based flash. Such factors make emerging NVMs a much stronger candidate for extending the memory.

Figure 2 shows a comparison between the expected lifetimes for a system that uses PCM memory extension versus

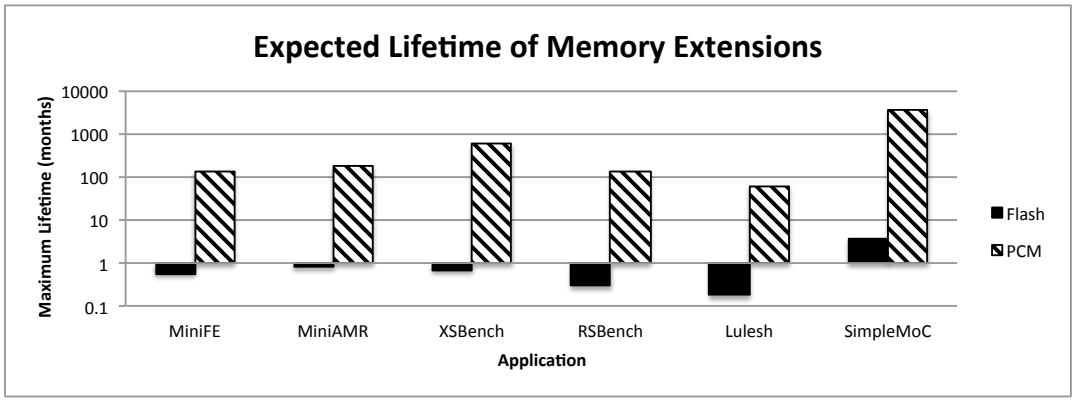


Figure 2: The expected lifetime of memory extensions (log scale).

Flash memory extension. For both memory extensions, we assume the memory extension has 4x the DRAM capacity (4GB DRAM and 16GB memory extension). We assume PCM with 150ns/1500ns page read/write latency and endurance of 10^7 writes per cell. For NAND-Flash, we use the model calibrated in [23], a 25 μs read/write latency per page. We also assume 10^4 writes per cell, which is typical for modern NAND-based Flash.

From Figure 2, we can observe that Flash-based memory extensions can wear out in less than a month for most of the HPC workloads we tried. In contrast, PCM-based memory extension can last for a few years for most of the applications, however, the lifetime can be shortened further in systems with other frequent write activities (e.g., checkpointing in HPC systems). For large-scale HPC systems, the write traffic can be multiple times higher, hence making memory extensions less reliable. Accordingly, in this paper, we focus on studying and improving both the lifetime and performance of memory extensions.

3. RELATED WORK

In this section, we summarize the related work. We categorize the related work as following:

- **Memory Extension**: using a dedicated device for solely extending the virtual memory has been proposed by previous work [18]. FlashVM [18] extended the virtual memory using NAND-based Flash SSDs. However, emerging NVMs are orders of magnitude faster than Flash drives, which makes using them as memory extensions much more attractive. Furthermore, emerging NVMs can endure many more writes than Flash [4]. Accordingly, our work shows how extending the virtual memory with emerging NVMs can affect lifetime and endurance.

- **Emerging NVMs as I/O Devices**: recent studies explored using emerging NVM devices, such as PCM, as building blocks for storage systems. In [25], the authors propose an optimized version from the state-of-the-art I/O host controller interface, NVM Express [1]. The authors propose DC Express, an optimized protocol that is more suitable for emerging NVM devices. Another work studied the performance bottlenecks and possible optimizations when using fast NVM devices over the standard NVM Express interface [6]. Our work is different in that we explore emerging NVM devices as main memory extensions.

- **Emulating NVMs for Persistent Memory Allocations**: previous work studied how an application can allocate a range of memory pages that are backed by a file in an NVM device [24]. Current Linux systems already have a system call called **mmap**, which can be used to back a specific range of virtual addresses by a file. Our work is different in that we do not require any modifications at the application-level, but all the system memory pages are managed by the OS (i.e., any page of the application heap can be located at NVM or DRAM, and that is solely managed by the OS). Furthermore, our design assumes no file system on the NVM device, hence there is no need to create files to hold application-specific pages. On the other side, our work focuses on enhancing performance and leaves out persistency, which can be handled by specific APIs or kernel drivers, such as PMEM [5]. Another recent work uses a commercial platform, PMEP, to emulate how storage-sensitive workloads can have their performance affected when replacing DRAM with emerging NVMs [31]. Our work is different in that we evaluate the use of emerging NVMs as memory extensions, mainly for enhancing capacity rather than taking advantage of persistency. Furthermore, our work focuses on HPC workloads with large memory footprints.

- **NVMs as Part of the Main Memory**: previous research studies dealing with NVMs as part of the main memory [17]. The main idea is to profile pages behavior and use such information to guide the placement of pages to be in NVM or DRAM. Unfortunately, such profiling requires hardware support and modifications. Our work aims to integrate emerging NVMs with the least amount of required modifications to current systems. Our design uses emerging NVMs as natural extension for DRAM, where the DRAM can be thought of as OS-managed page cache.

4. METHODOLOGY

In this section, we explain the emulation infrastructure we use, our assumptions and the applications we run.

4.1 Emulation Infrastructure

Our emulation infrastructure is based on the PerMA NVRAM emulator [24]. PerMA is used to emulate the performance of persistent memory allocations for customizable NVM device latency. The PerMA NVRAM emulator is implemented as a Linux device driver that allows application execution at native speeds. If any application wants to allocate persistent memory, it can simply use the mmap system call to map specific range of virtual addresses to a file. Typically, this is done by passing the starting address and the size of allocation to the mmap system call. Any access to that virtual address range causes a page fault that will be handled by the PerMA driver.

