Memorage: Emerging Persistent RAM based Malleable Main Memory and Storage Architecture

Ju-Young Jung
Computer Science Department
University of Pittsburgh
Pittsburgh, PA, USA
juyoung@cs.pitt.edu

Sangyeun Cho
Memory Solutions Lab., Memory Division,
Samsung Electronics Co., Korea
Computer Science Department,
University of Pittsburgh
Pittsburgh, PA, USA
cho@cs.pitt.edu

ABSTRACT

This paper addresses new system design issues that will occur when a large quantity of emerging persistent RAM (PRAM) is put on the main memory bus of a platform. First, we anticipate that continued technology advances will enable us to integrate (portions of) the system storage within the PRAM modules on a system board. This change calls for comprehensive re-examination of the system design concepts that assume slow disk and the block I/O concept. Next, we propose *Memorage*, a system architecture that virtually manages all available physical resources for memory and storage in an integrated manner. Memorage leverages the existing OS virtual memory (VM) manager to improve the performance of memory-intensive workloads and achieve longer lifetime of the main memory.

We design and implement a prototype system in the Linux OS to study the effectiveness of Memorage. Obtained results are promising; Memorage is shown to offer additional physical memory capacity to demanding workloads much more efficiently than a conventional VM manager. Under memory pressure, the performance of studied memory-intensive multiprogramming workloads was improved by up to 40.5% with an average of 16.7%. Moreover, Memorage is shown to extend the lifetime of the PRAM main memory by 3.9 or 6.9 times on a system with 8 GB PRAM main memory and a 240 GB or 480 GB PRAM storage.

Categories and Subject Descriptors

C.0 [**Computer Systems Organization**]: General—*System architectures*; D.4.2 [**Operating Systems**]: Storage Management—*Main memory, Secondary storage*

Keywords

Emerging persistent RAM; Memory hierarchy; Management; Performance; Write endurance

1. INTRODUCTION

DRAM has been exclusively used as a platform's main memory for decades, thanks to its high performance and low cost per bit. However, DRAM main memory already accounts for 20% to 40% of the system power consumption and its portion is growing [4]. Furthermore, according to the ITRS 2011 report [23], there is no known path for the DRAM technology to scale below 20nm. Eventually, DRAM may no longer be the best technology for main memory, with new memory technologies emerging to take over its role.

Indeed, researchers have recently started exploring the use of emerging persistent RAM (PRAM) as a DRAM replacement [28, 33, 43]. They focus on overcoming the relatively low performance and write endurance of phase change memory (PCM), a type of PRAM, with clever architectural techniques. Due to its non-volatility, PRAM is also expected to provide an adequate medium for high performance storage systems. Condit et al. [16] describe a file system that takes advantage of byte-addressable PCM and Dong et al. [17] address the design issues of a PCM-based storage device. However, most prior work, including the above, emphasizes only one aspect of PRAM: random accessibility (main memory) or persistence (storage and file system). These proposals maintain the traditional main memory and storage dichotomy in resource management.

We argue in this paper that *future PRAM based systems will have to manage the PRAM main memory resource and the PRAM storage resource in an integrated manner to make the best performance and cost trade-offs.* In a realization of the idea, the physical memory resource manager will need to book-keep the status of the storage resources as well as the memory resources. The integration of resource management will allow the system to flexibly provision the available resources across the memory and storage boundary for better performance and reliability. The following technology trends support this argument:

• **There will be little characteristic distinctions between main memory resources and storage resources.** Note that there are independent research and development efforts on scalable PRAM main memory and fast "PRAM storage devices" (or "PSDs") using high-density PRAM chips. If scaling predictions of ITRS [23] are realized (from 22nm flash half pitch in 2011 to 8nm in 2024) and given that the PCM cell density is comparable to that of NAND flash (see Table 1), a single PCM die can pack 256 Gbits by year 2024. This chip capacity, estimated conservatively, enables build-

	Latency			Program energy	Allowable access unit	Retention on power-off	Write endurance	Cell density*
	read	write	erase					
PCM	20ns	100ns	N/A	100 pJ	byte	Yes	10^8~10^9	$5F^2$
STT-MRAM	10ns	10ns	N/A	0.02 pJ	byte	Yes	10^{15}	$4F^2$
ReRAM	10ns	20ns	N/A	2 pJ	byte	Yes	10^6	$6F^2$
DRAM	10ns	10ns	N/A	2 pJ	byte	No	10^{16}	$(2/3)F^2$
NAND flash	25μs	200μs	1.5ms	10 nJ	page/block	Yes	10^4~10^6	4~$5F^2$
HDD	8.5ms	9.5ms	N/A	N/A	sector	Yes	N/A	2~$3F^2$

Table 1: Comparison of emerging PRAM and existing memory/storage technologies [27]. $^*F$ is the minimum feature size of a given process technology.

ing a small form-factor memory module carrying hundreds of gigabytes. Stacked multi-chip packaging techniques have matured and packages containing 8 to 16 chips are already commercially feasible [35].

In addition, fully exploiting the high bandwidth of the PRAM modules in a platform requires that all the capacity be interfaced via the main memory bus. Researchers have already explored the benefit of placing flash memory on the main memory bus [25]. Suppose that both main memory and system storage are comprised of PRAM and are on the same memory bus; then the main memory and storage resources are no more heterogeneous than DRAM and hard disk drive (HDD) in today's systems, but are quite homogeneous. This offers an unprecedented, practical opportunity for the system to manage the resources in an integrated manner.

• **Reducing I/O software overhead becomes even more important than in the past.** Many software artifacts have been incorporated in a conventional OS to deal with the slow, block-oriented HDDs. For example, complex I/O scheduling algorithms have been implemented in the block I/O layer of the Linux OS [10]. If the current software stack is unchanged, however, the major difference in data access latency between PRAM main memory and a PSD will be dominated by the overhead of the software stack that handles I/O requests. For example, a 4-KB data transfer between a PSD and main memory can be done in hardware in a microsecond, whereas the software overhead of an I/O operation—from a user I/O request to the OS file system to the block I/O layer to the device driver and vice versa—amounts to tens of microseconds [11, 15, 27, 28]! This software overhead has been acceptable because the same data transfer operation with a HDD typically takes milliseconds and the software overhead is only a fraction of the latency.

• **Storage density grows exponentially and a typical user underutilizes the available storage capacity.** HDD and NAND flash density improvement rates have outpaced Moore's Law [22, 26]. While there are different storage usage scenarios such as PC, embedded and server environments, the entire capacity of a given storage device is rarely filled up, leaving some space unused during its lifetime. Agrawal et al. [5] measured the file system fullness and quantified the annual file system size growth rate to be only 14% on average. Moreover, the storage administrator usually performs careful storage capacity provisioning to ensure there is room for the stored data set to grow. Provisioned but unused storage capacity is, in a sense, a lost resource.

In this paper, based on the above observations, we make a case for *Memorage*, a system architecture that utilizes the system's PRAM resources in an integrated manner. With its capability to address all PRAM resources in the system, Memorage can improve the main memory performance by granting more directly accessible physical PRAM capacity of the PSD to the main memory. Moreover, it increases the lifetime of the PRAM resource by spreading writes to all PRAM capacity without being limited by the main memory-storage wall. Effectively, Memorage achieves higher system performance, higher PRAM resource utilization, and longer lifetime through *virtual over-provisioning.* Our experiments based on a prototype system find that Memorage's performance improvement potential is large; the performance of individual programs in a memory-intensive multiprogrammed workload was improved by up to 40.5%, under a high memory pressure. Our analytical analysis finds that the lifetime improvement potential is also high. For example, we obtain 3.9× or 6.9× lifetime increase when a system has 8 GB main memory and a 240 GB or 480 GB PSD.

In what follows, Section 2 will first discuss emerging PRAM technologies and anticipated architectural changes from both hardware and software perspectives. Section 3 then introduces Memorage and discusses how it can be incorporated in the Linux kernel. Proposed Memorage concepts are studied experimentally and analytically in Section 4. Lastly, related work and conclusions are summarized in Section 5 and 6.

2. BACKGROUND

2.1 Emerging Persistent RAM Technologies

There are a handful of PRAM (also called storage class memory or SCM [18]) technologies being actively developed by industry. Table 1 lists and compares three promising PRAM technologies with DRAM, NAND flash and HDD technologies: PCM, STT-MRAM (spin-transfer-torque magnetoresistive RAM) and ReRAM (Resistive RAM) [34, 38, 39, 41]. Basically, PRAMs' operations are based on sensing the resistance of a cell material rather than electrical charge. PCM is considered to be the closest (among all PRAM technologies) to mass production, with commercial and sample chips available at high densities (1 to 8 Gbits) [3, 14, 18, 34].

