Towards Transparent and Seamless Storage-As-You-Go with Persistent Memory*

Hyeonho Song
UNIST
hustard@unist.ac.kr

Sam H. Noh
UNIST
samhnoh@unist.ac.kr

1 Introduction

In traditional computer systems, memory and storage are statically divide and separately allocated. In such a strictly dichotomized system, resource usage becomes unbalanced; more memory is always in need, yet large portions of storage remains unused [5, 18]. The goal of this paper is to break this strict division of memory and storage, and present a system that can dynamically move the boundary between memory and storage as need be.

Commercial Persistent Memory (PM) is now in the horizon. In particular, 3D XPoint based SSDs are now in the market [12]. More importantly, PM products based on the DIMM interface are expected to be available soon as well [11]. Such products are expected to bring many changes to ways systems behave [7, 9, 21].

In this paper, we assume a hybrid memory system composed of DRAM and DIMM interface PM, where the intention is to use PM as storage space [7, 9, 21]. For such a system, we present Storage-As-You-Go (SAY-Go), a system that transparently adjusts the use of PM such that PM can be used as memory as well as storage as need be. In particular, it has been observed that storage is almost never used to its full capacity [5, 18]. Our approach is to provide PM as a middle ground where PM that is not yet used as storage may be allocated as memory if the application requires more memory to alleviate the memory crunch.

There are two technical challenges in achieving this goal. One is providing a memory allocation service that can freely grow and shrink memory it is managing. The other is a file system that supports dynamic resizing of partitions. In this paper, we present our solution of the former, which we refer to as Persistent Memory Buddy (PMB). A design of PMB is presented along with its implementation in the Linux kernel. The latter is left for future work.

*This work was supported by Samsung Research Funding Center of Samsung Electronics under Project Number SRFC-IT1402-52.

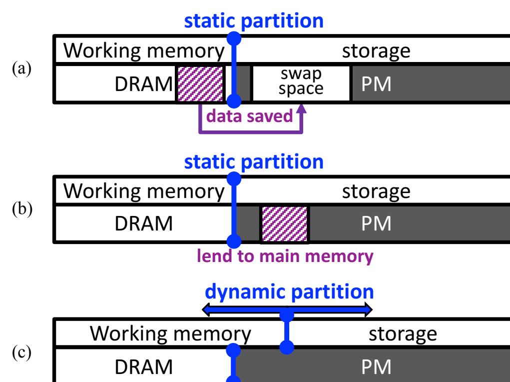


Figure 1: Working memory expansion methods: (a) Swapping (b) pVM and Memorage (c) SAY-Go

2 Related Work

Since the inception of modern computing, the hunger for more memory has yet to be satiated. From the early days of computing, and even to this day, techniques such as virtual memory and swapping have been used to provide the illusion of sufficient memory as shown in Figure 1(a). These techniques have come with sacrifice in performance as data needs to be transferred between fast memory and slow storage that are separated by a hard, static boundary. Such techniques were developed under the premise that DRAM capacity is (relatively) small, while storage capacity is large.

With the advent of PM, researchers have considered other options. As PM is compatible with DRAM in terms of performance, it was observed that PM itself could (temporarily) be used as memory [13, 14]. (We limit our discussions here to when PM is being used as storage. There is a whole body of work that attempts to make use of PM just as main memory, but again, a resource that is statically and separately divided from storage. See [16, 17, 22] and references within for more related work.) The basic idea behind these approaches is depicted in Figure 1(b) where part of PM is borrowed out of storage space and used as part of memory.

Specifically, pVM [14] is a recent study that focuses on providing persistent store similar to NV-Heap [6] and Mnemosyne [20], but making use of the memory system to enhance performance. In so doing, it also provides a feature to allocate part of the PM as non-persistent space to the process address space. Since providing persistent store is the primary purpose, it also provides a technique for ensuring consistency. In terms of transparency, users need to explicitly make calls to a library with calls such as `npmalloc()`, `nvmmap()` to allocate space from PM. This requires application modification and programmer awareness to make use of these features.