As mentioned earlier, the typical use of PerMA [24] emulator is to mmap a space from the file of the PerMA device into the process address space by using the mmap system call. However, using such technique is limited and insufficient for our study for several reasons. First, we need to modify the applications code to modify every malloc and replace it with mmap. Second, every malloc will cause an mmap system call that is expensive. Third, managing the allocated space through the malloc calls is now shifted to the PerMA device, which deals with page granularity, unlike typical malloc/free implementations. Finally, we also need to have the shared libraries allocating memory from the PerMA device, not only the malloc calls visible in the source code. To overcome the previous issue, at the linking time, we inter-position standard malloc with a simple implementation of malloc/free and new/delete calls.

Initially, the first malloc/new mmaps a very large space (e.g., 32GB) from the PerMA device into the process address space. Later, any subsequent malloc/free calls will use that space as if it is the heap of the process (i.e., they initially see a free 32GB starting from the address returned from the mmap occurring in the first malloc). All subsequent malloc calls will be looking into free space within that 32GB space, while free calls will free the allocated space from that 32GB space. In other words, the mmap'ed region is used as the heap space. Accordingly, malloc/free are now allocating regions from the space visible to the device and any page faults are handled through the PerMA device driver. We implement a simple first-fit malloc/free, with both coalescing at free and splitting allocations with large internal fragmentation.

4.2 System Configuration

We run all of our experiments on an AMD APU A10-7850K with 32GB main memory. We use Linux kernel version 3.19.6 with Ubuntu distribution.

4.3 NVM Device Model

Similar to current SSD drives, writes are expected to be hidden through several SSD optimizations, such as log-based writing and buffering. Indeed, previous papers calibrated read/write latencies of flash drives and found them to be similar [24]. For NAND-Flash, we use the model calibrated in [24], a 25 μs read/write latency per page. We study the sensitivity for Flash write latency on Section 6. For PCM, we use page read and page write latencies of 150ns and 1500ns, respectively. We assume a NAND-flash endurance of 10^4 write cycles, and PCM endurance of 10^7 write cycles. Our conservative estimations are based on projections from literature and recent announcements for emerging NVM technologies [16, 12, 4, 29, 7, 15].

Current emerging NVM technologies, such as 3D Xpoint, are expected to be deployed with 4x the capacity of DRAM in future systems [8]. Accordingly, unless explicitly mentioned, we assume DRAM:NVM ratio to be 25%.

4.4 Applications

In this study, we focus on HPC workloads that are expected to run on large-scale systems with real demand for memory capacity. Accordingly, we select several open-source proxy applications from the U.S Department of Energy. The selected applications were configured to run with large memory footprints (approximately 16GB). Table 1 presents a summary of the used applications.

5. EVALUATION AND ANALYSIS

In this section, we propose and discuss several write reduction schemes and compare their impact on number of writes and lifetime. Later, we study the impact of the DRAM to NVM ratio on performance and hence understanding how technologies with different densities can be used as memory extensions in future systems.

5.1 Write-aware Memory Management

One of the main limitations of emerging NVMs is their limited write endurance. For example, PCM cells can only endure few millions of writes. Furthermore, writing to PCM array consumes up to 43× the energy consumed by DRAM [11]. Accordingly, deploying such technologies as main memory extension requires careful consideration for how many writes they are exposed to. In this section, we study the impact of system software implementations on the number of writes. Later, we investigate the impact of using state-of-the-art hardware techniques on write reductions and enhancing the lifetime of NVM.

5.1.1 System Software

One important aspect to consider is the ***inclusion property*** of memory extensions. The inclusion property determines if a page that is allocated in DRAM is required to be allocated in the NVM as well. For example, the system software can either copy the accessed page to the DRAM and keep a copy at the NVM or simply move the page to DRAM and free up the NVM copy.

We refer to keeping a copy of the page in the NVM as ***inclusive*** memory extension. Inclusive systems are very important in the context of NVM systems; many pages get written a few times, and for the rest of the application, are only read. As depicted by Figure 3, in inclusive systems, when a page gets evicted from DRAM, the system software checks if the page has been updated since the last time it was brought from the NVM. If the page has not been updated (clean), no write to the NVM is required, as shown in Case 1. However, if the page has been updated (dirty), the page should be written back to the NVM. Checking if the page is dirty or not can be accomplished by checking the page dirty bit which is set by the Memory Management Unit (MMU) hardware. Most modern processor systems support

Application	Description
Lulesh	It represents a typical hydrocode. LULESH approximates the hydrodynamics equations discretely through partitioning the spatial problem domain to a collection of volumetric elements defined by a mesh [3].
XSBench	A mini-app representing a key computational kernel of the Monte Carlo Neutronics application OpenMC [21].
RSBench	A mini-app to represent the multipole resonance representation lookup cross section algorithm [20].
SimpleMoC	A mini-app that demonstrates the performance characteristics and viability of the Method of Characteristics (MOC) for 3D neutron transport calculations in the context of full scale simulation of light water reactor [9].
MiniFE	A proxy application for unstructured implicit finite element codes [10].
MiniAMR	A mini-app designed to support the study of adaptive mesh refinement (AMR) codes at scale [10].

Table 1: DOE proxy applications that we use.

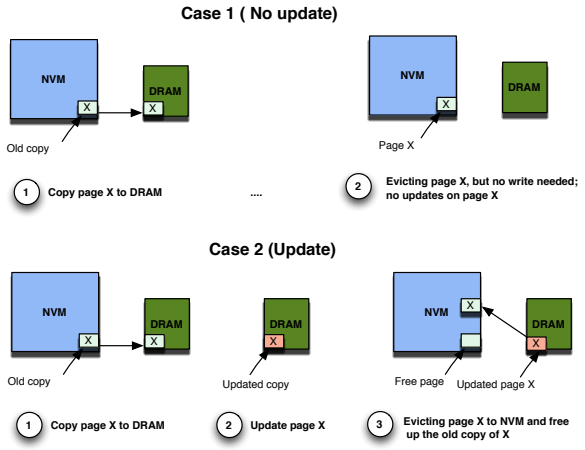


Figure 3: Handling page eviction in inclusive memory extensions.

the dirty bit in MMU. Such simple checking can save many writes to the NVM. To study the effectiveness of inclusive NVM policy, Figure 4 shows the number of cells written in inclusive memory extensions relative to non-inclusive system where old copies are simply discarded. We can see that up to 90% (MiniFE), and an average of 49.5%, of the writes could be eliminated by adopting an inclusive policy.