PRAMs have several desirable properties in common. Unlike DRAM, they are non-volatile. Compared with NAND flash, they are byte-addressable and have faster speeds. On the other hand, PRAMs have known shortcomings; PRAMs like PCM and ReRAM have limited write cycles, requiring aggressive wear-leveling in write-intensive applications. Moreover, their access latencies are longer than DRAM, and in certain cases, write latency is much longer than read latency. Likewise, write energy can be disproportionately large. Therefore, system designers must pay extra caution to hiding and reducing writes [13].

Meanwhile, ITRS [23] is anticipating multi-level cell (MLC) solutions of PRAMs. The MLC designs effectively reduce the cost per bit by packing more bits per memory cell. While

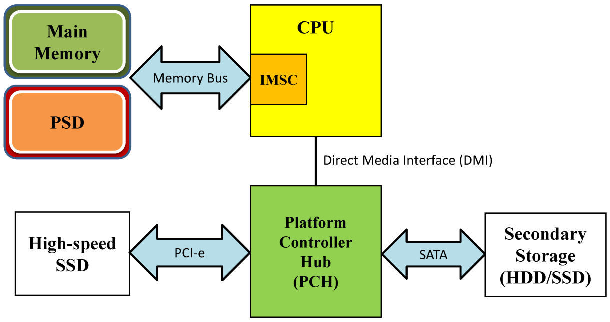


Figure 1: PRAM main memory and a PSD share the memory bus. IMSC represents Integrated Memory and Storage Controller. Note that not all storage devices in this figure have to be present simultaneously.

a single-level cell (SLC) can represent two logic values, '0' and '1' (with two resistance levels), an MLC cell stores more than two logical symbols. Future high-density MLC PRAMs may store more than two bits per cell [15, 23, 31].

Introducing MLC PRAM to a platform has several implications. First, MLC PRAM will be slower than SLC PRAM. Reportedly, MLC PCM read and write are 2 to 4 times slower [17, 32]. This is mainly due to more sensing levels for read and an iterative programming process for write [8]. The second implication is the lower write endurance of MLC PRAM because: (1) the iterative write process accelerates cell wearing; and (2) reduced resistance margins between different symbols make it harder to precisely control programming, especially when the cell is partially worn out.

Because of its higher density and lower performance, MLC PRAM is more suitable for use in a PSD than in main memory. If a PSD employs MLC PRAM and main memory uses SLC PRAM, there is a latency gap between the two resources. Still, this gap ($<10\times$) is not as significant as the gap between DRAM and HDD ($10^5\times$). There are also techniques to opportunistically enhance the speed of MLC PRAM, e.g., by treating an MLC PCM device like an SLC device when the system's capacity demand is low [8, 17, 20, 32].

2.2 Storage Attachable Hardware Interfaces

In today's commodity platforms, HDDs are the slowest hardware component connected through a relatively slow serial ATA (SATA) interface. With SATA, a disk access request must pass through multiple chips and buses (front-side bus to North Bridge to South Bridge to SATA) before it reaches the storage medium, incurring long latency. Early NAND flash SSD offerings provide a direct migration path from HDDs based on the legacy interface like SATA. However, both enterprise and consumer SSDs have quickly saturated the SATA bandwidth. Recent high-speed SSDs are attached via the higher-bandwidth PCI Express (PCI-e) interface [19].

Likewise, it is reasonable to predict that PSDs will interface with faster buses than PCI-e because PRAMs have superior performance to flash memory. To accommodate fast PSDs, a new higher-performance I/O standard may be created. However, in this case, or with legacy interfaces, the byte-addressability of a PSD could be lost. In addition, the memory bus is the highest-bandwidth bus of a platform. Hence, a PSD will be most likely and best co-located with PRAM main memory modules on a platform's memory bus.

We expect that future platforms will continue supporting legacy PCI-e and SATA interfaces. It is quite conceivable that multiple heterogeneous storage devices may co-exist. For instance, the HDD could provide ample capacity to store large file data while the PSD provides fast I/O for hot data. A particular storage configuration will be chosen based on the cost, performance and capacity trade-offs made by the user. Figure 1 illustrates a possible future computing platform with a PSD on the memory bus.

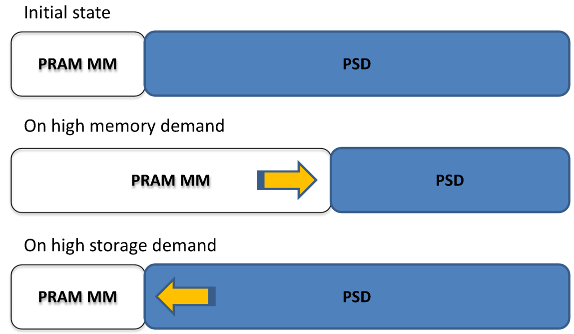


Figure 2: Illustration of the Memorage concept. The Memorage dynamically expands or shrinks the capacity of main memory (denoted "MM") and PSD on demand.

2.3 OS I/O Software Stack

Traditional OS I/O software stack and file systems have been designed for rotating HDDs. Hence, storage optimizations are mainly devoted to reducing disk seek time through efficient I/O scheduling. However, SSDs benefit little from such HDD-centric optimizations. Accordingly, as an outgrowth, system designers customize the OS I/O software stack with SSDs in mind [11, 36] and design new file system [24].

In a conventional OS I/O stack, a read or write request must first go to the block I/O layer that is responsible for I/O scheduling. The request then waits (in anticipation) for other requests that can be merged together to reduce accesses to the slow HDD. The Linux's default "completely fair queuing" (CFQ) scheduler [10] is an example of doing this. By comparison, with a fast SSD, it is not always beneficial to merge requests. Depending on the target configuration, it makes sense to bypass the block I/O scheduler and the block device driver (e.g., SCSI subsystem) all together.

Since PSDs are byte-addressable and even faster than SSDs, we expect further changes to occur. Especially, tradeoffs between different access granularities (block vs. byte) must be examined closely as many small updates will not require forming and handling blocks. Ultimately, adopting the PSD architecture warrants designing a new file system to fully exploit the fine access granularity as opposed to conventional block-based file systems. File caching policies also need close re-examination. While it is not our goal to study all possible changes, we believe our work presents an interesting first step in this direction.

3. MEMORAGE

3.1 Memorage Philosophy

Memorage tackles the inefficiency of PRAM resource utilization by collapsing the traditional static boundary between main memory and storage resources (see Figure 2). The Memorage approach is motivated by the fact that fast PRAM storage resources will likely remain underutilized if a system is designed based on the traditional dichotomy of memory and storage. It also enables us to mitigate the problem of PRAM's limited endurance with a global wear-leveling strategy that involves all available PRAM resources.

Storage capacity in a drive has increased with the improvement in storage density. Studies like Meyer et al. [30] and Agrawal et al. [5] show however that storage utilization has not been growing with the increasing storage capacity. Meyer et al. analyzed vast file system content data collected for over four weeks in 2009 in a large corporation. Agrawal et al. collected their data from 2000 to 2004 in the same company. According to their results, storage capacity has increased by almost two orders of magnitude, but the mean utilization of the capacity has actually decreased by 10% from 53% to 43%. Furthermore, 50% of the users had drives less than 40% full while 70% of the users had their drives no more than 60% full. These studies clearly suggest that a storage device in a system is likely to have *substantial unused space* during its lifetime.

Memorage aims to effectively address the above wastefulness by suggesting the following two principles:

1. Don't swap, give more memory. Under high memory pressure, a conventional OS virtual memory (VM) manager swaps out previously allocated pages into the storage to respond to memory requests. Significant performance penalty is incurred when frequent swap in and swap out operations occur. In Memorage, main memory borrows directly accessible memory resources from the PSD to cope with memory shortages. Offering more memory capacity effectively eliminates the need for costly swap operations.

2. Don't pay for physical over-provisioning. To guarantee reasonable lifetime, reliability and performance of the PRAM main memory, robust wear-leveling and garbage collection with over-provisioned capacity is required. Flash SSDs commonly resort to over-provisioning of as much as 20% of the (advertised) capacity. Over-provisioned capacity is typically hidden from the user, and may remain inefficiently utilized. In Memorage, as long as capacity planning of the PSD allows, the PSD can donate its free capacity to the main memory to relax the limited write endurance problem and facilitate wear-leveling. Effectively, Memorage offers "logical" or "virtual" over-provisioning without hiding any capacity from the user or incurring additional cost for physical over-provisioning.