The system most similar to our approach is Memorage proposed by Jung and Cho [13]. Memorage approaches the problem from a file system perspective. Operating on a PM-based file system, file system free blocks are taken and structured as a buddy system and set as the Memorage zone to provide PM space as memory. The file system has to be modified to reflect the fact that the blocks lent as memory have been allocated so that conflicts do not occur. The benefit of Memorage is that it does not require any application modifications and working memory expansion can be transparently provided. However, it requires modifications to the file system to support this feature. Additionally, runtime overhead is incurred as data structures for memory expansion are created and released dynamically. Finally, the original Memorage study overlooks the issue of data structure consistency, a critical issue when making use of PM.

A summary of the differences between the two previous closely related studies, pVM and Memorage, and SAY-Go, the system we propose, are given in Table 1. The key distinctive feature of SAY-Go is the transparent dynamic adjustment of the memory and storage boundary as shown in Figure 1(c). To the best of our knowledge, our work is the first to propose and provide a mechanism for supporting such a dynamic boundary.

Table 1: Summary and comparison to previous work

	pVM	**Memorage**	**SAY-Go**
Goal	persistent store	memory expansion	efficient use of resources
Memory Storage Division	fixed	fixed	dynamic
Consistency	yes	not considered	yes
Transparent (Automatic scaling)	no	yes	yes
Runtime overhead	no	yes	no

3 PM Buddy Design Goals

In the Storage-As-You-Go (SAY-Go) system, instead of a static division of PM into working memory and storage, the PM capacity used as working memory or storage is dynamically adjusted as need be. For such dynamic adjustment, appropriate memory management and file system support is required. In this section, we describe the design goals of PM Buddy (PMB), the memory management component to support SAY-Go.

Our design goal with PMB is as follows.

1. **Seamless Integration:** PMB should take action only when memory runs out. As we assume systems with DRAM and PM hybrid memory, DRAM will, in general, be used as working memory. It is only when DRAM runs out that PMB should start to take action and PM is allocated to applications. This is because PM is anticipated to have performance characteristics lower than that of DRAM. Through this flexible and dynamic allocation of PM free space, the system can dynamically change the logical boundaries of working memory and storage despite the physically fixed boundaries of DRAM and PM, as can be seen from Figure 1(c).
2. **Transparency**: PMB must be supported in a way such that applications are not aware of its happenings, but should only reap the benefits of employing PMB. That is, legacy applications should not change in any way. Furthermore, in developing new applications, legacy programming models should be sufficient and need not make use of new programming models.
3. **Consistency:** Finally, the system needs to remain consistent upon a normal reboot as well as a reboot after recovery from fault. Consistency here refers to the fact that the persistent part of PM should be viewed as being persistent and the non-persistent part being cleared of content for reuse. Such consistency, though seemingly trivial and natural, cannot be maintained without properly ensuring the permanence of state and their corresponding metadata upon a fault. For example, if not accurately designed, memory leaks may occur if a user program terminates due to a fault in the memory allocation step. Such memory leaks can accumulate as PM is, by nature, persistent, leading to performance degradation in the long run.

To achieve goals 1 and 2, we design and implement PMB within the memory management framework of the operating system kernel. In particular, as we describe in Section 4, within Linux, the Buddy system is adopted and extended. To achieve the third goal, we maintain data structures such as page descriptors and an allocation bit map in PM separately from the existing Buddy system and make use of logging.

We describe the detailed implementation of PMB in the following section.

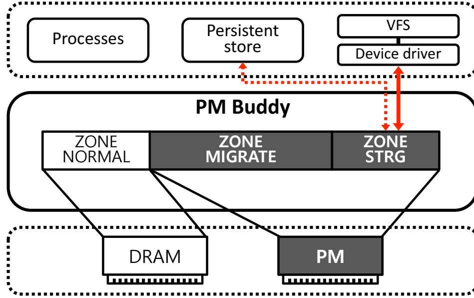


Figure 2: PM Buddy layout in Linux

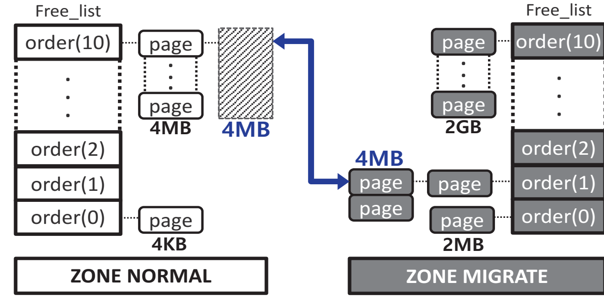


Figure 3: Page allocation from MIGRATE zone to NORMAL zone and vice versa

4 PM Buddy Implementation

In this section, we describe the working prototype of PMB as implemented in Linux. In order to satisfy the transparency design goal, we make minimal changes to the existing system. This is especially important with the memory management module as this module is complex and sensitive to changes. Our approach is to extend the functionality of the existing buddy memory manager. Thus, as PM is extended to be used as working memory, the extra memory space is simply added on to the existing Buddy system with no changes to other layers.