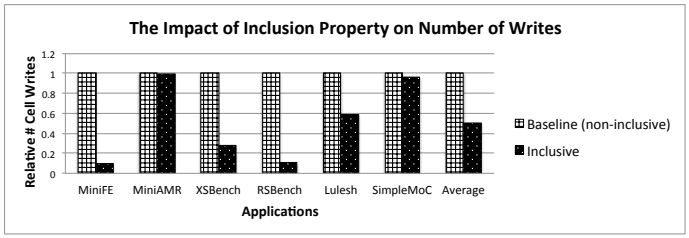


Figure 4: The impact of inclusion property on the number of writes.

The cost of deploying the inclusion policy is wasted NVM capacity. Systems with low DRAM:NVM ratio will benefit from the inclusion property at negligible ratio, however, systems with a high ratio (e.g., 50%) might prefer to use the whole capacity and disable the inclusion property. The decision of making the memory extension inclusive or not depends on applications' memory footprint and the DRAM:NVM ratio.

To reduce the number of writes further, we propose a simple compression technique based on a well-known insight that many pages have a large percentage of similar words within them. We call our technique Most Frequent Word Reduction (MFWR). As explained in Figure 5, a memory page can contain several words (e.g., 4 bytes words) that have similar content. Accordingly, at write time, the OS scans to find the most frequent word (MFW) in a page and constructs a bitmap that indicates all locations where the value of MFW appears. Later, all other words are packed together to form a compact page. The bitmap along with the MFW can be stored either in the page table metadata (e.g., struct page) or packed together with other words to be written to NVM. Since DIMM-based NVMs can be written in cache line granularity (e.g., 64B) the packed words will be written using the least number of cache lines that can include all words. Finding the most frequent word or value was studied and shown to be promising in the context of compressing cache lines in caches [27].

MFWR requires a single pass over the page data to find the most occurring word and construct its bitmap, then pack the words and the metadata. Our evaluation showed that the performance overhead is negligible (about 1.01% on average and up to 3.2% in the worst case) given that the writes occur in the background. Note that other reduction schemes, such as general compression, are possible, but they come at additional costly latency that can add to the read and write paths of the NVM device.

Figure 6 shows the impact of using MFWR technique on number of writes for the MFWR that only looks for the most frequent word (MFWR-1W) with other approaches that optimize the two most frequent words (MFWR-2W) and the three most frequent words (MFWR-3W).

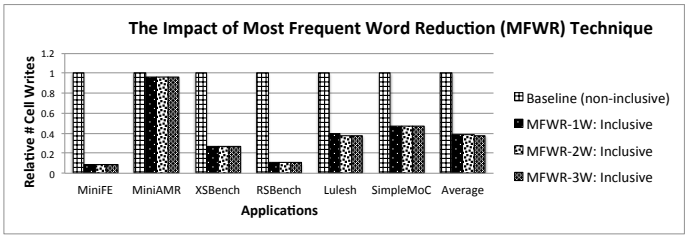


Figure 6: The impact of MFWR on the number of writes.

As shown in Figure 6, adding MFWR-1W to the inclusive policy can eliminate an average of 61.5% of writes compared to the non-inclusive memory extension policy. However, adding the number of frequent words to optimize did

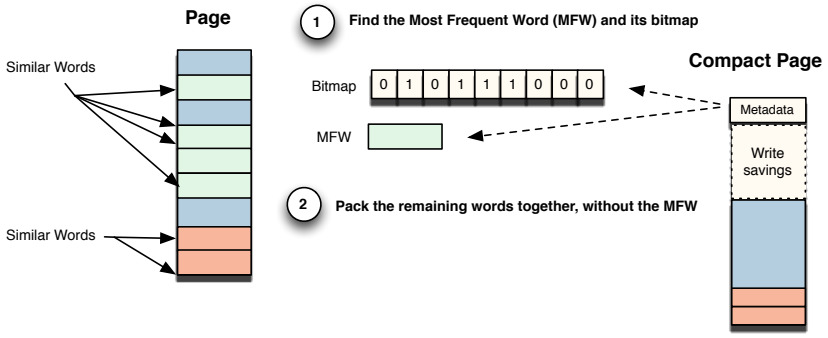


Figure 5: The Most Frequent Word Reduction (MFWR) technique.

not bring significant benefits than MFWR-1W. MFWR-2W and MFWR-3W reduce the number of writes by less than 1% relative to MFWR-1W. Thus, for the rest of the paper, we will use MFWR-1W as the default MFWR technique.

5.1.2 Hardware-aware System Software Management

While the inclusion property is a system software optimization, other techniques for write reduction has been proposed in hardware. For example, Data-Comparison Writes (DCW) technique [26] has been proven to significantly reduce the number of writes. DCW compares the old values of the NVM cells with the new data to be written, and then eliminates programming cells that have not changed in values. DCW is efficient due to the fact that the probability of having the bit written to a cell being similar to the old value is 50%, hence a promising write reduction. Unfortunately, integrating DCW is tricky due to the fact that NVMs are expected to deploy wear-leveling techniques that try to distribute the page writes uniformly across the device pages. DCW is most efficient when the pages get written in the ***same exact physical pages***. In other words, if the NVM controller deploys a wear-leveling mechanism, a page's physical location may have changed and hence the data being written on the new free page can be significantly different. A translation layer, similar to Flash Translation Layer (FTL), finds a free page depending on the wear-leveling algorithm and write the page there. Once the write is complete, the translation layer just logs the new NVM device physical address for that logical address. DIMM-attached NVM devices are also expected to deploy some wear-leveling mechanisms [15].