To summarize, we expect two important benefits from the Memorage principles. First, by granting more directly accessible memory capacity to the physical memory pool (principle 1), the system can decrease the frequency of page faults. Because PRAM is orders of magnitude faster than traditional HDDs, avoiding the software overheads of page faults can lead to significant performance improvement. Second, by dynamically trading resources between the main memory and the storage (principle 2), lifetime becomes more manageable because the write traffic to the main memory and the storage can be re-distributed with software control.

3.2 Key Design Goals

In this subsection, we discuss three goals that have guided the design and implementation of Memorage.

• **Transparency to existing applications.** It is impractical to require re-compiling all existing applications for a new system feature to be enabled. To ensure its seamless adoption, we encapsulate Memorage inside the OS kernel and do not modify application-level interfaces. While not required, the user may configure Memorage parameters to tune resource management policies according to particular system-level objectives. Our goal is to have a Memorage system autonomously and intelligently manage the underlying PRAM resources, considering user preferences.

• **Small development efforts.** Meeting the transparency goal may impose additional complexities on system software design, especially the memory and storage subsystem of an OS. The complexities hinder the fast adoption of Memorage architecture. Thus, we aim at avoiding long development time by reusing the existing system software infrastructure whenever possible. This paper describes our prototype design in detail so that other researchers and developers can easily implement Memorage in their systems.

• **Low system overheads.** An implementation of Memorage may incur performance and memory overheads because it adds a new layer of resource control. The performance overheads are incurred when PRAM resources are transferred from the storage side to the main memory side, and vice versa. The space overheads come from book-keeping and sharing resource usages across the two sides. In this work, we design a prototype Memorage system by reusing kernel-level functions and data structures to achieve this goal. We note that the performance overheads are paid fairly infrequently, only when PRAM resources are exchanged under memory pressure situations.

3.3 Memorage Design and Implementation

We now discuss our Memorage prototype, integrated in a recent Linux kernel. The prototype Memorage system runs on a non-uniform memory architecture (NUMA) platform with a large system memory that emulates a PSD, as will be explain in Section 4.1. The general strategies used in our implementation will also apply to other OSes.

We focus on how the first principle—*Don't swap, give more memory*—is incorporated in our implementation because the current Linux kernel has no provisions for memory wear-leveling. However, we will separately study via analytical modeling how Memorage helps improve the efficiency of wear-leveling in Section 4.4. That said, incorporating the first Memorage principle requires two major changes in an OS: (1) managing the status of PRAM resources in both memory and storage together; and (2) developing a strategy to dynamically expand and shrink the main memory capacity. We subsequently expatiate our design approaches to accommodate the two changes.

3.3.1 Managing Resource Information

As we discussed in the previous section, the Memorage prototype extensively reuses the existing memory management infrastructure of Linux. For example, key data structures to keep track of the state of a node (representing per-CPU memory resources), a zone (expressing a memory region in a node) or a page remain unchanged. The existing node descriptor, *struct pglist_data*, still contains the information of a node that includes the Memorage zone, while the zone descriptor, *struct zone*, keeps holding the information of the list of active and inactive pages in the Memorage zone. Besides, the status of a page is recognized by the page descriptor, *struct page* as usual (see [10] for more details).

To acquire the status information of resources in the Memorage zone, VM manager works closely with the PSD device driver and the file system. The PSD device driver builds on a

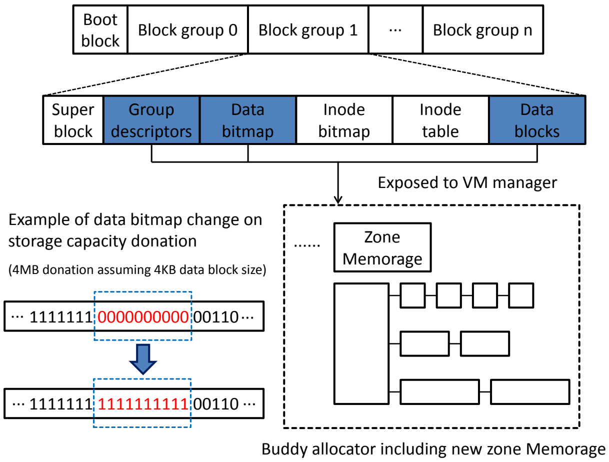


Figure 3: Memorage exposes free data blocks from a (mounted) file system to VM manager as a new zone. Buddy allocator treats the new Memorage zone the same way as other zones.

ramdisk driver to emulate the PSD with the system DRAM and can perform both block I/O operation and page-level allocation from the designated node resources. It takes as input the size of the Memorage zone, the resource amount to lend/reclaim at a time, and the memory node ID to contain the Memorage zone. The PSD device driver utilizes the Linux memory hotplug facility [1], which significantly simplifies the task of updating the OS-level information of available memory resources as they are traded between the main memory and the PSD.

PSD resource detection. An important design question that arose is: *When should the Memorage zone be prepared?* Linux for the x86 architecture obtains the memory resource information from the BIOS during the boot process and sets the memory related system-wide parameters like maximum number of page frames accordingly. To keep a system's resource discovery process consistent, our prototype system assumes that PSD resources are similarly detected at boot time and the OS book-keeps the PSD's physical resource information in addition to system memory resources. However, resource information relevant to PSD is marked to be unavailable (or offlined) on loading the PSD device driver that is also performed as a part of the boot process. As a result, OS's VM manager has the full information of PSD resources but cannot allocate a page from it until Memorage explicitly pumps in the predefined amount of PRAM resources from the PSD under memory pressure. Our prototype PSD device driver carries this out by hot-removing the PSD region (which is a memory node) during its initialization. However, the offlined PSD region is logically removed from VM manager rather than physically.

File system metadata exposure. When Memorage donates some capacity to main memory by transferring its resource to VM manager, the file system must catch and log such resource allocation events so that it does not allocate donated resources for new file data. For further illustration, Figure 3 depicts the basic on-disk layout of a file system (e.g., ext3) as well as its interaction with the buddy allocator system. The file system partitions the storage space into block groups, and each block group is comprised of *superblock*, *group descriptors*, *data block bitmap*, *inode bitmap*, *inode table* and *data blocks*. From each block group information, the following field information should be passed to the buddy memory allocator:

1. *Group descriptors* that specify the status of individual block groups such as the number of free blocks;
2. *Data block bitmap* that identifies which blocks within a block group are free blocks; and
3. *Data blocks* that stores file data in the file system.

The inode-related information does not have to be changed because blocks or pages in the PSD are free blocks with no file data on them. Given this information from the file system, the buddy allocator manages the Memorage zone just as other zones during memory allocation and deallocation. In our prototype, the Memorage zone is conceptually a node in the NUMA model that most modern OSes support. Thus, the memory allocator can utilize the fact that cost of accessing main memory nodes may be different according to the geometric distance from the requesting core.

Clean up on system reboot. In response to normal system shutdown request, Memorage mandates the file system to nullify the bitmap information previously marked for PSD donations because we assume data lifetime is over along with system reboot. By doing so, a file system consistency checker (e.g., *fsck*) can avoid reporting unwanted check result in a subsequent boot process. However, to address unexpected power failure, our prototype further needs to modify current *fsck* implementation to invalidate the inconsistent bitmap rather than fixing it up.

3.3.2 Memory Expansion and Shrinkage

When a system has little free memory pages, Memorage's VM manager dynamically expands the effective memory capacity by allocating pages from the PSD and marking the data block bitmap accordingly. Then, the file system treats the pages as if they hold file data and keeps them from being allocated. Likewise, when the file system notices a storage capacity shortage, Memorage's VM manager deallocates pages in the Memorage zone and returns them to the storage resource pool. Once the file system secures a safe number of pages, it resumes allocating them to file data. Note that Memorage's VM manager may not release the PSD pages immediately after they enter into either the inactive list or free list. This design choice helps Memorage's VM manager avoid frequent file system metadata manipulation.

The net effect of Memorage's flexible capacity sharing can be explained clearly by examining how a VM manager handles high memory pressure situations. Commodity Linux has three *watermarks* (i.e., *pages_high*, *pages_low*, and *pages_min*) used to control the invocation and sleeping of *kswapd*, the kernel swap daemon [10]. When the number of available physical page frames falls below *pages_low*, *kswapd* is invoked to swap out virtual pages and reclaim their page frames. When sufficient page frames have been reclaimed (above *pages_high*), *kswapd* is put into sleep. Essentially, *Memorage lowers the watermarks* by pumping in physical page frames borrowed from the storage capacity. As a result, *kswapd* will run less frequently with the same series of memory allocation requests. The difference between two points (1 and 2) in the figure captures the "expanded" margin to tolerate memory shortage with Memorage.

