IMPROVING THE PERFORMANCE OF HYBRID MAIN MEMORY THROUGH SYSTEM AWARE MANAGEMENT OF HETEROGENEOUS RESOURCES

by

Juyoung Jung

B.S. in Information Engineering, Korea University, 2000

Master in Computer Science, University of Pittsburgh, 2013

Submitted to the Graduate Faculty of

the Kenneth P. Dietrich School of Arts and Sciences in partial

fulfillment

of the requirements for the degree of

Doctor of Philosophy in Computer Science

University of Pittsburgh

2016

UNIVERSITY OF PITTSBURGH

KENNETH P. DIETRICH SCHOOL OF ARTS AND SCIENCES

This dissertation was presented

by

Juyoung Jung

It was defended on

December 7, 2016

and approved by

IMPROVING THE PERFORMANCE OF HYBRID MAIN MEMORY THROUGH SYSTEM AWARE MANAGEMENT OF HETEROGENEOUS RESOURCES

Juyoung Jung, PhD

University of Pittsburgh, 2016

Modern computer systems feature memory hierarchies which typically include DRAM as the main memory and HDD as the secondary storage. DRAM and HDD have been extensively used for the past several decades because of their high performance and low cost per bit at their level of hierarchy. Unfortunately, DRAM is facing serious scaling and power consumption problems, while HDD has suffered from stagnant performance improvement and poor energy efficiency. After all, computer system architects have an implicit consensus that there is no hope to improve future system's performance and power consumption unless something fundamentally changes.

To address the looming problems with DRAM and HDD, emerging Non-Volatile RAMs (NVRAMs) such as Phase Change Memory (PCM) or Spin-Transfer-Toque Magnetoresistive RAM (STT-MRAM) have been actively explored as new media of future memory hierarchy. However, since these NVRAMs have quite different characteristics from DRAM and HDD, integrating NVRAMs into conventional memory hierarchy requires significant architectural re-considerations and changes, imposing additional and complicated design trade-offs on the memory hierarchy design. This work assumes a future system in which both main memory and secondary storage include NVRAMs and are placed on the same memory bus.

In this system organization, this dissertation work has addressed a problem facing the efficient exploitation of NVRAMs and DRAM integrated into a future platform's memory hierarchy. Especially, this dissertation has investigated the system performance and lifetime *improvement endowed by a novel system architecture called Memorage which co-manages all available physical NVRAM resources for main memory and storage at a system-level. Also, the work has studied the impact of a model-guided, hardware-driven page swap in a hybrid main memory on the application performance. Together, the two ideas enable a future system to ameliorate high system performance degradation under heavy memory pressure and to avoid an inefficient use of DRAM capacity due to injudicious page swap decisions.*

In summary, this research has not only demonstrated how emerging NVRAMs can be effectively employed and integrated in order to enhance the performance and endurance of a future system, but also helped system architects understand important design trade-offs for emerging NVRAMs based memory and storage systems.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

1.0 INTRODUCTION

The memory hierarchy of modern computing systems beyond processor cache hierarchies has a hierarchical organization which is comprised of the system main memory and secondary storage subsystem. DRAM and HDD have been the best technologies for the main memory and storage, respectively, for the past several decades, because they provide a satisfactory performance and cost-per-bit required at each level of the hierarchy. Unfortunately, the DRAM popularity is being challenged by the scaling problem and limited power budget of DRAM technology. Similarly, the HDD keeps disappointing system designers with its slow yearly performance improvement rate and power-burning mechanical operation.

To address these problems, researchers recently started exploring the use of emerging persistent or non-volatile RAMs (shortly, PRAM, PM, or NVRAMs) [1] as a new medium of the memory hierarchy design. The emerging NVRAMs such as Phase Change Memory (PCM) and STT-RAM are differentiated from the traditional DRAM and HDD technology in that they are scalable, byte-addressable, persistent, and energy-efficient. Because of their many desirable characteristics other than the DRAM and HDD, a future platform is anticipated to include NVRAMs as main memory, storage, or both. However, emerging NVRAMs also have their drawbacks, and system designers must cautiously take care of them. For example, PCM technology has slower access latency than DRAM, an unbalanced read and write performance, and a limited cell endurance. Therefore, the integration of emerging NVRAMs into the conventional memory hierarchy necessitates new architectural changes and support to fully exploit their strengths while hiding the aforementioned shortcomings.

[1] The terminology NVRAM may cause confusion because many studies have often used the term to represent NAND flash memory technology which is neither byte-addressable nor scalable. As such, to avoid the confusion, we explicitly call the NAND flash memory as an *existing* NVRAM, not *emerging* NVRAM with which this dissertation work deals. The NVRAM denotes an *emerging* NVRAM unless otherwise mentioned.

In this dissertation, we have investigated the integration of emerging NVRAMs into the existing memory hierarchy, and tackled the lack of NVRAM awareness in the traditional memory hierarchy design to improve the performance and lifetime of an NVRAM-based future system. Specifically, a primary goal throughout this study is to improve the performance of main memory, which is a dominant system component that greatly influences the overall system performance. Other advantages of emerging NVRAMs such as the energy efficiency and non-volatility are also interesting to explore, but we will leave them as our future work.

1.1 RECENT TRENDS OF MEMORY HIERARCHY DESIGN

In modern computer systems, having DRAM main memory and HDD secondary storage has been a de-facto standard organization of memory hierarchy design. However, computer system designers are recently facing critical challenges on the memory hierarchy design due to the changed computing environment.

First and most importantly, there are much larger memory demands in a system than ever before. Keeping up with Moore's Law [101, 84], multi-core [85, 73] and many-core processors [45, 40, 104] with multi-level cache hierarchies have become a mainstream of processor design in the last decade, and the processors produce a great deal of memory requests to service within a given time by running multiple applications on multiple cores simultaneously. To make matters worse, the memory footprint size of applications continues to grow and is unlikely to stop increasing in the near future [69, 117]. To meet the increased memory demand without a detrimental application performance degradation, contemporary memory systems have multiple memory channels to serve numerous memory requests concurrently and independently, and exploit the memory-level parallelism among them. Also, the memory controllers associated with these memory channels have been integrated into a processor chip and the DRAM device control signals are now generated completely on chip. This integration enables cores to speed up communications between cores and memory controllers and as a result, achieve performance improvement [8, 61]. However, a continuing growth of the memory demand requires current memory systems to further increase memory bandwidth and bus speed in addition to memory capacity. To increase the memory bandwidth and link speed, researchers are exploring the use of distributed, master-slave memory controllers which control a large aggregate capacity of memory devices [28, 42]. Two factors differentiate this approach from the traditional memory system design. First, only a master memory controller is integrated into a processor, and many other smaller slave memory controllers reside on a system motherboard off-chip. Second, the interconnect among memory devices is based on a fast, point-to-point, serial link technology rather than a conventional parallel bus interconnect. While the bandwidth and link speed of a memory subsystem can be improved with the techniques just described, increasing a memory device's density is not as easy as bandwidth or links. The DRAM technology for system memory faces the scaling wall of process technology in which a DRAM cell's feature size – the minimum transistor length – cannot shrink further to increase a DRAM chip density (and hence memory capacity) in a functionally correct and cost-effective way [35].