To tackle the problem, we propose a placement hint we denote by **PIN**. PIN hint is provided to the NVM controller when writing a page. For example, we can use one of the reserved fields in the NVM Express protocol [1] command structure to hint the NVM controller to enforce placing the logical page to its previous mapping in the translation table. Otherwise, the NVM controller will simply deploy its default wear-leveling technique. In the case of DIMM-attached NVM, the hint could be as simple as writing to a memory mapped register that is visible to the memory controller, and hence avoid applying wear-leveling techniques when writing the physical page to NVM. To emulate the impact of wear-leveling on DCW effectiveness, we assume that the wear-leveling technique will pick a physical location that has no logical relationship to the actual value of the page to be written, hence we use a randomly filled data for the physical destination.

Figure 7 shows the impact of DCW technique with and without pinning hints. From Figure 7, we can observe that DCW with pinning and without pinning can save an average of 95% and 83.8% of writes, respectively.

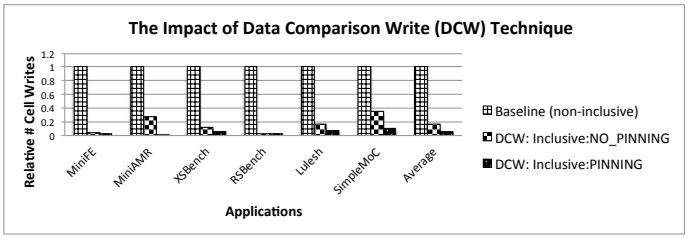


Figure 7: The impact of DCW on the number of writes.

5.1.3 Comparison Between Write Reduction Schemes

Figure 8 shows how different HW/SW approaches can affect the number of writes on the NVM device. The number of cell writes was calculated depending on the write reduction algorithm. As an example, the baseline writes 4096 bytes for every page write, while the DCW technique only writes the cells having their values changed. We observe that for some applications, such as XSBench and MiniFE, the inclusion property could eliminate about 90% of the writes. The main reason behind this is that a large percentage of pages are only being read and rarely get written after initialization. However, for some other applications, such as MiniAMR and SimpleMOC, inclusion only saves less than 10% of writes. We can observe that by adding our proposed MFWR scheme we can even reduce the number of writes further by 23.9% relative to the inclusive case. The main advantage for both of the previously mentioned techniques is that they are implemented completely in the system software side and hence no hardware support is needed. However, in the presence of hardware write reduction techniques, the system software should be aware of that and help guide the NVM controller. NVM devices with DCW hard-

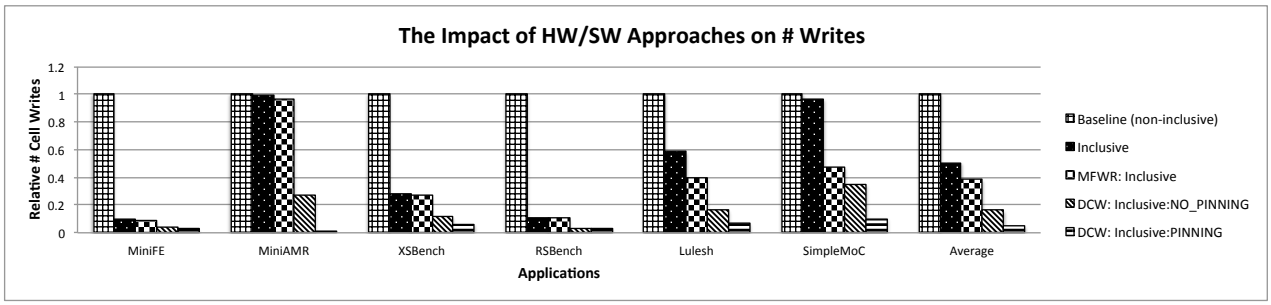


Figure 8: The impact of various write reduction schemes on the number of writes.

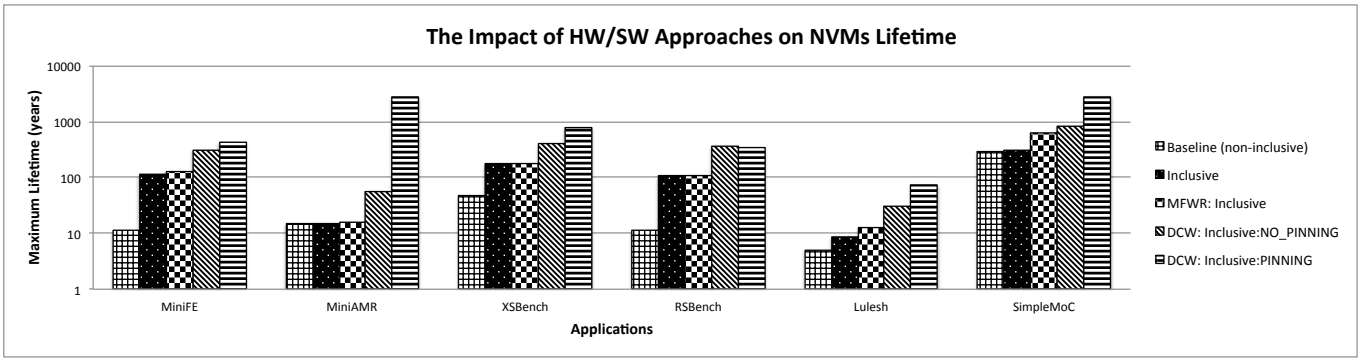


Figure 9: The impact of various write reduction schemes on the expected NVM lifetime.

ware support utilizing system software hints can reduce the number of writes by an average of 32.3% compared to those without software hints.

In summary, we presented several practical schemes that can also benefit from hardware support that could eliminate up to an average of 95% of writes. Saving such amount of writes reduces the overall energy and increases system lifetime. The additional lifetime can be used for other essential HPC-related purposes, like checkpointing and fault tolerance. Figure 9 shows a conservative estimation for the NVM lifetime depending on the application run on the system. Our calculation assumes that every single cell of the NVM device can endure ten million writes and we have a 16GB NVM device. We calculate the rate of writing cells for every application and calculate the estimated lifetime of the NVM device. We can observe that some applications, such as Lulesh, can wear out in about 5 years, but with using proper write reduction techniques it can survive up to 100 years.