Allocator modifications for resource sharing. To realize the capacity sharing, our implementation modifies the buddy memory allocator in two ways. First, it adds a new watermark (WMARK_MEMORAGE) between *page_high* and

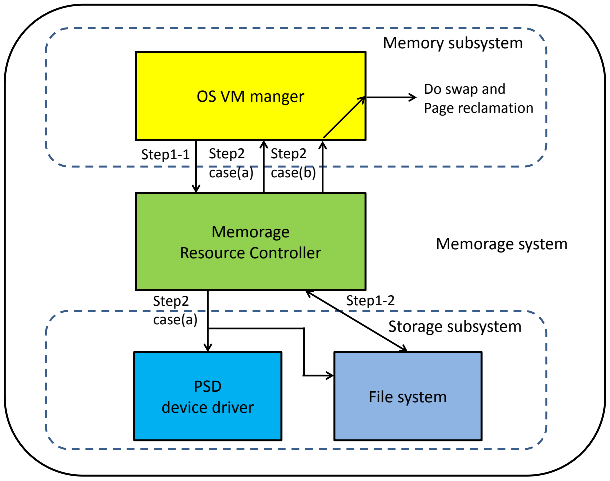


Figure 4: Cooperation between OS VM manager and storage subsystem to implement Memorage's flexible capacity sharing.

page_low. In particular, the watermark is set to one page lower than the value of *page_high* to minimize the chance of page swap and reclamation. Second, it endows the allocator with the ability to inquire about Memorage zone's resource availability when the number of allocatable pages reaches the new watermark. Figure 4 further illustrates the interaction between the memory allocator and the storage system under Memorage. Different from the conventional memory allocator, which starts page swap and reclamation under memory pressure, Memorage checks the possibility of borrowing resources from PSD before it allows the OS VM manager to swap the extant pages (Step 1-1 and 1-2). In response to this inquiry, Memorage either grants the memory allocator to allocate PSD resources (Step 2 case (a)) or suggests the allocator to follow the normal swap and reclamation process due to the unavailability of PSD resources (Step 2 case (b)).

Resource transfer size. What amount of resources the PSD provides to main memory at one time is an important knob to control Memorage's overhead associated with updating file system metadata and kernel data structures. To mitigate the overhead, Memorage allocates PSD pages in large chunks (but not all excess pages) at a time. Whereas the transfer size is a tuning parameter dependent on the system platform, our prototype system uses 2 GB granularity based on measurement results (see Figure 6). If a system later undergoes more severe memory demands that cannot be met with the current Memorage zone, then Memorage "dynamically" appends another chunk of PSD pages to the zone. By doing so, Memorage manipulates file system metadata as well as memory pool-related data structures less frequently, and sidesteps the potentially large resource exchange overhead. This mechanism is similar to a mixture of the pre-allocation feature in `xfs` or `ext4` file system and the thin-provisioning scheme for dynamic storage growth [21].

Memorage zone shrinkage. An aggressive reclamation strategy returns the donated storage resources to storage resource pool immediately after the memory resource deficit has been resolved. Even if this choice can make a right resource provisioning, it may increase resource transfer overhead in a situation of bursty memory pressure which needs previously reclaimed storage resources again. On the other hand, a lazy reclamation strategy requests Memorage to reclaim the donated storage resources only when a storage notifies its resource shortage. Although this strategy can help Memorage avoid frequent resource transfer, it may leave the memory system in an undesirable over-provisioned state. Therefore, we leverage a balanced approach which is neither aggressive nor lazy. It shrinks Memorage zone with the help of a kernel thread which reclaims the donated PSD pages when a system is not under memory pressure. Also, it considers reclaiming frequently referenced pages first because keeping those pages on the donated slower (MLC) PSD resources will degrade system performance.

3.4 Comparison with Possible Alternatives

Memorage provides a future PRAM system with a seamless evolving path to overcome the inefficient resource utilization of the traditional "static" memory management strategy. There are other possible alternative strategies to utilize the PRAM resources.

First, a platform may feature only fast SLC PRAMs on the main memory bus and the system partitions the memory space into main memory and storage capacity. This way, the system may grow and shrink each space dynamically to respond to system demands. In a sense, this approach throws away the traditional notions of main memory and storage dichotomy completely. Unfortunately, this strategy may not result in the most cost-effective platform construction because it does not exploit the cost benefits of MLC PRAMs. Moreover, deciding when to control the growth of a particular space, especially the storage, is not straightforward. By comparison, Memorage honors the traditional notion of main memory and storage capacity (that a user perceives) but manages the underlying PRAM resources such that overheads for swapping and main memory lifetime issues are effectively addressed.

Second, in lieu of dynamically reacting to memory pressure, a system may statically "give" PSD resources that correspond to the traditional swap space to main memory. In essence, the main memory capacity is increased by the swap space. When the system's memory usage does not exceed the total main memory capacity, this static approach may result in better performance than Memorage. However, since it sets apart the fixed amount of PRAM resources for main memory, it does not adapt to dynamic changes of working set sizes like Memorage does; it brings about the over- and under-provisioned memory resource problem again.

Yet another approach may create a file and delegate the file system to handle memory shortage situations with the capacity occupied by the file. This approach is analogous to giving a swap file that dynamically adjusts its size. However, since this approach must always involve the file system layer to get additional pages on memory deficit, it cannot achieve raw PRAM performance of the PSD due to unnecessary file operations unrelated to page resource allocation.

3.5 Further Discussions

Caveat for reclaiming the donated PSD pages. All pages are not reclaimable immediately on request. If the donated pages have a page reference count greater than zero when storage shortage is reported, they cannot be reclaimed shortly. Instead, those pages can be reclaimed only after their contents first migrate onto other pages. To make matters worse, not all pages can be migrated. For example, direct mapped kernel pages are not. Therefore, it is im-

System Component	Specification
Platform	HP Proliant SL160g6 Server
CPU	Two 2.4 GHz Intel Xeon E5620 processors (4 cores/processor)
Hyper-threading	8 hardware threads/processor
Last-Level Cache (L3)	12MB
Main Memory	192 GB DDR3-1066 SDRAM
Operating System	Linux kernel 3.2.8
File System	Ext3

Table 2: Experimental platform.

Configuration	Description
Baseline	4.4 GB effective memory available. This size causes significant memory shortage.
Memorage	In addition to 4.4 GB (Baseline), Memorage provide an additional 2 GB capacity on low memory. Thus, the workload sees 6.4 GB total, and this capacity is larger than the aggregate memory footprint of the workload.

Table 3: Evaluated memory configurations. Each configuration has a distinct effective memory capacity.

portant to take into account the fact that reclamation of donated pages may not be instant (reporting an error such as -EBUSY) and PSD pages should not be used for non-migratable kernel pages.

Handling a race condition between VM manager and file system. Since VM manager can manipulate the metadata of file system, designers must carefully handle potential race conditions that may be caused by accessing shared information such as block bitmap simultaneously. One possible way to address the problem is to use a conventional locking scheme that mandates serializing accesses to the shared information. Alternatively, one can design a new file system dedicated for PRAM, e.g., Wu et al. [42] delegates storage resource management to VM manager entirely. However, this requires compromising the compatibility with existing file systems. For this reason, we prefer the locking strategy to avoid modifications to file systems.

Determining lifetime ratio of main memory to PSD. How global wear-leveling is done in Memorage determines the relative lifetime of PRAM main memory and the PSD. For example, if we treat all the PRAM resources indifferently, we could make both main memory and PSD have an equal lifetime. Others may want the PSD to live longer than main memory and limit the amount of PSD capacity used in Memorage's over-provisioning. Such decisions rely on the preference of a user as well as the system upgrade period; it is hard to say simply which strategy is better. Section 4.4 examines the effect of lifetime ratio of main memory to PSD.

4. EXPERIMENTAL RESULTS

This section evaluates Memorage's benefits. After describing methodology, we will discuss separately (1) software latency of a page fault (Section 4.2); (2) application performance improvement (Section 4.3); and (3) main memory lifetime improvement (Section 4.4).

4.1 Evaluation Methodology

We employ a number of methods for evaluation. To obtain the software latency of a page fault, we measure the actual latency using fine-grained instrumentation of the Linux kernel. For application-level performance measurements, we use the Memorage prototype described in Section 3.3. Lastly, to evaluate the lifetime improvement with Memorage, we develop intuitive analytical models.

We employ two different platforms for experiments. Our latency measurement is performed on a desktop platform. The platform runs Linux kernel 2.6.35.2 with the `ext3` file system and features an Intel Core i7 quad processor, 8 MB L3 Cache, and 9 GB DDR3-1066 DRAM. Our Memorage prototype runs on a dual-socket Intel Xeon-based platform and a newer kernel. This platform has a large memory capacity organized in NUMA and eases emulating the PSD capacity. We perform application performance evaluation on this platform. Table 2 summarizes the platform's specification.