The choice of researchers to address the DRAM scaling problem largely falls into two approaches. Some researchers are investigating three dimensional (3D) die stacking technology as a promising means to increase a per-device memory capacity [75, 97]. For example, Hyrid Memory Cube (HMC) and High Bandwdith Memory (HBM) increase a per-chip memory capacity by stacking conventional DRAM dies [30, 53]. This can also improve the performance and dynamic power consumption of the memory chips because it shortens the signaling distance, which is enabled by the through-silicon via (TSV) technology [38, 87]. However, in addition to these advantages, this approach also has some negative effects which not only increase the vertical dimension of DRAM chips but also aggravate the thermal-induced problems caused by a heat dissipation in a vertical direction [81, 116]. Although the 3D stacked DRAM technology can increase the per-chip DRAM capacity, it does not fix the more fundamental problem. The problem unable to continually scale down a DRAM cell size remains unresolved until the DRAM technology itself is replaced by another. For that reason, other researchers are actively looking for new scalable memory technologies and they think that emerging NVRAM technologies are possible candidates for DRAM replacements or complements. Since a transistor feature size of these NVRAMs can be shrunk with an advancement in process technology, emerging NVRAMs can pack more bits in the same die area without increasing chip dimension. Moreover, it is also possible to apply 3D stacking technology to the NVRAMs in order to further increase the chip capacity. For example, Intel and Micron have recently announced that their 3D XPoint ("crosspoint") memory would be ten times denser than the conventional DRAM [52]. Hence, the approach which adopts emerging NVRAMs has been considered a better solution to solve the DRAM scaling problem.

Second, in the contemporary computing environment, the power consumption has been regarded as one of the most important design considerations in a computer system design, because the cost of electricity gradually becomes a major component of the total operating expense of modern computer systems [10, 26]. However, it is not surprising that a fraction of the power consumption of the DRAM memory and HDD storage to the overall system power is already significant. For instance, previous research showed that the DRAM main memory in a server system accounted for more than 30% of a total server power consumption [80, 66]. In addition, the power consumption of the DRAM memory, while a system is idle, is rapidly increasing and becoming comparable to the dynamic power consumption. For example, in a mobile system, the DRAM power consumption exceeds 30% of a total power consumption of a system in a stand-by mode. This is due to the fact that a DRAM cell must be periodically refreshed to prevent data loss caused by the leakage current [114]. Even worse, the power consumption by the wasteful DRAM refresh operations is projected to account for 50% of the total power consumption of a memory system with future 64Gb DDR4 DRAM devices [72, 93]. Meanwhile, the HDD storage system contributes to a system power consumption of an additional 20–30% [10, 58] and exacerbates the problem of overly high power dissipation in the memory hierarchy. As such, achieving power efficiency in the memory hierarchy is another crucial problem to address, and needs to be accomplished at both the memory and storage along the memory hierarchy. Though system architects will likely continue to harness and improve the existing architectural techniques in order to reduce the power consumption for DRAM memory and HDD storage, a large power consumption within the memory hierarchy is fundamentally attributed to the underlying technologies, DRAM and HDD, to build the memory and storage system. For instance, the HDD consumes a lot of power for performing the required mechanical operations such as spinning platters and moving actuators. Therefore, building the main memory and storage system with other energy efficient technologies is better solution to reduce the power consumption of the memory hierarchy. In response of the need, emerging NVRAM technologies have been explored for constructing an energy-efficient memory hierarchy.

Third, the performance gap between DRAM memory and HDD storage system has never decreased and is growing. Even if the HDD technology's capacity improvement rate has managed to keep up with the annual growth rate of data volume to store, the performance improvement of the technology is quite disappointing. For example, the HDD's capacity has increased by 40% annually, but an annual increase in the random I/O performance of the HDD drive is as low as 2% [29]. To address this problem, recent systems attempt to diminish the performance gap with solid-state storage device (SSD) drives that are built with NAND flash memory chips. Since the NAND flash memory is faster and more energy efficient than the magnetic disk media, some of the traditional HDD spans have gradually been superseded by SSD drives. In addition, an MLC (multi-level cell) NAND flash memory has surpassed the areal density of HDD [48, 79]. As a result, the SSD in current systems is gradually taking the role of fast swap devices, file caching devices, a whole HDD replacement, and main memory extension devices [9, 83]. However, the NAND flash memory cell has a very poor endurance limit as well as scaling issue. For example, an MLC NAND flash memory cell can tolerate at most about 3000 writes [110]. In addition, since there is still a large performance gap between main memory and storage even with NAND SSD, researchers believe that NAND flash memory will eventually be superseded by emerging NVRAM technologies because they provide even better operational characteristics than NAND flash for all aspects but cost per GB. As such, emerging NVRAM based SSD storage devices such as Intel's `Optane` are under prototype-level tests, awaiting mass production to enter the market [6, 34, 52].

In summary, DRAM and HDD have exposed their technological limits which cannot satisfy unwieldy demands of the current computing environment. To overcome the limits and meet the needs, the adoption of emerging NVRAMs into memory hierarchy is being examined carefully. Although DRAM and HDD will not suddenly disappear from the memory hierarchy, system architects are making preparations for featuring emerging NVRAM technologies in both the main memory and storage system in future computer platforms.

1.2 RESEARCH OVERVIEW AND CONTRIBUTIONS

In this section, we present an overview of our research problems and contributions. We first describe an emerging NVRAM-based system model which we have studied throughout this dissertation. Then, we explain two research problems which we have addressed for the given system model, and highlight our contributions. Lastly, we describe the organization of this dissertation.

1.2.1 An NVRAM-based System Platform

Before we describe the challenging problems to solve in this dissertation work, we outline the memory organization in a future NVRAM platform that we envision. According to a recent ITRS [35, 36] technology projection, a future system is expected to feature mature NVRAM technologies across the entire memory hierarchy by 2020. In particular, when the emerging NVRAM technologies become fully mature, their performance may not only be comparable to DRAM, but also accomplish a significant enhancement of per-chip density. For instance, current commercial PCM uses a single-level cell (SLC) technology which represents two states per bit (logic '0' and '1'), but it is anticipated to pack more than two logic states into a cell in the form of a multi-level cell (MLC) [7, 12, 55, 90]. When both main memory and secondary storage in a future platform are built with emerging NVRAMs, we foresee that the main memory will be built with SLC technology to provide higher performance, whereas the storage will be established by MLC technology to provide larger capacity.