5.2 The Impact of DRAM:NVM Ratio on Performance

Different NVM technologies are expected to have very high densities. For example, PCM is expected to have 4× the DRAM density [16]. However, the way of organizing the cells can also affect the density. For example, a cross-point organization can highly increase the density. Another way of improving the density further is to use multi-level cells (MLCs) where different resistance levels for the same cell can encode more than a single bit. In this section, we study the impact of varying the DRAM:NVM ratio for the memory footprint of the applications. For example, 1:4 (25%) ratio indicates that up to 20% (exclusive) or 25% (inclusive) of the application memory footprint may be present in DRAM and the rest is in the NVM. The ratio itself can be determined by the OS and restricted with the maximum capacities of both of the DRAM and the NVM. Figure 10 shows the impact of DRAM to NVM ratio on performance for the applications we study.

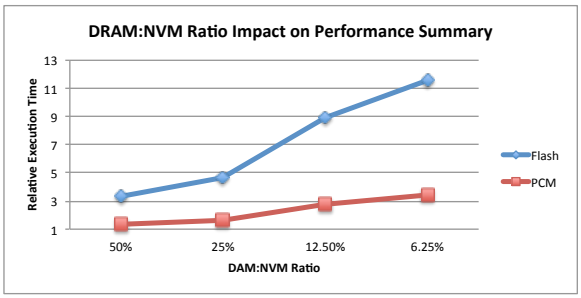
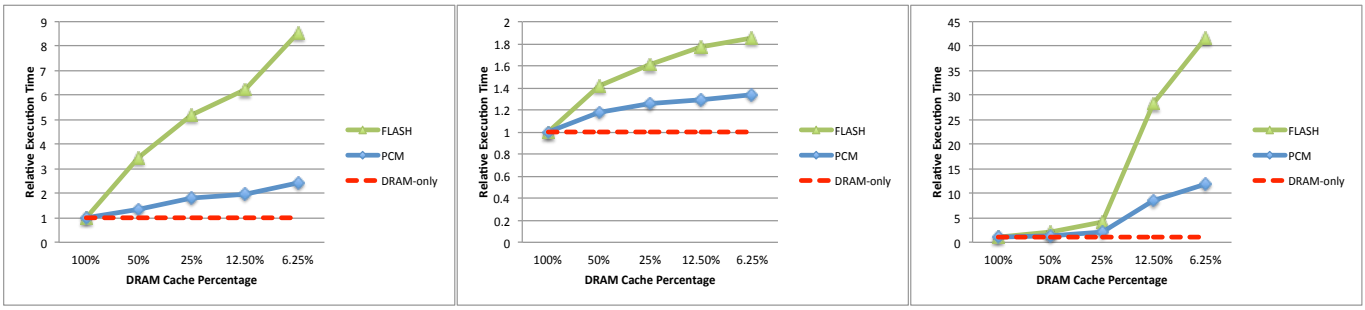
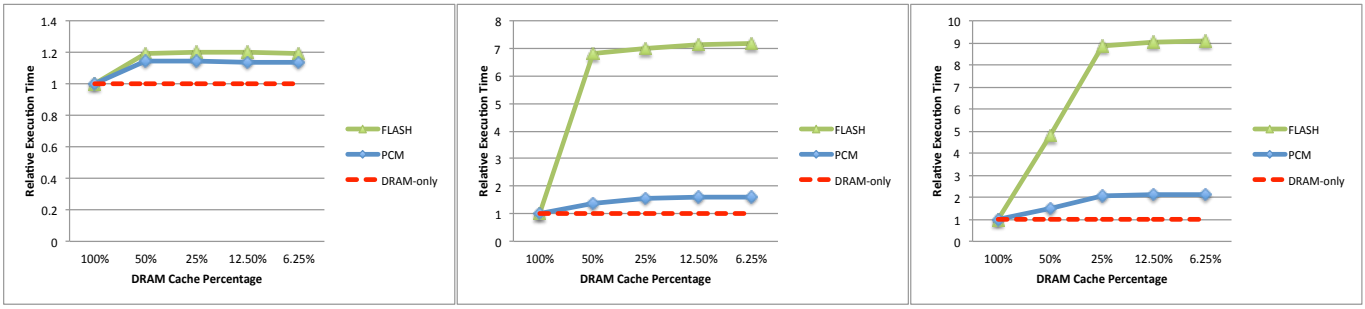


Figure 11: The impact of the DRAM:NVM ratio on performance.

We can observe that different applications are affected differently by the DRAM:NVM ratio. For instance, some applications, such as MiniAMR and SimpleMoC, show less sensitivity than other applications. The main reason behind this is that the memory pages are accessed subsequently after the first access and rarely get reused after that, hence the number of page faults changes slightly with changing



(a) Lulesh (b) XSBench (c) RSBench



(d) SimpleMoC (e) MiniAMR (f) MiniFE

Figure 10: The impact of DRAM:NVM ratio on performance for a system using PCM vs. Flash.

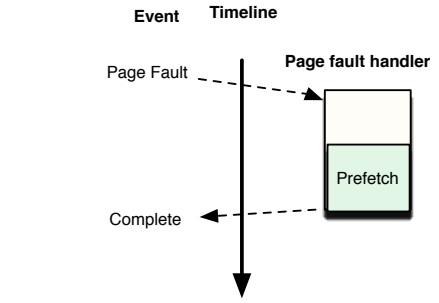
the DRAM:NVM ratio. In case of Lulesh, MiniFE and RSBench, the actual working set is large enough to fit into DRAM and hence changing the DRAM:NVM ratio can affect performance.

The exact DRAM:NVM ratio where the application performance gets degraded differs across applications. For example, MiniFE performance degrades significantly when DRAM:NVM ratio is less than or equal to 25%, however, for RSBench the DRAM:NVM ratio of 12.5% is the threshold point. Note that determining the DRAM:NVM ratio where the application's performance starts to get degraded significantly relies on the actual working set (not only memory footprint) of the application.