To form our workload, we select eight benchmarks from the SPEC CPU2006 suite because they are memory-bound applications [32]. The applications and their memory footprint (dynamic resident set size or RSS) are *bwaves* (873 MB), *mcf* (1,600 MB), *milc* (679 MB), *zeusmp* (501 MB), *cactusADM* (623 MB), *leslie3d* (123 MB), *lbm* (409 MB), and *GemsFDTD* (828 MB). Thus, the aggregate RSS of our multiprogrammed workload is 5.6 GB. Since our experimental platform has eight hardware threads, all applications in our workload can run simultaneously.

The first set of experiments focus on measuring the software latency of page fault handling and use *mcf*, whose RSS is 1.6 GB and is the largest of all applications. To ensure we observe many page faults (opportunities for measurements), we run *mcf* after seizing all physical memory capacity but only 1.3 GB. Measurement was done with the Linux kernel function tracer *Ftrace* [9], which instruments the entry and exit points of target functions with a small overhead.

For application performance studies, we assume that main memory is built with SLC PRAM whereas the PSD uses MLC PRAM. This assumption implies that the target system is built cost-effectively—fast, small capacity main memory and large capacity, yet slightly slow storage. As discussed in Section 2.1, this construction does not necessarily imply much longer access latency to the PSD-provided memory capacity because MLC PRAM can be read and programmed like SLC PRAM. We assume that the PRAM capacity donated to the main memory realm (Memorage) and the PSD capacity reserved for swap space (conventional system) are operating in the fast SLC mode. Given the assumptions, we emulate both the PRAM main memory and the PRAM PSD with the host machine's DRAM. In particular, all applications are pinned to run on the first NUMA node only. The PSD capacity is offered by the second NUMA node. The swap partition is implemented using a ramdisk.

As the primary metric for performance comparison, we use the program execution time to complete each of eight applications in our workload, started at once. Given the multi-core CPU with eight hardware contexts, this simple metric is intuitive and relevant. Co-scheduling of applications also ensures that platform resources are not underutilized. To reduce the effect of measurement noises, we repeat experiments three times and average the results. We also reboot the system after each experiment to keep the system state fresh and ensure that our workload runs under as much identical system conditions as possible.

We consider two target machine configurations shown in Table 3: Baseline and Memorage. The Baseline configura-

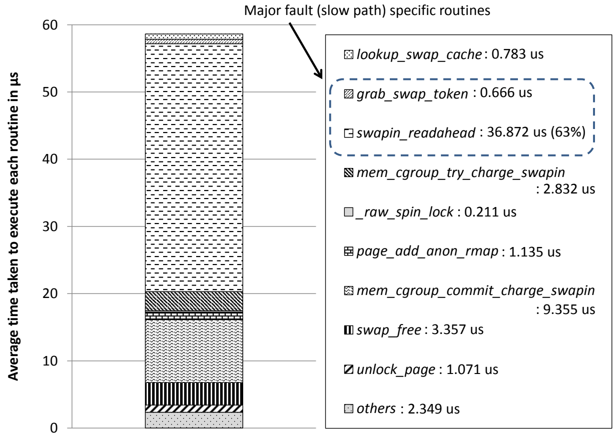


Figure 5: Breakdown of software latency of the Linux page fault handler. Routines in the legend are invoked in order.

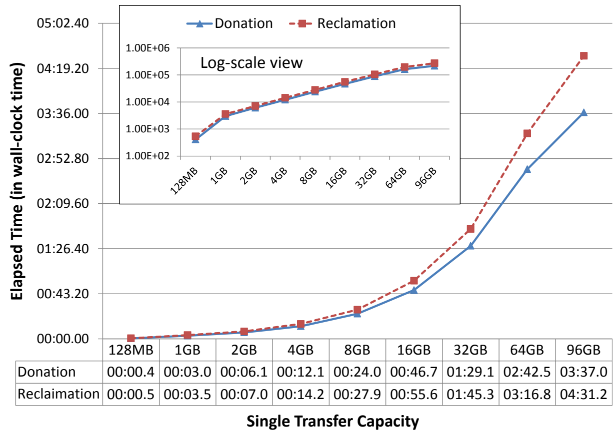


Figure 6: The latencies (in min:sec.ms) needed for Memorage to offer and retrieve PSD resources as the transfer size changes.

tion offers the total physical memory capacity of 4.4 GB. Note that this capacity is smaller by about 20% than the studied workload's aggregate footprint (5.6 GB). Accordingly, this configuration exposes the impact of frequent swap operations in a conventional system under memory pressure. The Memorage configuration offers 2 GB of additional memory capacity from the PSD.

4.2 Software Latency of a Page Fault

In this section, we obtain and report two latencies, one for "fast path" (taken for a minor fault) and another for "slow path" (major fault). A minor fault happens when the swap-in operation finds the missing page in the in-memory cache. On the other hand, a major fault requires fetching the missing page from the swap space on the storage device (PSD). On our platform, the fast path latency was measured to be 21.6 μs and the slow path latency was 58.6 μs. We find that even the fast path places nontrivial software latency on the critical path. Considering the capabilities of the underlying hardware, 21.6 μs is not a small number at all. For instance, the latency to read a 4 KB page can be as small as 0.47 μs with the 8.5 GB/sec DDR3-1066 DRAM in our platform.

Figure 5 breaks down the page fault handling latency according to the routines involved. Most routines are executed on both the fast path and the slow path. Two routines are executed only on the slow path, `grab_swap_token()` and `swapin_readahead()`, and they account for a dominant portion in the entire latency of the slow path—37.5 μs of 58.6 μs. The `swapin_readahead()` turns out to be the most time-consuming; it reads the missing page from the swap area and performs DRAM copying. This routine's latency grows with the number of pages that should be brought in, because the page reclaiming routine must first make enough room for the pages under high memory pressure. Another major contributor, responsible for 12.18 μs, is a family of "memory control group" routines (prefixed with `mem_cgroup`). Their main goal is to reduce the chances of swap thrashing.

After all, given the long latency required on each page fault, a memory-bound application's performance will suffer if its memory requirements are not fully satisfied due to memory shortage. The situation can be considerably relieved with Memorage because it grants more physical memory capacity to the system and satisfies memory demands from applications directly without involving time-consuming page fault handling.

4.3 Application Performance

The result of the previous section suggests that even if we provide a very fast swap space with a PSD, the page fault overhead will remain high because of the long software latency. Let us now turn our attention to evaluating the benefit of Memorage by comparing the application-level performance measured under different memory configurations.

Before we evaluate the application performance, we first examined the achievable memory bandwidth of our experimental platform by running STREAM benchmark [29]. This allows us to quantify the performance difference between main memory (on the local node) and the emulated PSD (on the remote node). Our result shows that main memory achieves higher bandwidth than the PSD by about 15% on average. We consider this difference to be a reasonable artifact of our experimental environment, as reading from and writing to MLC PRAMs (even if they are operating in the SLC mode) could take slightly longer than SLC PRAMs.

Next, we measure the time needed for our Memorage implementation to offer physical resources from the PSD to main memory or vice versa. This latency represents an artifact of Memorage that applications would not experience if sufficient memory were given to them initially, and is paid only when the physical resources are transferred between main memory and the PSD, not on each (minor) page fault. Figure 6 presents our result, as a function of the amount of memory capacity to donate or reclaim at one time. The plot clearly shows that the measured latency increases linearly, proportional to the transferred data size. Based on the result, we determined that a 2 GB transfer size is a reasonable choice during our application-level performance study.

Figure 7 presents the performance of Memorage normalized to that of Baseline. We show results for individual benchmark applications in the workload. Performance improvement rate varies among the applications. Six out of eight applications gained significant performance improvement. Memorage improves the total execution time by 16.5% on average and by up to 40.5% in mcf, compared to Baseline. mcf is the most memory demanding application in the workload and it benefited the most. On the other hand, leslie3d saw little performance improvement.