When using PCM as a basic building block of the main memory, the system needs to handle an imbalanced read/write performance, and the limited lifetime of PCM cells. In particular, the 2× and 10× slower read and write latency than that of DRAM, and the limited write endurance around 10^6–10^8 are critical problems that hamper a wide adoption of PCM main memory. To properly manage the challenges, computer architects have proposed a main memory in which a small amount of DRAM is collocated with a large capacity PCM main memory either as an OS-invisible hardware cache or as an independently addressable memory region [15, 16, 68, 91, 92]. Following this reasonable design approach, our future

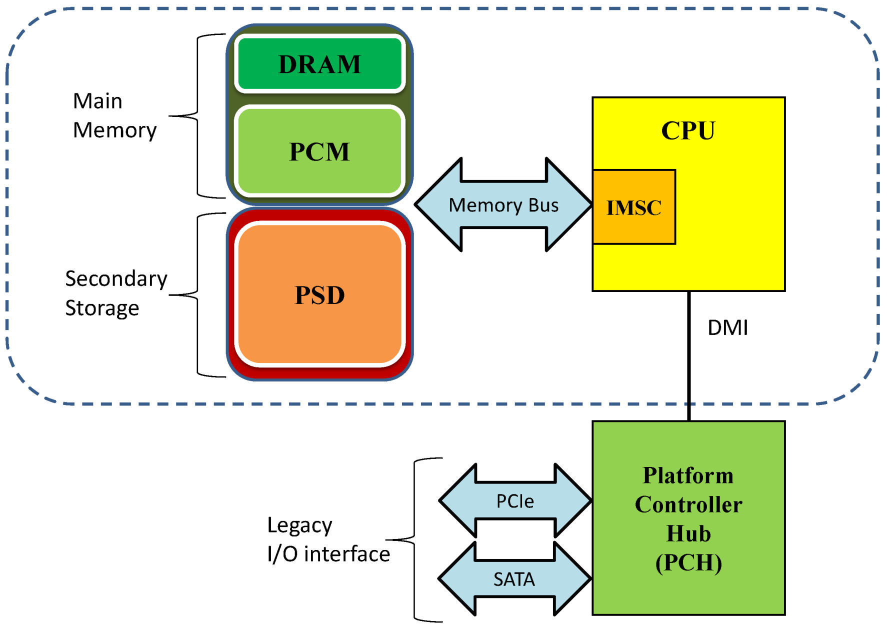


Figure 1: A future platform built with a DRAM/PCM hybrid memory and a PCM storage. The IMSC stands for an integrated memory and storage controller, while the PSD denotes an emerging **P**ersistent RAM based solid state **S**torage **D**evice.

platform assumes that a small amount of DRAM may be used to mitigate the drawbacks of NVRAM, forming the main memory with heterogeneous memories.

Figure 1 illustrates our envisioned future platform and its memory hierarchy. Different from a conventional system, the dotted box shows that the platform has a main memory comprised of DRAM and PCM, a PCM storage (called PSD, a persistent RAM based solid state storage device), and a memory bus interface to connect them to a processor. Even though the figure depicts the platform with a specific type of NVRAM technology, PCM, we also envision that other emerging NVRAM (e.g. 3D XPoint) may be adopted in lieu of PCM in the architecture. However, even in this case, we envision that the organization of the memory hierarchy from the figure will remain mostly unchanged.

Hybrid main memory, not DRAM cache: if DRAM assumes the role of a memory-level hardware cache by posing itself in front of PCM memory, this hierarchically organized memory is classified as the DRAM cache main memory. This organization has an advantage of making a hardware implementation simple and enabling the use of some familiar processor caching policies. However, it has several crucial shortcomings that may offset the benefits. For example, the given DRAM resources are invisible to the OS memory manager, possibly making their addressable space unnecessarily wasted. In addition, unlike the SRAM used for storing the processor cache's tags, the DRAM device is inappropriate for supporting fast cache tag lookup operations, which are on a critical path. On the contrary, a flat organization which equally treats DRAM and PCM can better and more flexibly utilize the given DRAM resources by exposing them to the OS for allocation and deallocation. For example, the OS may sometimes differentiate the usage of DRAM resources as a special high performing and write-absorbing addressable region, adapting to a system's needs. This flat organization is classified as a hybrid main memory. Since recent research showed that a hybrid main memory outperformed a hierarchical DRAM cache main memory [94, 16], the main memory in our future platform is organized as a hybrid main memory.

Storage connected to a fast, high bandwidth bus: Since the access latency of PSD is much faster than the traditional HDD or even NAND flash memory SSD, we believe that the PSD storage shall interface with the memory bus which is the fastest, highest bandwidth interconnect in a system platform. In this way, a system can exploit the full advantage of the unprecedentedly fast NVRAM based storage. As seen in Figure 1, the organization results in slotting storage into the same bus where the main memory resides. Even now, without a commodity PSD storage device, it is not uncommon to have a storage device on the memory bus in recent high-end server systems. For example, data center server platforms are gradually equipped with the NAND flash memory SSDs in a form of NVDIMM (Non-Volatile Dual-Inline Memory Module) [46, 21]. This enables the storage subsystem to achieve higher bandwidth and shorter latency than the conventional I/O interconnects by sharing the memory bus with DRAM main memory. In line with this trend, therefore, our studied future platform features the secondary storage, PSD, which is closely integrated with the main memory. Since the modern processor architecture does not have the north-bridge chip, the PSD-host interconnection may be one of the memory channels. However, note that the system may still include other conventional storage devices via the legacy I/O interconnect such as PCIe (Peripheral Component Interconnect Express) [18] or SATA (Serial ATA) [98].

1.2.2 Problem Overview

In this dissertation, we address two imminent challenges which a given future NVRAM system model will undergo while integrating NVRAMs into conventional system architectures. The first challenge is to mitigate a dramatic, sometimes intolerable degradation of the system performance under high memory pressure. The second challenge is to improve the utilization of the small, fast DRAM resources in a hybrid main memory and boost the applications performance via page swaps.