4.1 Layout

Memory in Linux is divided into zones, in particular, ZONE_DMA, ZONE_NORMAL, and ZONE_HIGHMEM [10]. We concentrate our discussion on ZONE_NORMAL as this is the zone from which pages are allocated as working memory. Hereafter, we do not consider ZONE_DMA and ZONE_HIGHMEM as they are irrelevant to our discussion.

With PMB, the volatile DRAM is covered by ZONE_NORMAL (denoted NORMAL, henceforth), while PM is divided into two new zones, ZONE_MIGRATE and ZONE_STRG (each denoted by MIGRATE and STRG, respectively, henceforth) as shown in Figure 2. STRG is the minimum storage area that will always be used as storage space (solid line in Figure 2). It is also used to persist metadata that is needed to satisfy the consistency goal of the design, that is, maintain consistency of PM. In particular, STRG stores the PM page descriptor, which contains the current order information that is essential in maintaining the buddy system. It also stores the bitmap that maintains the allocation state of PM pages and a log table used to recover from faults that occur during allocation.

Though we do not consider the use of recent programming models such as Mnemosyne [20], NV-Heaps [6], or pVM [14] in this paper, such models can be supported by making use the STRG zone as depicted by the dotted arrow in Figure 2. MIGRATE, on the other hand, is the PM area that is used as memory or storage as flexibly as need be. This area is the core of PMB that is used to satisfy the seamless integration design goal.

4.2 Managing pages

The MIGRATE zone is the key component in PMB. Pages in MIGRATE are either allocated as persistent or non-persistent pages depending on its use as storage or memory. As shown in Figure 3, the management of pages in MIGRATE (right hand side) is almost exactly the same as Buddy management in the NORMAL zone (left hand side). The main difference here is the unit of each page. Whereas in Buddy, each page is 4KB, with PMB, the page size is 2MB, the size used for huge pages in Linux. PMB takes the 2MB huge page size as the default page size. This choice is made as the existing Buddy supports a maximum of 4MB contiguous memory space, which is relatively small for the storage layer. With the 2MB default page in PMB, zone MIGRATE and STRG can be allocated contiguous space ranging from 2MB to 2GB, and this allows for the storage layer to handle 2GB of I/O at once. Note 2MB is chosen as this size is the huge page unit supported in our platform architecture and may be changed according to the architectural support.

Page movement occurs in two directions: from MIGRATE to NORMAL, which we refer to as 'migration', and vice versa, which we refer to as 'retrieval'. Migration and retrieval are both done in 4MB page chunks, between order 10 of the NORMAL zone and order 1 of the MIGRATE zone, as depicted in Figure 3. Note that 4MB is the largest contiguous page unit supported by the Buddy system. This allows for PMB to be seamlessly integrated into the existing memory module.

Actual migration of pages from MIGRATE to NORMAL is instigated by a free space watermark. If the number of free pages falls below this watermark, the `migrator` thread calls the page migration function, which transfers the 4MB of free space.

For retrieval, in our current implementation, there is no watermark and the pages are retrieved, that is, returned from NORMAL to MIGRATE, when all pages belonging to the same order 10 page chunk are released. As we maintain the zone of origin in the page descriptor, this is checked to see where the chunk is from. If it is from MIGRATE, then it is returned to MIGRATE. If the

Table 2: Workload characteristics [3, 4, 8]

	FFT	Redis
Scale	Memory intensive application	In-memory database
Domain	Signal processing	Key-value store
Benchmark suite	Splash2x in Parsec 3.0	YCSB
Input	Native (largest)	1:1 (read:write)
Memory usage	12GB	20GB

Table 3: Evaluation Platform

CPU	Intel Xeon E5-2620v3
Memory	16GB PC4-17000 × 16 (Total 256GB)
OS	Ubuntu 14.04 with Linux v4.11.1