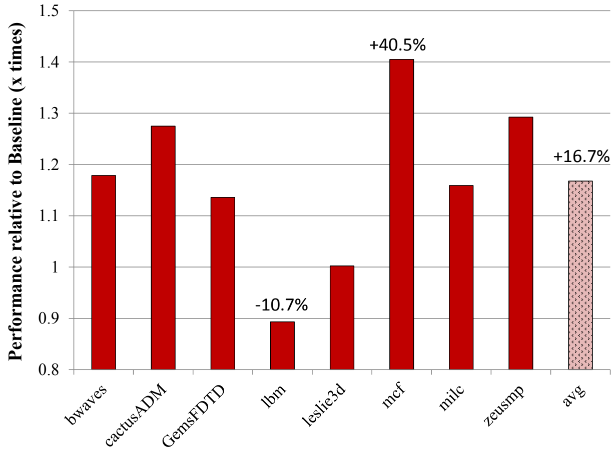


Figure 7: Relative performance of benchmarks with Memorage (based on wall clock time measurement). Performance improvement can be identified with a value greater than 1.

The plot also shows that all applications achieved performance improvement with the additional memory capacity offered by Memorage, except one application; lbm actually loses performance with more memory. This seemingly counter-intuitive result is caused by the uncontrolled resource contention in the processor core (hyper-threading) and the shared cache. lbm has a relatively small memory footprint that can easily be cached. Hence, when other more memory-demanding applications are blocked waiting for memory allocation (in Baseline), it actually gets more CPU cycles. As a result, lbm grabs shared cache capacity it needs and executes faster. We also find that slight memory shortage (e.g., several dozens of MBs) is successfully handled by *kswapd*, whereas the large gap between the total required memory size and the currently available memory size (20% in our experiment) is hard to overcome with the Linux page reclamation capability, resulting in crawling software latencies very often.

Figure 8 shows that Memorage dramatically reduces the portion of system time in the total execution time of the studied benchmarks. The programs spend a large portion of their execution time in system execution under heavy memory pressure because they have to block (in the sleep state) and yield the CPU to other processes until the faulting page becomes available through page fault handling. Because this handling is slow, the faulting process may be blocked for a long time. The increase of user time with Memorage implies that CPU cycles are spent on useful work of the user application. More user time results in fast program execution time and gives the system more chances to serve other user applications, improving the overall system throughput.

Finally, Figure 9 depicts the average number of dynamic memory instructions executed between two successive major or minor page faults. In Memorage—having generous memory capacity—the system rarely experiences a major fault. Baseline suffers a sizable number of major faults, as many as three to six orders of magnitude more major faults than Memorage. It also shows that minor page faults occur more frequently than major faults. They occur even when there is enough memory capacity to elude memory pressure because modern OSes implement many in-memory cache structures and resource management schemes. For example, the widely used copy-on-write mechanism may cause a minor page fault

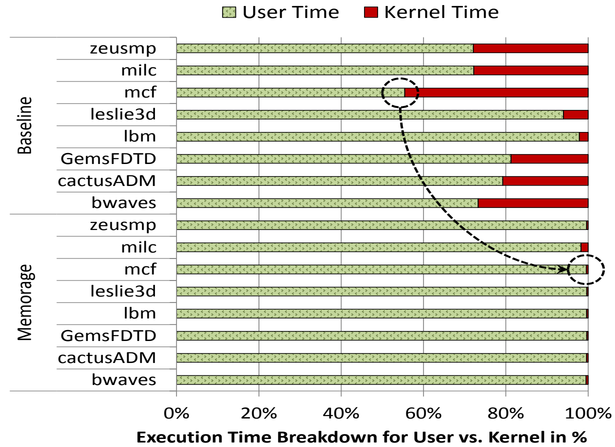


Figure 8: Total execution time is normalized to highlight the relative time spent in user applications and system routines.

due to a write on a shared page. Therefore, it is not surprising that Memorage is relatively insensitive to the number of minor faults, which have smaller impact on system performance compared to major faults. Nonetheless, Memorage decreases the number of minor faults as well because page faults often incur additional minor faults (e.g., swap cache hit) during fault handling but Memorage avoids them.

4.4 Main Memory Lifetime

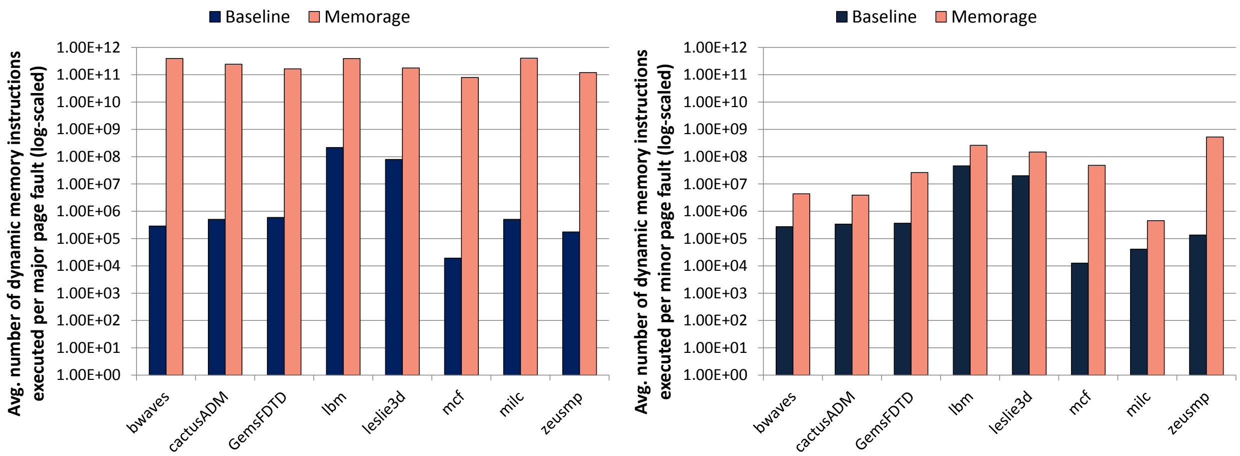
Main memory lifetime improvement when system lifetime is maximized. In a typical platform use scenario where main memory update rate is substantially higher than storage update rate, the system lifetime would be determined by the main memory lifetime. In this section, we analytically obtain the lifespan improvement of PRAM main memory with Memorage's virtual over-provisioning. Let us first focus on the case when the system lifetime is maximized (i.e., main memory lifetime equals storage lifetime).

Let L_m and L_s be the lifespan of PRAM main memory and PSD in the conventional system, respectively. They represent the time taken until all PRAM cells are worn out through memory and storage writes. Also, let C_m and C_s be the main memory capacity and the PSD capacity and let E_m and E_s be the specified write endurance of PRAM resources for main memory and PSD. Then, in the conventional system, the total data volume, D_m and D_s, writable to the main memory or the PSD before their write endurance limit is reached, are: $D_m = E_m \cdot C_m$ and $D_s = E_s \cdot C_s$.

Now, let B_m and B_s denote the average data update rate or write data bandwidth, for the main memory and the PSD, respectively. Then the lifetime of the two entities are calculated by: $L_m = D_m/B_m$ and $L_s = D_s/B_s$. At this point, we assume that perfect wear-leveling is in place for both the main memory and the PSD.

In order to relate the resources initially dedicated to main memory and PSD, we introduce α and β. Then, $C_s = \alpha \cdot C_m$ and $B_m = \beta \cdot B_s$. Because storage capacity is in general larger than that of main memory ($C_s > C_m$) and the data update rate of main memory is higher than that of storage ($B_m > B_s$), $\alpha > 1$ and $\beta > 1$ would normally hold. Similarly, we introduce γ to relate the endurance limit of the main memory and the PSD. That is, $E_m = \gamma \cdot E_s$. We normally expect γ to be greater than 1.

On a system with Memorage, let L_{new} be the lifespan



(a) Avg. number of memory instructions per major fault

(b) Avg. number of memory instructions per minor fault

Figure 9: Memorage reduces the impact of major and minor page faults by increasing the average number of memory instructions between two faults. A larger value means that more memory references are made without a page fault.

of the main memory. Ideally, Memorage could expose the whole PRAM resource capacity to global wear-leveling because it manages all PRAM resources. If we define D_{new} and B_{new} to be the total writable data volume and the data update rate for the total, Memorage-managed PRAM capacity, we have $L_{new} = D_{new}/B_{new}$, where $D_{new} = E_m \cdot C_m + E_s \cdot C_s$ and $B_{new} = B_m + B_s$. Finally, by rewriting L_{new} we obtain:

$$\begin{aligned} L_{new} &= \frac{E_m \cdot C_m + E_s \cdot C_s}{B_m + B_s} = \frac{E_m \cdot (C_m + \frac{\alpha}{\gamma} \cdot C_m)}{B_m + \frac{1}{\beta} \cdot B_m} \\ &= \frac{E_m \cdot C_m \cdot (1 + \frac{\alpha}{\gamma}) \cdot \beta}{B_m \cdot (1+\beta)} = L_m \cdot \frac{(1+\frac{\alpha}{\gamma}) \cdot \beta}{(1+\beta)} \quad (1) \end{aligned}$$

Equation (1) captures the key trade-offs that determine the new lifetime. For example, with a higher α (i.e., storage is larger than memory), the main memory lifetime increases. If γ is larger, implying that the write endurance of the main memory is better than the write endurance of the PSD, the relative benefit of global wear-leveling of PRAM resource decreases. Finally, given that α/γ is reasonably greater than 0 (e.g., PSD capacity is large enough and/or the write endurance of the PSD is close to that of the main memory), β determines the overall lifetime gain. With a larger β, the lifetime improvement increases.