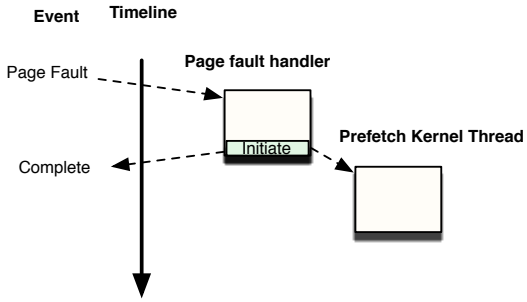
In summary, as shown in Figure 11, at DRAM:NVM ratio of 1:4 (25%), which is expected to be the actual DRAM:NVM ratio in future systems [8], PCM is expected to deliver an average performance with only 63.8% performance degradation compared to the ideal DRAM-only design. However, Flash is expected to deliver an average of 368% performance degradation to the ideal DRAM-only design.

5.3 OS-level Page Prefetching

Prefetching can be used to reduce the impact of OS page cache misses by speculating which pages will be used in the future, and bring them ahead of time. However, many parameters should be considered when using software prefetching (e.g., what drives the prefetching, where in the code to execute the prefetching). The most recent approaches for prefetching suggest that software prefetching should be added to the page fault handler [18, 14]. In our implementation, we exploit the workqueue threads feature in modern Linux kernels. Workqueues enable asynchronous work sub-

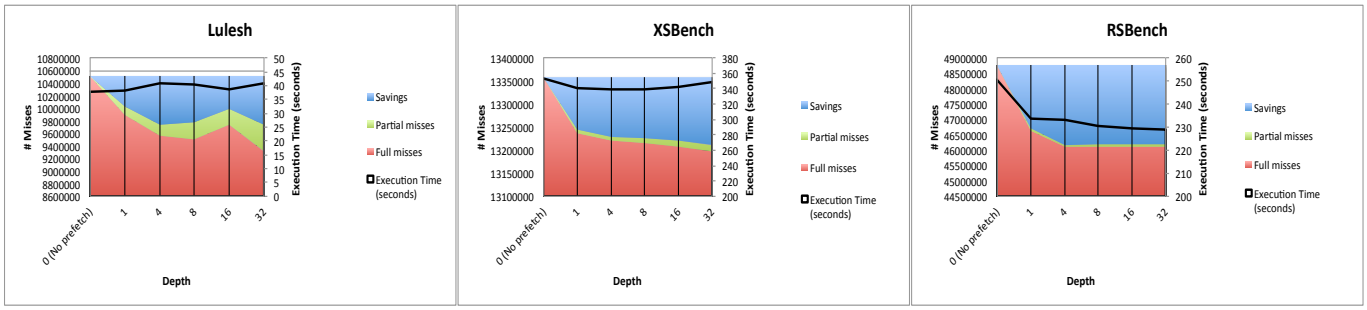


(a) Synchronous prefetching

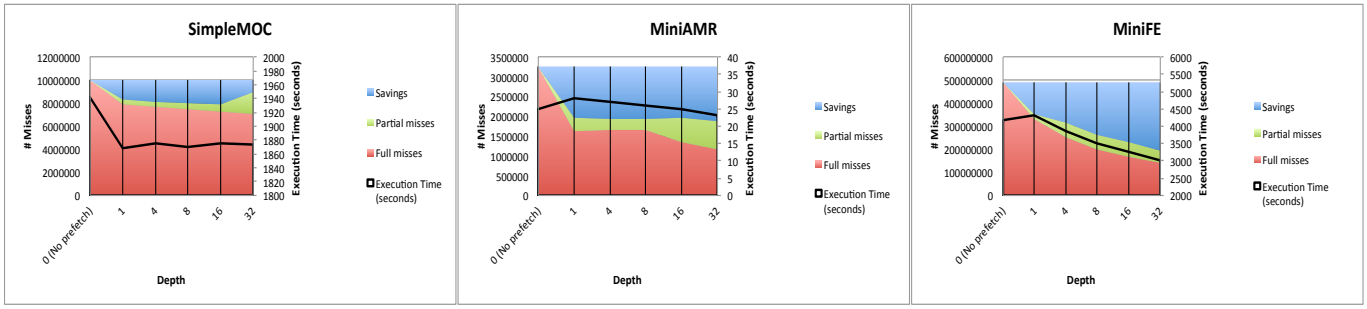


(b) Asynchronous prefetching

Figure 12: Synchronous vs. asynchronous prefetching.



(a) Lulesh (b) XSBench (c) RSBench



(d) SimpleMoC (e) MiniAMR (f) MiniFE

Figure 13: The impact of OS-level prefetching.

mission, where a function can simply submit a work item that can be shortly handled by a kernel thread. Figure 12 shows the difference between synchronous and asynchronous prefetching. In the case of synchronous prefetching, the kernel has to wait to prefetch the pages before completing the handling of the page fault. In contrast, asynchronous prefetching will just add a work item to the workqueue, which will be serviced by a kernel thread whenever possible.

Synchronous prefetching can add significant overhead to the page fault latency, and hence limiting the number of pages that can be prefetched without affecting the average page-fault handler latency; the page fault handler will be delayed until the prefetching is complete. For example, in [18], the authors found that prefetching more than two pages can increase the average page fault latency. Accordingly, we use asynchronous software prefetching. When a page fault occurs, the page fault handler decides whether to issue a prefetch or not, depending on if there is a currently running prefetch thread. If the decision is made to execute prefetching, a separate kernel task/workqueue will be issued. The prefetch thread will be scheduled by the kernel scheduler as any other tasks with low-priority.

As observed by Oskin and Loh [14], at page-level granularity, simple stride prefetcher can be more efficient than much more complicated prefetchers, such as Markov prefetcher. Accordingly, to evaluate the effectiveness of prefetching, we use a simple stride prefetcher that records the most recent 8 strides. The prefetcher issues prefetching requests for strides that appeared at least 5 times since it was initially recorded, hence avoid prefetching wrongly detected streams. The number of prefetches for each stride is determined by the depth of the prefetcher. Figure 13 shows the impact of prefetching on performance while varying the depth of the prefetcher. For each application, the primary axis shows the total number of page faults, while the secondary axis (to the right) shows the execution time. The page faults (i.e., DRAM misses) are categorized into: full and partial misses. Partial misses are those occurred while a prefetch for that page has already started, while full misses are for those not in DRAM and have no prefetching in progress. The top area on each figure represents the actual savings in number of misses compared to no prefetching.