1.2.2.1 Effective Handling of Memory Pressure The memory organization in our NVRAM system from Figure 1 adheres to the traditional memory hierarchy of the main memory and storage which is stratified by media access latency. The choice allows system designers to embrace emerging NVRAM technologies without making radical changes to current system design methodologies. On the other hand, unless we make some architectural changes, it also means that the future platform inherits an inefficiency from a conventional system to handle high memory pressure, which may considerably damage the overall system performance [9, 99]. Since modern applications constantly call for larger system memory capacity to meet the required performance and throughput targets, it is evident that future applications will put far more demand onto memory subsystems. As a result, they have a high likelihood of suffering from a worse system performance due to the increased memory pressure. For this reason, it is critically important that a system designer improves the main memory performance of future computing systems.

Meanwhile, the problem of the memory pressure is also related to the storage system. When a conventional system experiences high memory pressure, it initiates the page reclamation by moving the contents of the infrequently referenced pages from the main memory to a swap space on the secondary storage. This process is called *demand paging* or *page swapping*, and produces free pages that can be used for new page allocations from applications. However, because of a large performance gap between the main memory (DRAM) and storage (HDD), frequent external page swaps significantly impair the overall system performance, slowing down the actual execution of user applications. While a system has to reduce the number of page swaps to storage in order to prevent system performance downfall, there is no other way to avoid the swapping under high memory pressure in a conventional system. Without timely disk swaps, the unmanaged memory pressure will result in a catastrophic situation in which the OS would arbitrarily kill one or more running user applications. Since each page swap requires to access the sluggish storage device, it is important to improve the swap process so that a system can not only ensure the persistent execution of applications (instead of being randomly killed) but also avoid an intolerable performance drop under heavy memory pressure.

In addition, it is important to understand the system-level implications of the future NVRAM-based memory hierarchy compared to a conventional system. Recent studies found that the system software latency to access files on an emerging NVRAM-based storage device accounts for larger portions of overall latency than the hardware access latency to an NVRAM device [19, 58]. For instance, a conventional OS I/O scheduler anticipatorily waits for the coming requests to merge with current I/O requests in order to reduce the number of expensive HDD accesses. However, the waiting delay may be longer than the access latency of emerging NVRAM-based storage. In case that the storage access is on a critical path, the conventional I/O access behavior worsens the overall system performance. Therefore, it is justifiable that an access to NVRAM-based storage device bypasses the I/O scheduler and is directly sent to the storage.

In summary, while keeping NVRAM-awareness in mind, it is imperative to study how we can further **improve traditional page swapping mechanisms between the main memory and NVRAM-based storage in order to prevent an excessive system performance degradation under high memory pressure**. This challenge is a problem of the interplay of the main memory and storage in a future NVRAM-based system, aiming at improving the overall system performance.

1.2.2.2 Efficient Utilization of Fast DRAM Resources As seen in our envisioned system model from Figure 1, system designers for a future platform cannot stay with a traditional DRAM-only approach to build a system memory due to a scaling wall and power inefficiency of DRAM technology. Instead, the system memory will be a hybrid main memory being comprised of DRAM and emerging NVRAM. As such, a future system needs to carefully orchestrate the heterogeneous characteristics of the two disparate memories by supplementing a shortcoming of one memory with a strength of the other, so that the system can achieve better performance. For example, even if emerging NVRAMs are scalable and are pushing the limits of increasing the system memory capacity, an emerging NVRAM such as PCM has an imbalanced, slower read/write performance in addition to the limited cell endurance than DRAM. Especially, the access latency slower than DRAM is the NVRAM's Achilles's heel that a hybrid main memory must hide from a system's critical path.

Meanwhile, as mentioned in Section 1.1, previous research [94, 102] have proposed a hybrid memory organization in which a small amount of DRAM is collocated with a large capacity of PCM main memory as a normal addressable region. However, even though this research showed that a hybrid main memory outperformed a DRAM cache organization by providing the system software/hardware with the flexibility of a choice of the memory type for a page allocation and placement, it is believed that there are still much room to further improve the application performance in the hybrid main memory through better page placement [15, 88, 94, 102]. Therefore, a momentous challenge of a DRAM/PCM hybrid main memory is to achieve the optimal page placement for boosting application performance at runtime. A system featuring a hybrid memory should be able to determine hot pages to be placed onto a small, fast DRAM region, and cold pages to be replaced onto a large, slow PCM region. If a system places frequently referenced pages onto the slower PCM region mistakenly, it may result in a tremendous degradation of the overall system performance due to the wrong page mapping to the memory type. Hence, it is imperative to investigate efficient mechanisms that not only accurately identify hot/cold pages, but also place only the hottest pages on the small, fast DRAM region throughout a program execution.

Unfortunately, recent research on a hybrid main memory have made an over-simplified assumption in devising their page placement/replacement strategies [92, 94, 115]. Specifically, they commonly overlook the overhead of page replacement between DRAM and PCM. As a result, the implications of the OS-managed page information on the hybrid main memory have not been studied well so far. Therefore, it is important to fill up the research hole by investigating a diversity of page swap schemes at the system level, and by evaluating the effect of the page swap on the performance of a hybrid main memory. In summary, the problem to address is to **improve the application performance when executing on a DRAM-NVRAM hybrid memory through better utilization of a small, fast DRAM region so as to approach the performance of running on a system with the DRAM-only main memory**.

1.2.2.3 How Are Both Problems Related To Each Other? The aforementioned two problems are closely related to each other in that they are both dealing with a page swap between two system components. While the first problem from Section 1.2.2.1 addresses the inefficiency of page swapping between the main memory and storage, the second problem from Section 1.2.2.2 deals with the page swapping between two disparate memories in the main memory. In addition, both problems shall be handled through the system-aware management rather than a conventional approach to tackle them at the corresponding levels of the memory hierarchy. Whenever it is necessary to differentiate the two types of page swaps, the page swap in a hybrid main memory is called as an in-memory page swap, and the one between the memory and storage as an external page swap.

1.2.3 Research Contributions

In this dissertation, we addressed the two problems described in Section 1.2.2. For each problem, the contributions of this dissertation can be summarized as follows:

1.2.3.1 Contributions to the External Page Swap Problem Most of this work was published in ICS 2013 [58] and CF 2011 [56].

1. We proposed and designed the Memorage architecture to improve the mechanism of handling high memory pressure in future NVRAM-based systems.
2. We implemented a prototype system of Memorage architecture in commodity Linux operating system, and evaluated it for application performance.
3. We developed an analytical model to investigate the effect of Memorage architecture on the lifetime of the main memory and storage system. We also showed that the model

was a useful tool to estimate the lifetime of a system or the part replacement period.