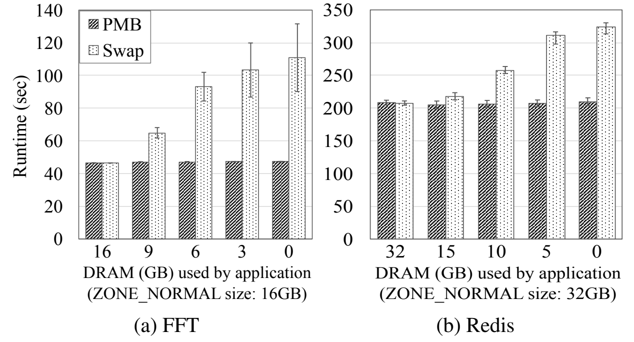
chunk is originally from NORMAL, naturally, it remains in NORMAL. The actual code change required for migration (and retrieval) is simply a series of list link and unlink operations.

5 Evaluation

In this section, we discuss the experimental results of PMB. The goals of the experiments are twofold. The first goal is to show that PMB is beneficial in performance compared to the traditional swapping method that is typically used to provide a larger virtual address space in traditional systems. The second is to show that our implementation correctly allocates and retrieves the pages between MIGRATE and NORMAL. These together, in effect, is used to show the practical usefulness of PMB.

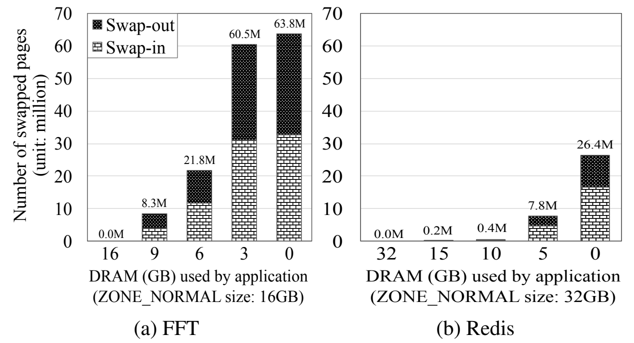
We make use of two workloads, namely, FFT and the Redis in-memory database, both of which are memory intensive applications. The detailed characteristics of each workload are shown in Table 2. Note that memory usage is 12GB and 20GB, respectively. The experimental environment is summarized in Table 3. As real PM is unavailable, we simply make use of DRAM to emulate PM. NORMAL is simply considered to be DRAM as it should be, while MIGRATE and STRG are considered to be PM and set as pseudo-PM by using the modified PMEM device driver [19].

Of the total 256GB of DRAM, 16GB is set aside for STRG. Currently, STRG has no real function except to hold the metadata for consistency. The rest of the 240GB are partitioned between NORMAL and MIGRATE, of which the size is differently set for each workload. For FFT, it is set to 16GB and 224GB, and for Redis, it is set to 32GB and 208GB. This is to accommodate the entire memory needs of each application in DRAM, which serves as the base case. In order to strain the memory usage of each application so that migration will occur, we employ a stress tool that will take up a specified amount of memory capacity [2]. Using this tool limits the memory usage of the applications that we consider requiring



(a) FFT (b) Redis

Figure 4: Performance of applications as the DRAM used by the application is limited to the values in the *x*-axis.



(a) FFT (b) Redis

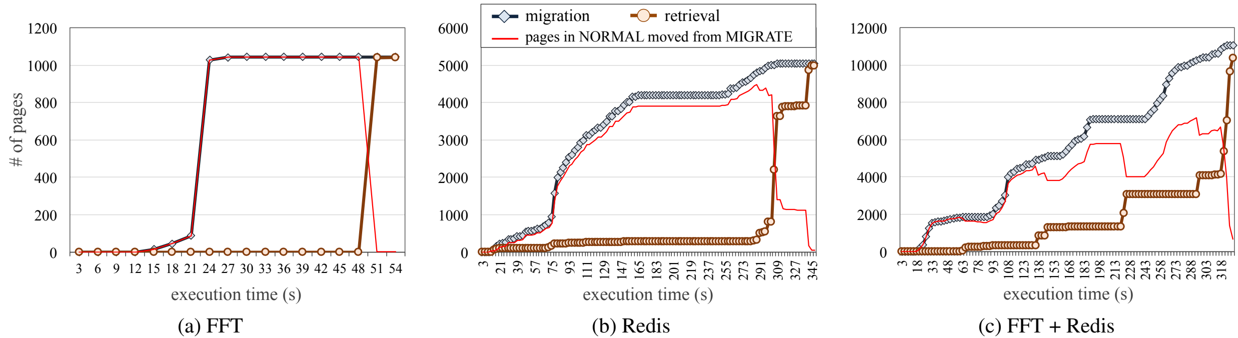
Figure 5: Total number of swap-ins and swap-outs, each of which incurs a memory copy, observed during execution of applications.

it to either swap out (as in the traditional system) or to invoke PMB to expand its memory usage to PM.