We can observe that prefetching can reduce the number of page faults by different amounts for different applications. For example, MiniFE and MiniAMR have significant percentage of their faults eliminated due to prefetching. However, other applications slightly benefit from prefetching. We also observe that for applications that do not get significant reduction in number of faults, the performance can be penalized. The main reason behind this is the processor time spent by the kernel thread for prefetching. Furthermore, submitting a work item to the kernel worker is not free; our measures showed that about 3-4 microseconds could be added to the page fault handler just for submitting the work. For example, we can see that most of the applications get their performance degraded at depth of 1, because for almost every page fault, the page fault handler will find that there is no prefetching in progress and hence issue a prefetching request. This adds the work submission overhead to the page fault handler almost for each miss. Interestingly, some applications, such as MiniFE, could gain up to 37.2% performance improvement. While beyond the scope of this paper, the OS can adaptively enable/disable prefetching by learning how efficient prefetching is for different workloads.

5.4 Page Replacement Policy

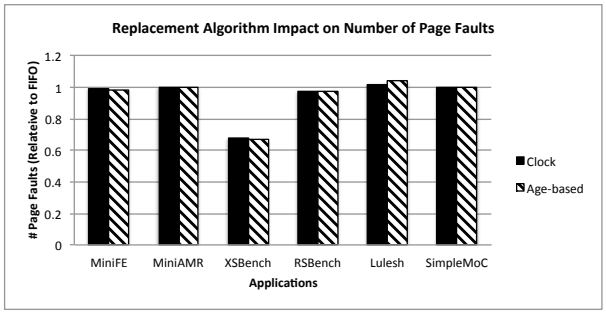


Figure 14: The impact of page replacement policy.

As we have a fixed number of pages that can fit in the DRAM, we need to have a policy to choose which pages to evict when bringing pages from NVM. For this purpose, we study the effectiveness of the clock algorithm [19] and compare it against the simple FIFO. The clock algorithm works as follows: a pointer points to the page right after the most recently inserted one. Each time we want to evict a page, we start from the page pointed by the pointer to check the reference bit. The reference bit is set by the hardware at any access to the page. The reference bit can be checked by the operating system through the page table entry for the page. If the reference bit is set to 1, we clear it and move the pointer to the next page and check it. The process continues until finding a page with the reference bit cleared, that page will be chosen for eviction and the pointer will move to the next page. Finally, we also implement an age-based algorithm that is different than the clock algorithm in that it has a counter per page. The counter is incremented if the reference bit is found to be set, and decremented otherwise. If a page has its counter with value less than or equal to zero, it will be selected for eviction.

Figure 14 shows the relative number of page faults for both clock and age-based replacement algorithms compared to FIFO. We can observe that most of the benchmarks don't benefit and even some of them, such as Lulesh, incur more page fauls when changing the replacement policy. Only XSBench benefits from the replacement policy change. We also vary the DRAM:NVM ratio and did not observe any change except for XSBench, where clock and age-based algorithms could save roughly 50% of the page faults. The reason why XSBench benefits from the clock algorithm is that it uses some indirection table that holds pointers to small arrays. The accesses to the whole structure are random, but keeping the pointers structure, which is frequently accessed is beneficial. The clock and age-based algorithms can detect such pages that are frequently accessed and try to keep them in DRAM as much as possible. For all other applications, our conclusion is consistent with previous research findings [22] in that at the page-level granularity, a simple algorithm such as FIFO would be more efficient than other more complex schemes such as clock algorithm.

6. SENSITIVITY STUDY

Flash devices can have different write latencies, depending on the technology and number of levels in cells, i.e., Single-Level Cell (SLC) or Multi-Level Cell (MLC). For instance, the authors in [30] found that the effective write latency can reach up to several milliseconds. Accordingly, we perform a sensitivity study to estimate the impact of various write latencies of flash memory on the execution time of the memory extension, compared to that of the PCM-based memory extension. Figure 15 presents a comparison for a flash latency with 25μs read latency, while varying the write latency from 25μs, 100μs, 200μs, and up to 400μs.

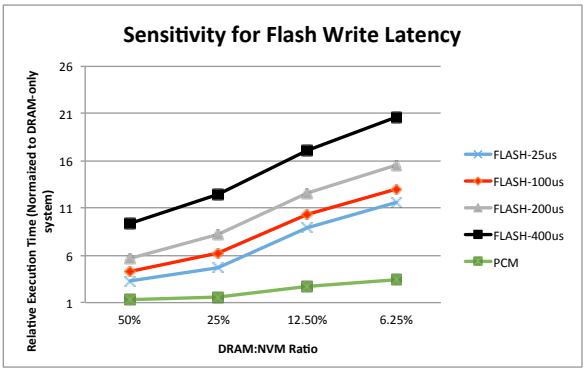


Figure 15: The impact of NVM write latency on performance, for various DRAM:NVM ratios.

From the figure, we can observe that longer flash write latency can affect the performance of Flash-based memory extensions significantly; the gap between PCM-based and Flash-based memory extensions becomes even greater.

7. CONCLUSION

Our work studied the effectiveness of using emerging NVMs as memory extensions. We showed how using emerging NVMs as memory extensions without write-aware management techniques can have serious impact on systems' lifetime. We discussed and proposed several management schemes and evaluated their impact on reducing the number of writes. We also explored the impact of varying the DRAM:NVM ratio of memory extensions on performance, for both Flash and PCM. To enhance performance, we investigated and proposed OS-level prefetching, which showed promising results for some applications. Furthermore, we studied the impact of page replacement policy on number of page faults (NVM accesses). Our study showed that only one out of six applications benefits noticeably from changing replacement policy to temporal reuse aware policies, such as the clock and age-based algorithms.