4. We developed a unique experimental methodology on a real platform to study future NVRAM-based systems, which was emulated with a commodity two socket server system. This compensated for the limitation of a simulation-only approach for studying emerging NVRAM-based systems.
5. We quantified and analyzed the OS software overhead to resolve high memory pressure through the demand-paging mechanism of state-of-the-art Linux OS. We confirmed that the OS software latency accounted for a large portion of disk swap latency.

1.2.3.2 Contributions to In-Memory Page Swap Problem These contributions were published in SBAC-PAD 2016 [59].

1. We developed an analytical model to investigate the effect of a page swap in a hybrid main memory on the application performance. Specifically, the model takes a queuing-theoretic, flow-control approach to improve page swap, and explicitly considers the profitability of a page swap, which distinguishes it from prior studies.
2. We compared the outcome of the model with the results from an existing architecture simulator [63] that was validated against a real hardware. Also, we showed that the model is useful to understand the effect of a page swap while analyzing the simulation results.
3. We proposed a model-guided, hardware-driven page swap mechanism, and evaluated its effectiveness by comparing with other page swap schemes. Furthermore, we showed how the model can be integrated into a micro-architecture.
4. We observed that OS-managed LRU lists is not able to identify hot and cold pages for page swap while running memory-intensive applications. This finding should guide other researchers not to blindly trust what appeared to be a conventional belief.

1.3 DISSERTATION ORGANIZATION

The remaining chapters of this dissertation are organized as follows. First, we present the background of this dissertation work in Chapter 2. In Chapter 3, we describe the design and implementation of the Memorage architecture which overcomes memory pressure by an integrated management of the main memory and storage NVRAM resources, and evaluate the performance improvement endowed by the architecture. Chapter 3 also gives analytical models to qualitatively analyze the impact of the Memorage architecture on the main memory and system lifetime, and show the usefulness of the developed models with realistic usage scenarios. Chapter 3 then summarizes related work. Chapter 4 explains an efficient hardware-driven page swap in a hybrid memory, presents an analytical model to investigate the effect of page swap on the application performance. In addition, Chapter 4 evaluates the usefulness of the model as well as a model-guided, hardware-driven page swap mechanism through extensive simulations. Then, Chapter 4 presents related work. Lastly, we summarize and conclude this dissertation, and discuss the future work in Chapter 5.

2.0 BACKGROUND

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 this fraction is growing [4]. Furthermore, according to the ITRS reports [35, 36], 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. In response to this problem, researchers have recently started exploring the use of emerging Non-Volatile RAM (NVRAM) as a DRAM replacement [118, 68, 92]. Another technological breakthrough is coming with emerging NVRAM technologies.

2.1 EMERGING NVRAM TECHNOLOGIES

Since DRAM and HDD technology have a looming challenge of performance, scalability, and power consumption, a handful of emerging NVRAM (also called storage class memory or SCM [37]) technologies are being actively developed by industry. Table 1 lists and compares three promising NVRAM technologies with DRAM, NAND flash and HDD technologies: PCM, STT-MRAM (spin-transfer-torque magnetoresistive RAM) and ReRAM (Resistive RAM) [95, 103, 111, 106]. Basically, NVRAMs' operations are based on sensing the resistance of a cell material rather than electrical charge. PCM is considered to be the closest (among all NVRAM technologies) to mass production, with commercial and sample chips available at high densities (1 to 8 Gbits) [37, 95, 3, 24].

NVRAMs have several desirable properties in common. Unlike DRAM, they are non-

	Latency			Program	Access	Non-	Write	Cell
	read	write	erase	energy	unit	volatile	endurance	density*
PCM	20ns	100ns	N/A	100 pJ	byte	Yes	$10^8 \sim 10^9$	$5F^2$
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	Yes	$10^4 \sim 10^6$	$4 \sim 5F^2$
HDD	8.5ms	9.5ms	N/A	N/A	sector	Yes	N/A	$2 \sim 3F^2$

Table 1: Comparison of emerging NVRAMs and existing memory/storage technologies [65]. *F is the minimum feature size of a given process technology.

volatile. Compared with NAND flash, they are byte-addressable and have faster speeds. Meanwhile, NVRAMs have known shortcomings; NVRAMs 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 hide and reduce write operations [23].

Meanwhile, ITRS [35] is anticipating multi-level cell (MLC) solutions of NVRAMs. Multi-level cell (MLC) designs effectively reduce the cost per bit by packing more bits per memory cell. While 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 NVRAMs may store more than two bits per cell [35, 27, 90, 22].

Introducing MLC NVRAM to a platform has several implications. First, MLC NVRAM will be slower than SLC NVRAM. Reportedly, MLC PCM read and write are 2 to 4 times slower [32, 91]. This is mainly due to more sensing levels for read and an iterative programming process for write [12]. The second implication is the lower write endurance of MLC NVRAM 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 NVRAM is more suitable for use in a NVRAM-based SSD than in main memory. If a NVRAM-based SSD employs MLC NVRAM and main memory uses SLC NVRAM, there is a latency gap between the two

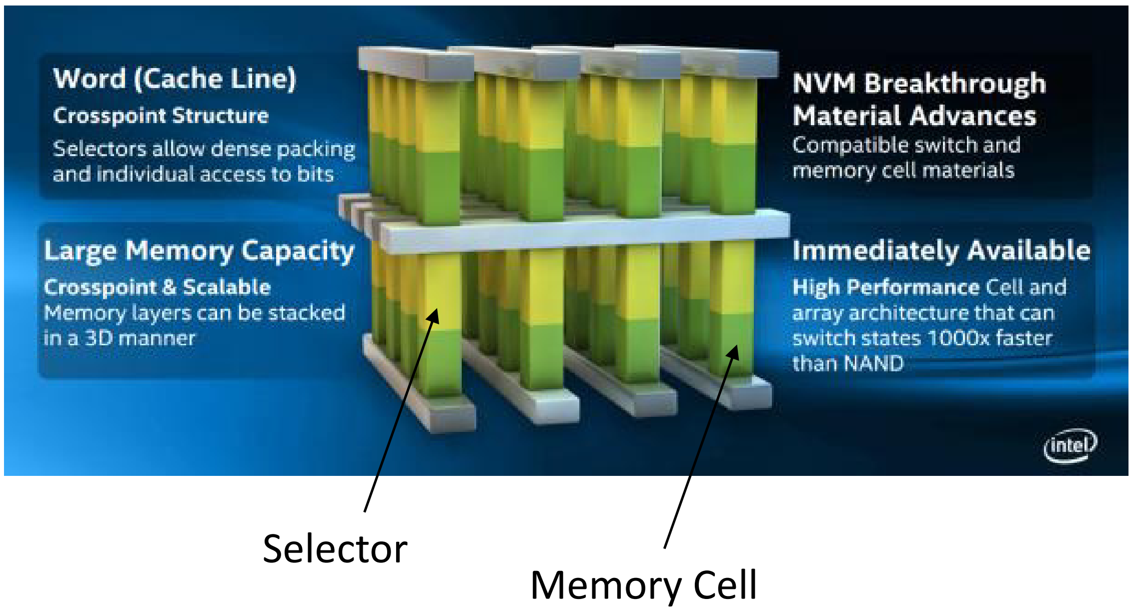

Word (Cache Line)
Crosspoint Structure
Selectors allow dense packing and individual access to bits
Large Memory Capacity
Crosspoint & Scalable
Memory layers can be stacked in a 3D manner
NVM Breakthrough Material Advances
Compatible switch and memory cell materials
Immediately Available
High Performance Cell and array architecture that can switch states 1000x faster than NAND
intel
Selector
Memory Cell

Figure 2: Intel and Micron's 3D XPoint Technology [52].