Suppose for example a platform that has 8 GB main memory and a 240 GB or 480 GB PSD (α is 30 or 60, common in high-end notebooks). Figure 10 illustrates how the lifetime improvement of PRAM main memory changes as we vary the relative data write bandwidth to main memory and PSD (β). We assumed γ is 10. The lifetime improvement is shown to rapidly reach a maximum value, even when β is small—write bandwidth difference is small (e.g., see the points near $\beta = 10$). Even in an unrealistic worst-case scenario of $\beta = 1$, Memorage achieves 2× and 3.5× longer lifetime than the conventional system. Realistically, write bandwidth seen by the main memory tends to be much larger (β is large) [2,33], and hence, we expect that the large main memory lifetime improvement of Memorage will be effectively achieved.

Understanding trade-offs in global wear-leveling. Our formulation so far assumed that $L_m = L_s = L_{new}$ and a perfect, global wear-leveling method with zero overhead. To gain insights about realistic wear-leveling, we consider a hypothetical wear-leveling method where the main memory borrows an extra capacity of η from the PSD constantly. This borrowed capacity resides within the main memory realm for time τ and is then returned back. The PSD immediately lends a fresh capacity of η to replace the returned capacity. To improve lifetime, the main memory "rests" its own capacity of η, covered by the borrowed capacity.

The hypothetical wear-leveling method follows the spirit of Memorage in two ways. First, it involves the available physical resources in the main memory and the PSD only, without assuming any over-provisioned physical resources. Second, it uses borrowed capacity from the PSD to improve the main memory lifetime, across the traditional memory and storage boundary. Furthermore, the method exposes two important trade-offs (η and τ) a realistic wear-leveling scheme may also have. η captures the amount of provisioned capacity and determines the degree of lifetime improvement. On the other hand, τ dictates the frequency of resource exchange, and hence, reveals the overhead of resource management and the potential wear amplification.

Let us assume that $\eta < C_m < C_s$ (i.e., the borrowed capacity is relatively small, compared with C_m and C_s) and let L'_m and L'_s denote the new lifetime of the main memory and the PSD under the hypothetical wear-leveling scheme. Deriving the new lifetimes is fairly straightforward.

$$L'_m = (D_m \cdot C_m)/(B_m \cdot (C_m - \eta)),\ L'_s = D_s/(B_s + \frac{\eta}{C_m} \cdot B_m + \frac{2\eta}{\tau})$$

The effect of exchanging resource is manifested by the transfer of bandwidth from the main memory to the PSD ($\eta \cdot B_m/C_m$). The overhead of resource trading is revealed by the added bandwidth ($2\eta/\tau$) to the PSD side.
Introducing a new variable $h = \eta/C_m$, main memory lifetime improvement and PSD lifetime degradation are:

$$L'_m/L_m = \frac{1}{(1-h)}, \qquad L_s/L'_s = 1 + \beta \cdot h + \frac{2h \cdot C_m}{\tau \cdot B_s} \quad (2)$$

As expected, the memory side lifetime improvement is a function of h (h is the relative size of η to C_m). It is also shown that h and β (B_m/B_s) plays a role in the PSD lifetime. Intuitively, the PSD's lifetime degrades faster if: (1) the main memory bandwidth is relatively large and (2) larger capacity is delegated from the PSD to the main memory. Lastly, the wear-leveling overhead becomes larger with the

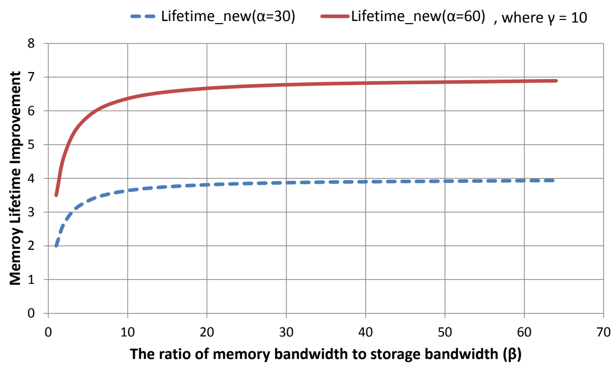


Figure 10: Main memory lifetime improvement.

traded capacity size and decreases with a longer capacity trading period and higher storage side bandwidth.

To express the above as a function of h and τ only, let us fix other variables. Like before, imagine a platform with 8 GB main memory and a 480 GB PSD. Also, assume B_s = 12 GB/day [2] and $\beta = 720$ (large bandwidth difference between main memory and PSD).

Figure 11 plots Equation (2). It is shown that the main memory lifetime increases with h while the lifetime of PSD decreases. Furthermore, the degradation of PSD lifetime is affected substantially by the choice of τ (expressed in "days"). For example, the degradation ratio difference between $\tau = 1$ (resource exchange occurs once a day) and $\tau = 0.01$ (hundred times a day) was more than an order of magnitude. Given the trend, how would the user choose h and τ? A feasible strategy would consider the lifetime improvement or degradation ratio target. For example, if the user desires at least 2× main memory lifetime improvement, h need to be 0.5 or larger (shown in the lower circle on the plot). If the same user would like the maximum PSD lifetime degradation of 1,500, he/she could choose $\tau = 0.0001$ and ensure $h = 0.7$ or less (upper circle). The user could use any value between 0.5 and 0.7 for h. This concrete example demonstrates that the developed model is powerful and can guide wear-leveling management policies.

The improvement of main memory lifetime comes at a high cost of degrading the PSD lifetime. In the previous example, obtaining 2× improvement in the main memory lifetime corresponds to 1,000× PSD lifetime degradation. This cost may appear excessive. However, in fact, the cost is justified when the system lifetime is limited by the main memory lifetime. For example, if $E_s = 10^5$, the expected lifetime of the 480 GB PSD is over 10,000 years when B_s = 12 GB/day. When $E_m = 10^6$, the lifetime of the 8 GB PRAM main memory, even with perfect wear-leveling, is only 2.5 years at B_m = 100 MB/s. In this case, reducing the PSD lifetime from 10,000 years to 10 years (1,000× degradation) to increase the main memory lifetime to 5 years makes perfect sense. Our analysis demonstrates that Memorage's ability to trade resources between PSD and main memory is extremely valuable toward improving not only the performance but also the lifetime of a platform.

5. RELATED WORK

The work by Freitas et al. [18] gives an excellent overview of the PRAM technologies. They suggest that a PRAM is a "universal memory," providing capacity to both main memory and storage of a platform. They also suggest that the

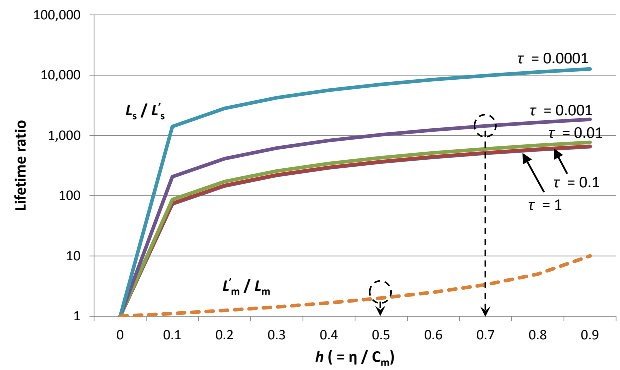


Figure 11: Main memory lifetime improvement and PSD lifetime degradation as a function of h and τ.

PCM technology is closest to mass production. However, they do not discuss in detail the notion of system-wide, dynamic co-management of memory and storage resources in an integrated framework and how the notion can be realized.

Recently the computer architecture community paid due attention to building PCM main memory [28, 33, 43]. Because PCM has lower performance and smaller write endurance than DRAM, the focus was on developing techniques to hide the long PCM access latency, reduce write operations, and distribute wearing among PCM cells. These studies looked only at the main memory architecture.

On the other hand, Caulfield et al. [11, 12] and Akel et al. [6] evaluated the potential of PRAM as a storage medium. They studied the performance of a block storage device that employs PRAM and is connected to the host system through the high bandwidth PCI-e interface. Furthermore, like our work, Caulfield motivated the need to revamp the key OS components like the block I/O layer; however, they assume a conventional DRAM main memory and stick to the traditional main memory and storage dichotomy. Accordingly, their work focused only on streamlining the storage access path in the OS and the storage hardware built with PRAM.