We believe that our work gives insights about the impact of using emerging NVMs as memory extension and helps in understanding how to manage such memory extensions to make them more reliable.

8. ACKNOWLEDGMENT

We would like to thank the anonymous reviewers, Myoungsoo Jung and Mark Oskin for their generous feedback and insights to improve the paper.

9. REFERENCES

[1] A. Huan. NVM Express, Revision 1.0c. Intel Corporation, 2012.

[2] Huai, Yiming, et al. Observation of spin-transfer switching in deep submicron-sized and low-resistance magnetic tunnel junctions. Applied Physics Letters 84.16: 3118-3120, 2004.

[3] Hydrodynamics Challenge Problem, Lawrence Livermore National Laboratory. Technical Report LLNL-TR-490254.

[4] Intel 3D XPoint.

[5] Persistent Memory Programming. http://pmem.io, 2014.

[6] A. Awad, B. Kettering, and Y. Solihin. Non-Volatile Memory Host Controller Interface Performance Analysis in High-performance I/O Systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2015*, pages 145–154, March 2015.

[7] S. Chhabra and Y. Solihin. i-NVMM: A Secure Non-volatile Main Memory System with Incremental Encryption. In *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ISCA '11, pages 177–188, 2011.

[8] R. Crooke and A. Fazio. Intel Non-Volatile Memory Inside. The Speed of Possibility Outside. In *Intel Developer Forum (IDF), 2015*.

[9] G. Gunow, J. R. Tramm, B. Forget, and K. Smith. Simplemoc - a performance abstraction for 3d moc. In *ANS MC2015*. American Nuclear Society, 2015.

[10] M. Heroux, D. Doerfler, P. Crozier, J. Willenbring, C. Edwards, A. Williams, M. Rajan, E. Keiter, H. K. Thornquist, and R. W. Numrich. Improving performance via mini-applications. In *Sandia Report 2009*.

[11] B. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable dram alternative. In *International Symposium on Computer Architecture (ISCA)*, 2009.

[12] Z. Li, R. Zhou, and T. Li. Exploring high-performance and energy proportional interface for phase change memory systems. *IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*, pages 210–221, 2013.

[13] P. J. Nair, D.-H. Kim, and M. K. Qureshi. ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 72–83, 2013.

[14] M. Oskin and G. H. Loh. A Software-managed Approach to Die-stacked DRAM. In *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2015.

[15] M. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali. Enhancing lifetime and security of PCM-based Main Memory with Start-Gap Wear Leveling. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 14–23, 2009.

[16] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. *ACM SIGARCH Computer Architecture News*, 37(3):24–33, 2009.

[17] L. E. Ramos, E. Gorbatov, and R. Bianchini. Page Placement in Hybrid Memory Systems. In *Proceedings of the International Conference on Supercomputing (ICS)*, pages 85–95. ACM, 2011.

[18] M. Saxena and M. M. Swift. FlashVM: Virtual Memory Management on Flash. In *Proceedings of the 2010 USENIX conference on USENIX Annual Technical Conference (USENIX-ATC)*, pages 14–14. USENIX Association, 2010.

[19] A. S. Tanenbaum. Modern Operating Systems. 2009.

[20] J. R. Tramm, A. R. Siegel, B. Forget, and C. Josey. Performance analysis of a reduced data movement algorithm for neutron cross section data in monte carlo simulations. In *Solving Software Challenges for Exascale*, pages 39–56. Springer, 2014.

[21] J. R. Tramm, A. R. Siegel, T. Islam, and M. Schulz. XSBench - The Development and Verification of a Performance Abstraction for Monte Carlo Reactor Analysis. In *PHYSOR 2014 - The Role of Reactor Physics toward a Sustainable Future*, Kyoto.

[22] A. J. Uppal and M. R. Meswani. Towards Workload-Aware Page Cache Replacement Policies for Hybrid Memories. In *Proceedings of the 2015 International Symposium on Memory Systems (MEMSYS)*, pages 206–219, 2015.

[23] B. Van Essen, H. Hsieh, S. Ames, R. Pearce, and M. Gokhale. DI-MMAP: a scalable memory-map runtime for out-of-core data-intensive applications. *Cluster Computing*, 18(1):15–28, 2013.

[24] B. Van Essen, R. Pearce, S. Ames, and M. Gokhale. On the role of NVRAM in data-intensive architectures: an evaluation. In *Proceedings of IEEE 26th International Parallel & Distributed Processing Symposium (IPDPS)*, pages 703–714, 2012.

[25] D. Vučinić, Q. Wang, C. Guyot, R. Mateescu, F. Blagojević, L. Franca-Neto, D. L. Moal, T. Bunker, J. Xu, S. Swanson, and Z. Bandić. DC Express: Shortest Latency Protocol for Reading Phase Change Memory over PCI Express. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (USENIX-FAST)*, pages 309–315, 2014.

[26] B.-D. Yang, J.-E. Lee, J.-S. Kim, J. Cho, S.-Y. Lee, and B. gon Yu. A low power phase-change random access memory using a data-comparison write scheme. In *Proceedings of the IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3014–3017, 2007.

[27] J. Yang, Y. Zhang, and R. Gupta. Frequent Value Compression in Data Caches. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture (MICRO)*, pages 258–265, 2000.

[28] J. J. Yang, D. B. Strukov, and D. R. Stewart. Memristive Devices for Computing. *Nature Nanotechnology*, 8(1):13–24, 2013.

[29] V. Young, P. J. Nair, and M. K. Qureshi. DEUCE: Write-Efficient Encryption for Non-Volatile Memories. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 33–44, 2015.

[30] J. Zhang, G. Park, M. M. Shihab, D. Donofrio, J. Shalf, and M. Jung. OpenNVM: An open-sourced FPGA-based NVM controller for low level memory characterization. In *Proceedings of the 33rd IEEE International Conference on Computer Design (ICCD)*, pages 666–673, 2015.

[31] Y. Zhang and S. Swanson. A study of application performance with non-volatile main memory. In *Proceedings of the 31st Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2015.

[32] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*, pages 14–23, 2009.