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 NVRAM, e.g., by treating an MLC PCM device like an SLC device when the system's capacity demand is low [32, 12, 91, 41].

Even if there are currently few details about its cell-level operational principle, a command set, and media characteristics, the most up-to-date progress of emerging NVRAMs has come from the announcement from Intel and Micron. Figure 2 illustrates a simplified cell structure of 3D XPoint memory [52]. It is reported that a 3D XPoint cell is three orders of magnitude faster and has three orders of magnitude longer lifetime than the current NAND flash memory. In addition, it is ten times denser than DRAM memory. All these information seems to nearly be in lined with the NVRAM technology projection.

2.2 SYSTEM HARDWARE INTERFACES TO NVRAM-BASED SSD

In today's commodity platforms for consumer and enterprise applications, HDDs are the slowest hardware component connected through either a relatively slow serial ATA (SATA) or serial SCSI (SAS) interface. The SATA [98] and SAS [49] are a communication protocol that moves data between host and storage devices and use ATA and ATAPI (ATA Packet Interface) command set and a SCSI command set, respectively. Both SATA and SAS replace with a serial point-to-point connection technology their predecessors, parallel ATA and parallel-attached SCSI which revealed the limitation of data bandwidth due to parallel wires' slow signaling and cross-talk, and as a result achieve orders of magnitude faster than prior interfaces for rotating HDDs. Even if the protocols continue to improve their performance and features, with SATA and SAS, a disk access request must pass through multiple chips and buses (front-side bus to host bus adapter to SATA/SAS controller) before it reaches the storage medium, incurring long latency. For example, the most recent SAS 3.0 protocol support at most 12Gbps interface speed and SATA 3.1 specification can achieve up to 6Gbps throughput performance.

Early NAND flash SSD offerings provide a direct migration path from HDDs based on the legacy interface like SATA and SAS. According to Gartner 2010 report, SAS protocol accounts for 37%, while SATA takes about 20% out of total SSD interfaces. However, both enterprise and consumer SSDs have quickly saturated the SATA bandwidth. Recent high-speed SSDs are attached via the higher-bandwidth PCI Express (PCIe) interface [39]. To further improve the performance, the industry consortium has defined and released NVM Express (NVMe) interface for PCIe interfaced SSDs [112]. The NVMe specification allows SSD vendors to standardize SSD's host interface and make it efficient through the register-level communication.

Likewise, it is reasonable to predict that PSDs will interface with faster buses than PCIe because NVRAMs 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 NVRAM-based SSD is lost. In addition, the memory bus is the highest-bandwidth bus of a platform. Hence, a NVRAM-based SSD will

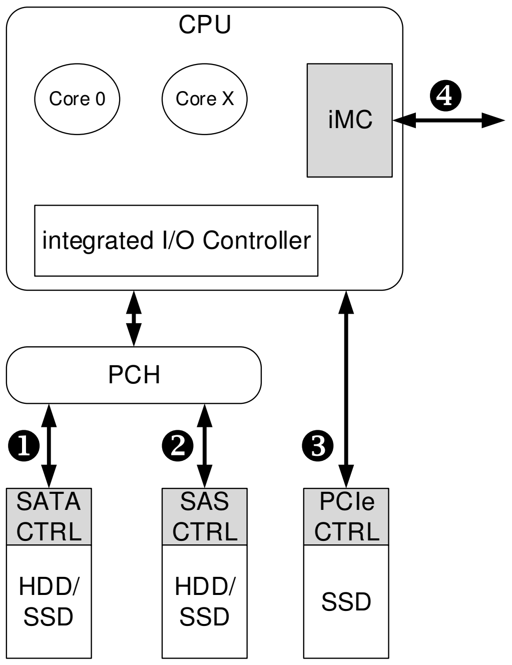


Figure 3: System hardware interfaces for NVRAM storage devices.

be most likely and best co-located with NVRAM memory modules on a platform's memory bus.

We expect that future platforms will continue supporting legacy PCIe 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 NVRAM-based SSD 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 3 displays possible hardware interfaces [1] to attach a storage device to current system platforms. Conventional storage systems are connected to host CPUs via Platform Controller Hub (PCH). Each interface bus is marked by a circled number in chronological order, and higher number also implies higher throughput performance.

[1]Even though we showed all storage interfaces in today's systems, note that we do not exclude the possibility that new interfaces between ❸ and ❹ may be defined. In this case, however, we classify them as variants of PCIe interface because their hardware capabilities are inferior to ❹.

2.3 OS MEMORY MANAGEMENT AND I/O STACK

Our descriptions in this section focus on Linux OS because of its relevance to our experimental study. However, other modern OSes also implement the key concepts explained here possibly with some modifications.

2.3.1 OS Memory Management

User applications request necessary memory resources via system calls such as *malloc()* or *mmap()* which are served by OS virtual memory manager (VMM). Basically, all physical memory resources (or pages) on a system are registered as free pages into buddy allocators after a kernel gets allocated essential memory resources during the boot process. Since the kernel classifies physical page frames into zones with different properties (eg. DMA zone) for it purpose, each zone has its own buddy allocator [17, 74, 77]. Then, the zoned buddy allocator takes a responsibility of de/allocating memory pages from its zone. However, if there is no free page to allocate under high memory pressure, OS VMM performs the page reclamation to make room for the new page. Since the new page allocation request causes a page fault, VMM moves to swap space anonymous pages which is not backed by file storage or simply de-allocate clean mapped pages backed by file storage in the process of page fault handling. The manager detects the status of scant free memory by three thresholds or watermarks, and recovers the level of a violated watermark through iterative page reclamation operations. At the end, the buddy allocator produces free pages for new allocations. Unfortunately, a page fault handling involved with page swap operations takes a long time to execute, and thus significantly degrades the system performance. This is due to the fact that it has to not only perform page table walking and manipulation, but also access slow storage device.