Work done by other researchers also reveals that the maximum SSD performance is not realized without streamlining the traditional I/O software layers. Accordingly, they propose specific techniques to optimize the OS I/O stack and file system [24,36,37]. However, their concerns so far are for NAND flash SSDs on a legacy interface rather than byte-addressable PSDs. Even if these changes are applicable to the PSD, the techniques are strictly for the storage side of the PRAM technology.

Badam et al. [7] proposed to use SSD as main memory extension rather than storage by providing users with new set of APIs to explicitly allocate pages from SSD. While the method improves the performance of large memory footprint applications, it requires applications using the APIs to be recompiled. Also, they treat SSD just as memory resources without considering the role as file storage. In our proposal, PSD is not only used as memory extension without modifications to an application, but also keeps performing its innate duty to store files.

Qureshi et al. [32] and Dong et al. [17] proposed to improve the access latency of MLC PCM by adaptively changing the operating mode from slow MLC to fast SLC mode. While the former focused only on main memory, the latter dealt only with storage. By comparison, Memorage makes use of both main memory and storage resources together to obtain the best system performance. Memorage is a flexible architecture in that it neither binds itself to the PCM technology, nor does it require the use of fast page mode in MLC.

Finally, there are two inspiring studies that address the inefficient resource utilization in main memory and the storage system. Waldspurger [40] shows that memory usages are uneven among co-scheduled virtual machines and proposes "ballooning" to flexibly re-allocate excess memory resources from a virtual machine to another. This technique helps manage the limited main memory resources more efficiently. Similarly, Hough et al. [21] present "thin provisioning" to enhance the storage resource utilization by allocating storage capacity on demand. While both proposals tackle an important problem, their focus is limited to a single resource—main memory or storage (but not both).

Compared to the above prior work, we address the system-level performance and resource utilization issue of a future platform whose main memory and storage capacity collocated at the memory bus are both PRAM. A Memorage system collapses the traditional main memory and the storage resource management and efficiently utilizes the available PRAM resources of the entire system to realize higher performance and longer lifetime of the PRAM resources.

6. CONCLUSIONS

Emerging persistent RAM (PRAM) technologies have the potential to find their place in a computer system and replace DRAM and (a portion of) traditional rotating storage medium. This paper discussed potential architectural and system changes when that happens. Furthermore, we proposed Memorage, a novel system architecture that synergistically co-manages the main memory and the storage resources comprised of PRAM. Our experimental results using a prototype system show that Memorage has the potential to significantly improve the performance of memory-intensive applications (by up to 40.5%) with no additional memory capacity provisions. Furthermore, carefully coordinated PRAM resource exchange between main memory and PRAM storage is shown to improve the lifetime of the PRAM main memory (by up to 6.9×) while keeping PSD lifetime long enough for the studied system configurations. Memorage presents a practical, plausible evolution path from the long-standing memory-storage dichotomy to integration and co-management of memory and storage resources.

Acknowledgements

This work was supported in part by the US NSF grants: CCF-1064976 and CNS-1012070.

7. REFERENCES

[1] Memory hotplug. http://www.kernel.org/doc/Documentation/memory-hotplug.txt, 2007.

[2] Nand evolution and its effects on solid state drive (ssd) usable life. http://www.wdc.com, 2009.

[3] Micron announces availability of phase change memory for mobile devices. http://investors.micron.com/releasedetail.cfm?ReleaseID=692563, 2012.

[4] N. Aggarwal et al. Power-efficient DRAM speculation. *HPCA*, pages 317–328, 2008.

[5] N. Agrawal et al. A five-year study of file-system metadata. *TOS*, 3(3):1553–3077, Oct. 2007.

[6] A. Akel et al. Onyx: A protoype phase-change memory storage array. *HotStorage*, pages 2–2, 2011.

[7] A. Badam and V. Pai. Ssdalloc: hybrid ssd/ram memory management made easy. *NSDI*, pages 16–16, 2011.

[8] F. Bedeschi et al. A bipolar-selected phase change memory featuring multi-level cell storage. *JSSC*, 44:217–227, 2009.

[9] T. Bird. Measuring function duration with Ftrace. *Japan Linux Symposium*, 2009.

[10] D. Bovet and M. Cesati. *Understanding the Linux Kernel.* O'Reilly, 2005.

[11] A. Caulfield et al. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. *MICRO*, pages 385–395, 2010.

[12] A. Caulfield et al. Providing safe, user space access to fast, solid state disks. *ASPLOS*, pages 387–400, 2012.

[13] S. Cho and H. Lee. Flip-N-write: a simple deterministic technique to improve PRAM write performance, energy and endurance. *MICRO*, pages 347–357, 2009.

[14] Y. Choi et al. A 20nm 1.8V 8gb PRAM with 40MB/s program bandwidth. *ISSCC*, pages 46–48, 2012.

[15] G. F. Close et al. A 512mb phase-change memory (pcm) in 90nm cmos achieving 2b/cell. *VLSIC*, pages 202–203, 2011.

[16] J. Condit et al. Better I/O through byte-addressable, persistent memory. *SOSP*, pages 133–146, Oct. 2009.

[17] X. Dong and Y. Xie. AdaMS: Adaptive MLC/SLC phase-change memory design for file storage. *ASP-DAC*, pages 31–36, 2011.

[18] R. Freitas and W. Wilcke. Storage-class memory: The next storage system technology. *IBM J. of R & D*, 52(4-5):439–448, 2008.

[19] FusionIO. ioDrive2 datasheet. http://www.fusionio.com/platforms/iodrive2, 2012.

[20] L. Grupp et al. Characterizing flash memory: anomalies, observations, and applications. *MICRO*, pages 24–33, 2009.

[21] G. Hough. 3par thin provisioning: Eliminating allocated-but-unused storage and accelerating roi. Technical report, 3PAR Coporation, 2003.

[22] C. Hwang. Nanotechnology enables a new memory growth model. *the IEEE*, 91(11):1765 – 1771, 2003.

[23] ITRS. 2011 edition. http://public.itrs.net, 2011.

[24] W. Josephson et al. Dfs: A file system for virtualized flash storage. *TOS*, 6(3):14:1–14:25, Sept. 2010.

[25] D. Kim et al. Architecture exploration of high-performance PCs with a solid-state disk. *Trans. Computers*, 59(7):878–890, 2010.

[26] D. Klein. The future of memory and storage: Closing the gaps. Microsoft WinHEC, 2007.

[27] M. Kryder and C. Kim. After hard drives–what comes next? *Trans. Magnetics*, 45:3406 – 3413, 2009.

[28] B. Lee et al. Architecting phase change memory as a scalable DRAM alternative. *ISCA*, June 2009.

[29] J. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. http://www.cs.virginia.edu/stream/, 1995.

[30] D. Meyer and W. Bolosky. A study of practical deduplication. *TOS*, 7(4):14, 2012.

[31] H. Pozidis et al. Enabling technologies for multi-level phase change memory. *E\PCOS*, 2011.

[32] M. Qureshi et al. Morphable memory system: A robust architecture for exploiting multi-level phase change memories. *ISCA*, pages 153–162, June 2010.

[33] M. Qureshi et al. Scalable high performance main memory system using phase-change memory technology. *ISCA*, June 2009. IBM T.J. Watson RC.

[34] S. Raoux et al. Phase-change random access memory: A scalable technology. *IBM J. of R & D*, 52(4-5):465–480, 2008.

[35] Samsung. Fusion Memory. http://www.samsung.com, 2010.

[36] M. Saxena and M. Swift. Flashvm: virtual memory management on flash. *USENIXATC*, pages 14–14, 2010.

[37] E. Seppanen et al. High performance solid state storage under linux. *MSST*, pages 1–12, 2010.

[38] A. Smith and Y. Huai. STT-RAM - a new spin on universal memory. *Future Fab International*, 23:28–32, 2007.

[39] D. Strukov et al. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[40] C. Waldspurger. Memory resource management in vmware esx server. *SIGOPS OSR*, 36(SI):181–194, Dec. 2002.

[41] Z. Wei et al. Highly reliable taox reram and direct evidence of redox reaction mechanism. *IEDM*, pages 1–4, 2008.

[42] X. Wu and N. Reddy. SCMFS: a file system for storage class memory. *SC*, pages 39:1–39:11, 2011.

[43] P. Zhou et al. A durable and energy efficient main memory using phase change memory technology. *ISCA*, 2009.