5.1 Comparison with swap

The traditional mechanism to support virtual address space larger than the physical address space is to temporarily place part of the virtual address space in storage space, for example, by means of swapping [15]. Naturally, if memory space could be extended, there would be less need to make use of storage space resulting in performance improvements. Quantitative observations of such benefits have been made in a previous study [14]. In this section, we present our own experiments and observations in comparison to the swap mechanism based on our implementation of PMB.

Figure 4 shows the average execution time of 5 executions for each setting with its standard deviation shown with the line range on top of the bars. The results show that as less and less memory is available for the workload considered (which is controlled with the stress tool mentioned above), performance with PMB remains stable. With real PM, we anticipate performance to degrade somewhat as read/write latency is expected to be slightly worse than DRAM. In the interest of space, we did not consider this issue in this paper. In contrast, we see that



(a) FFT (b) Redis (c) FFT + Redis

Figure 6: Page migration and retrieval count with PMB

swapping performance degrades considerably as memory allocated become limited. We also note the standard variation for swapping is much larger than that of PMB showing that PMB performance is relatively stable.

Specifically, our measurements show that page migration with PMB is done in 400 nanoseconds on average, while for each swap-in and swap-out, it takes 6 and 41 microseconds, respectively, as measured when the function `do_swap_page()` and `shrink_zone()`, respectively, is called and returned. While the actual copying of the page using the `movntq` instruction for each swap takes roughly 1.2 microseconds, the rest is due to software overhead. In particular, the high overhead of swap-out is due to a couple of reasons. First, within the `shrink_zone()` function, the system tries to avoid swap-out by trying to make use of the unused pages that the slab allocator had pre-allocated and by flushing the page cache to free up memory. Second, if, and only when, these attempts fail, it scans the pages in the zones using a complicated selection algorithm to select the victim page. In the case of swap-in, the page fault handler simply checks whether the requested page is in the swap space, and if so, brings it back to working memory, which incurs much less overhead compared to swap-out.

Figure 5 shows the actual number of pages swapped in and out for the two applications as reported by the `/proc/vmstat` utility provided in Linux [1]. We see that the number of swaps has a strong influence on the performance shown in Figure 4.

5.2 PM page migration and retrieval

In this section, we observe the number of pages that are moved between NORMAL and MIGRATE as the application executes. Figure 6 shows the number of pages that are migrated to NORMAL, retrieved back to MIGRATE, and the difference between the two, that is the number of pages from MIGRATE that are being used as memory pages, measured in 3 second intervals, as time progresses. Figure 6(a) is the results for when FFT is making use of 9GB of DRAM (second bar of Figure 4(a)), Figure 6(b) is for Redis making use of 15GB of DRAM (second bar of Figure 4(b)), Figure 6(c) is when both FFT and Redis are executed simultaneously, with the stress tool taking up 8GBs of DRAM. As FFT has shorter execution time, it is restarted upon termination after a 20 second lapse to show the effect of termination. The results presented are average values of three runs.

We observe that pages are being migrated and retrieved dynamically as applications execute. Figures 6(a) and (b) show that for applications run independently and alone, most of the migration and retrieval is happening in a monotonic manner. We observe that the changes are more dynamic for Figure 6(c) as FFT terminates and relinquishes the PM space that had been allocated. When run with Redis, FFT showed an average execution time of 60 seconds. Since there is a 20 second lapse before restarting its execution, we see that pages are being freed and retrieved in the vicinity of 60, 140, 220, 300 seconds, being reflected in the fluctuating red line in Figure 6(c) around these points.