Meanwhile, Linux VMM manages the information of page reference recency, modification, and others in page tables. When a page reclamation and swap is required, VMM refers to this information in order to select less important pages for disk swap. Specifically, Linux VMM maintains two types of LRU lists, the active list and the inactive list, for anonymous and file

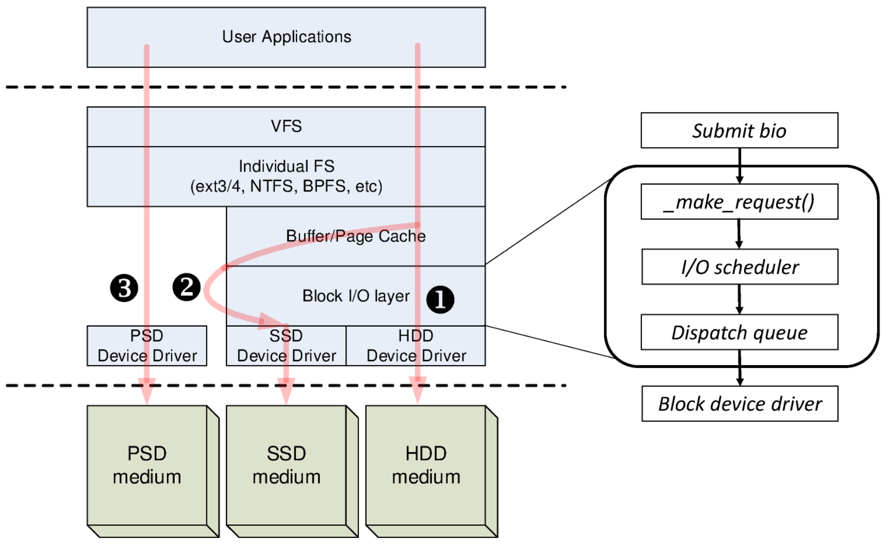


Figure 4: Linux I/O software stack. An I/O request from user applications in conventional systems goes through a full layer of software stacks before storage devices can service it.

pages, respectively. While anonymous pages are allocated for head, stack, and copy-on-write, file pages are for file-backed data such as binaries and data files. These lists keep track of hot and cold pages that have been recently referenced and not referenced, respectively [77]. Thus, VMM swaps out pages belonging to inactive page lists when `pageout` swap routine is called. However, if the inactive lists do not have enough number of pages to swap out, VMM first replenishes the lists by moving pages from active list to inactive list, then moves pages in inactive lists to a swap area as usual.

2.3.2 OS I/O Stack

Figure 4 displays I/O software stack in Linux. Since the traditional OS I/O software stack and file systems have been designed for rotating HDDs, storage optimizations are mainly devoted to reducing disk seek time through efficient I/O scheduling. For example, a read or write request in a conventional OS I/O stack 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 (❶). The Linux's default "completely fair queuing" (CFQ) scheduler [17] does this. By comparison, SSDs without mechanical rotation benefit little from such HDD-centric optimizations. 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 (❷). As a result, system designers have customized the OS I/O software stack with SSDs in mind and design new file systems [19, 54, 99]. This trend of retrofitting I/O stack can be also found even with PCIe PCM SSD [6]. Since PSDs are byte-addressable and even faster than SSDs, further changes to I/O stack can occur. For example, Linux's DAX (Direct Access) enabled file system bypasses OS file cache and directly accesses data on block I/O device, further reducing storage access latency [51] (❸).

2.4 OS PAGE MOVEMENT TECHNIQUES

There has been extensive research on page movement techniques in the context of a classic shared memory multi-processors (SMPs) [67, 86, 107]. The latency non-uniformity in SMPs is incurred by a CPU-to-memory affinity rather than memory media disparities like a hybrid memory. Due to a distance difference from CPUs to memory modules, a process running on a core in a CPU socket experiences different memory access latency from when the process is rescheduled to a core in another socket. To dynamically exploit the access latency difference in a memory system, a system can move a frequently accessed, distant page to a location closer to a core executing the program. This is very analogous to the need of page swap in a hybrid main memory.

A page movement in today's systems requires the OS to manipulate a page table in order to update the memory mapping information of pages in the movement. For this, the VMM first disables the write permission to the two pages to be swapped, whereas write access to other pages are not banned. Next, a (destination) page to accommodate an incoming page is allocated and mapped to an address. Once page data transfer completes, the original pages in the page movement will be unmapped, and released to a memory pool. Last, the write permission of the pages is reactivated. In current Linux implementation, VMM's `migrate_pages` and `move_pages` system calls can perform this page movement. The former

```
long migrate_pages (
        int pid,                              // process ID
        unsigned long maxnode,                // max number of new locations
        const unsigned long *old_nodes,       // old locations
        const unsigned long *new_nodes        // new locations
);

long move_pages (
        int pid,                  // process ID
        unsigned long count,      // the number of pages to move
        void **pages,             // array of pointers to pages to move
        const int *nodes,         // array of new locations
        int *status,              // array of move result status per page
        int flags                 // specifying type of page to move
);
```

Figure 5: Linux's page movement system calls.

is used for moving all pages associated with the specified process ID to a new location, while the later is used for moving the given list of pages to the specified new location. Figure 5 displays the APIs for these system calls. For this dissertation work, it is worthy of seeing what kind of information the OS expects for the software-based page movement.

Unfortunately, this OS-driven process may sometimes bring about a considerable overhead by frequent TLB shootdown [108] and cache line flushes which may offset the benefit of a page movement. In addition, not all requested page movements succeed, and any failed trial fruitlessly incurs extra overhead.

3.0 UNIFIED MANAGEMENT OF SYSTEM'S MEMORY AND STORAGE NVRAM RESOURCES

In this chapter, we present our noble observations on an envisioned NVRAM-based future system to motivate this work. Using them, then, we propose and evaluate Memorage architecture to better handle heavy memory pressure in the future system. Also, we examine the impact of Memorage architecture on the system-level lifetime via an analytical modeling.

3.1 MOTIVATION AND A STUDIED SYSTEM ARCHITECTURE

As described in Chapter 1 earlier, we envision in this study that a *future emerging NVRAM based systems will have to manage the NVRAM main memory and the NVRAM storage resources 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 device characteristic distinctions between main memory resources and storage resources**

Note that there are independent research and development efforts on scalable NVRAM main memory and fast "Persistent RAM solid-state Storage Devices" (or "PSDs") using high-density NVRAM chips. If scaling predictions of ITRS [35] 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 building 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 [96].