6 Summary and Future Work

In this paper, we proposed a system called Storage-As-You-Go (SAY-Go) that transparently adjusts the use of PM such that PM can be used as memory as well as storage as need be. Whereas previous studies considered DIMM interface PM as storage, this study proposed to break the boundary between memory and storage so that PM is used as memory or storage as needed and for performance benefit.

In particular, in this paper, we presented the design and implementation of, what we call Persistent Memory Buddy (PMB). PMB allows for memory allocation service that can freely grow and shrink memory it is managing. We presented experimental results that show that PMB performs considerably better than the traditional swapping technique and that PM is being dynamically allocated as memory or storage space.

There are many issues to resolve to reach our eventually goal of SAY-Go. First and foremost, a file system that supports dynamic resizing of partitions is required. We are in the process of designing this file system. Even within PMB, there are many policy issues that need to be carefully analyzed. This is also on-going work.

References

[1] Linux programmer's manual: process information pseudo-filesystem. http://man7.org/linux/man-pages/man5/proc.5.html.

[2] Linux stress tool. https://people.seas.harvard.edu/ apw/stress/.

[3] PARSEC. http://parsec.cs.princeton.edu/.

[4] Redis. https://redis.io/documentation/.

[5] AGRAWAL, N., BOLOSKY, W. J., DOUCEUR, J. R., AND LORCH, J. R. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS) 3*, 3 (2007), 9.

[6] COBURN, J., CAULFIELD, A. M., AKEL, A., GRUPP, L. M., GUPTA, R. K., JHALA, R., AND SWANSON, S. NV-Heaps: Making Persistent Objects Fast and Safe with Next-Generation, Non-Volatile Memories. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[7] CONDIT, J., NIGHTINGALE, E. B., FROST, C., IPEK, E., LEE, B., BURGER, D., AND COETZEE, D. Better I/O Through Byte-Addressable, Persistent Memory. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2009).

[8] COOPER, B. F., SILBERSTEIN, A., TAM, E., RAMAKRISHNAN, R., AND SEARS, R. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the ACM Symposium on Cloud Computing (SoCC)* (2010).

[9] DULLOOR, S. R., KUMAR, S., KESHAVAMURTHY, A., LANTZ, P., REDDY, D., SANKARAN, R., AND JACKSON, J. System Software for Persistent Memory. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2014).

[10] GORMAN, M. *Understanding the Linux Virtual Memory Manager*. Prentice Hall, 2004.

[11] GRAHAM, DEBRA H. Intel's 3D XPoint^{TM} Technology Products – What's Available and What's Coming Soon. https://software.intel.com/en-us/articles/3d-xpoint-technology-products.

[12] INTEL. Intel optane SSD 900p series. hhttps://www.intel.com/content/www/us/en/products/memory-storage/solid-state-drives/gaming-enthusiast-ssds/optane-900p-series.html.

[13] JUNG, J.-Y., AND CHO, S. Memorage: Emerging persistent ram based malleable main memory and storage architecture. In *Proceedings of the International ACM Conference on International Conference on Supercomputing (ICS)* (2013).

[14] KANNAN, S., ADA, G., AND KARSTEN, S. pVM: persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the European Conference on Computer Systems (EuroSys)* (2016).

[15] KAY, T. Linux swap space. http://www.linuxjournal.com/article/10678/, 2011.

[16] KIM, J. H., MOON, Y. J., AND NOH, S. H. An Experimental Study on the Effect of Asymmetric Memory Latency of New Memory on Application Performance. In *Proceedings of the IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)* (2016).

[17] LIU, R.-S., SHEN, D.-Y., YANG, C.-L., YU, S.-C., AND WANG, C.-Y. M. Nvm duet: Unified working memory and persistent store architecture. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2014).

[18] MEYER, D. T., AND BOLOSKY, W. J. A study of practical deduplication. *ACM Transactions on Storage (TOS) 7*, 4 (2012), 14.

[19] SNIA. pmem.io persistent memory programming. http://http://pmem.io.

[20] VOLOS, H., TACK, A. J., AND SWIFT, M. M. Mnemosyne: Lightweight Persistent Memory. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2011).

[21] XU, J., AND SWANSON, S. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *Proceedings of the Conference on File and storage technologies (FAST)* (2016).

[22] ZHANG, Y., AND SWANSON, S. A study of application performance with non-volatile main memory. In *Proceedings of the IEEE Conference on Mass Storage Systems and Technologies (MSST)* (2015).