In addition, fully exploiting the high bandwidth of the NVRAM 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 [62]. Suppose that both main memory and system storage are comprised of NVRAM 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 software overhead out of an overall I/O service latency 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 [17]. If the current software stack is unchanged, however, the major difference in data access latency between NVRAM 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 [68, 65, 19, 27]! 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 [47, 64]. While there are different storage usage scenarios such as PC, embedded and server environments,

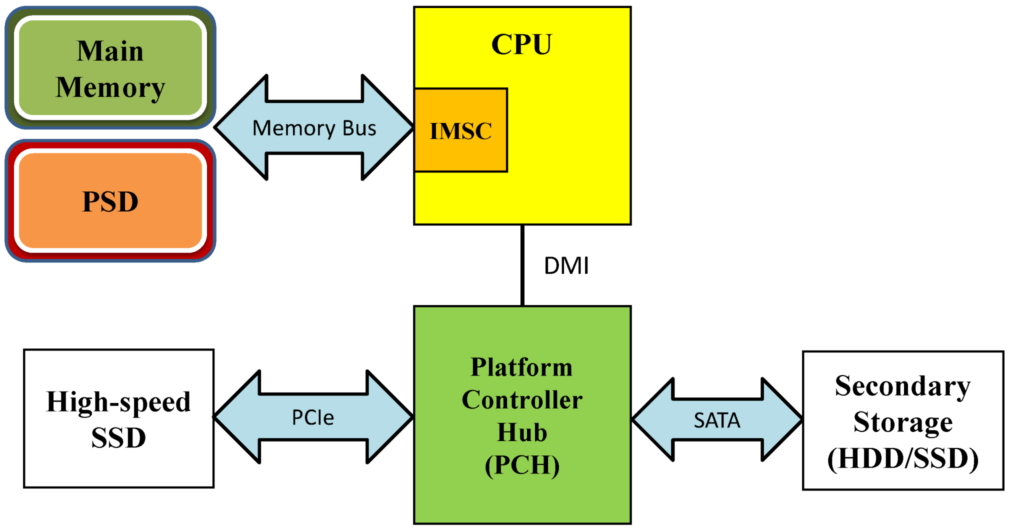


Figure 6: NVRAM main memory and a PSD share the memory bus. IMSC represents Integrated Memory and Storage Controller. Note that not all storage devices in this figure have to be present simultaneously.

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, the lost resources.

Based on the above observations, we make a case for *Memorage*, a system architecture that utilizes the system-wide NVRAM resources in an integrated manner in order to efficiently mitigate the performance degradation of a system under high memory pressure. Figure 6 illustrates a studied future system platform, which features NVRAM-only [1] main memory and PSD storage device on a memory bus.

[1]The main memory may be a hybrid memory in the near future. However, Memorage with NVRAM-only main memory foresees the long-term research, when NVRAM technology becomes mature and achieves the performance comparable to DRAM.

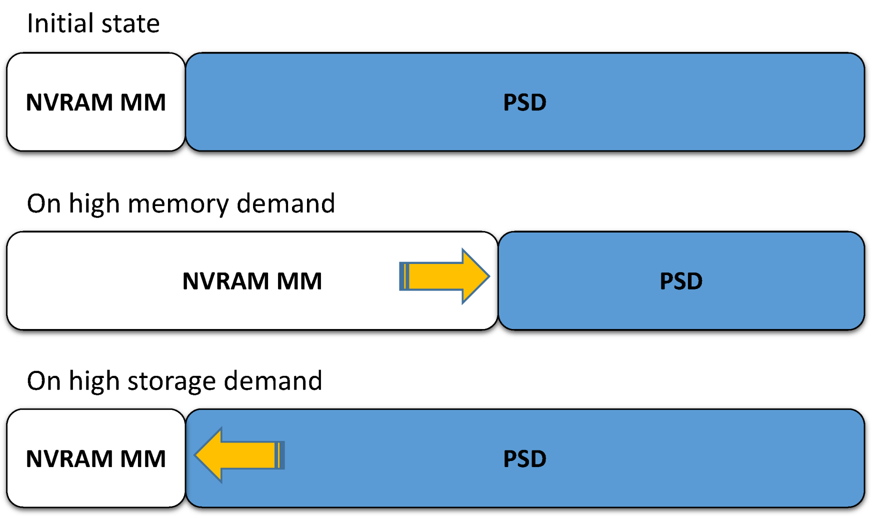


Figure 7: Illustration of Memorage concept. Memorage dynamically expands or shrinks the capacity of main memory (denoted "MM") and PSD on demand.

3.2 THE MEMORAGE ARCHITECTURE

In this section, we propose Memorage architecture and explain its underlying principles and design goals. In addition, we detail the design and implementation of the proposed Memorage architecture which can effectively be integrated into a conventional memory hierarchy.

3.2.1 Memorage Philosophy

Memorage tackles the inefficiency of NVRAM resource utilization by collapsing the traditional static boundary between main memory and storage resources (see Figure 7). The Memorage approach is motivated by the fact that fast NVRAM 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 NVRAM's limited endurance with a global wear-leveling strategy that involves all available NVRAM resources.

Storage capacity in a drive has increased with the improvement in storage density. Studies like Meyer et al. [82] 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 60% full. These studies clearly suggest that a storage device in a system is likely to have *substantial unused space* during its lifetime.

The Memorage architecture aims to effectively address the above wastefulness by suggesting the following 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 NVRAM 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 NVRAM 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 (principles 2), lifetime becomes more manageable because the write traffic to the main memory and the storage can be re-distributed with software control.

3.2.2 Key Design Goals

In this subsection, we discuss three design 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 NVRAM 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 infrastructures whenever possible. In Section 3.3.1, we 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 NVRAM 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 NVRAM resources are exchanged under memory pressure situations.

3.2.3 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 explained in Section 3.3.1. 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 today's 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 3.4. That said, incorporating the first Memorage principle requires two major changes in an OS. First, it requires managing the status of NVRAM resources in both memory and storage together. Second, it calls for developing a strategy to dynamically expand and shrink the main memory capacity. We subsequently expatiate our design approaches to accommodate the two changes.

3.2.3.1 Managing Resource Information As we discussed in Section 3.2.2, the Memorage prototype extensively reuses existing memory management infrastructures of Linux in order to reduce the development efforts. 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 [17] for more details).

To acquire the status information of resources in the Memorage zone, virtual memory manager works closely with the PSD device driver and the file system. The PSD device driver builds on a custom ramdisk driver to emulate the PSD with the system DRAM memory, and can perform both block I/O operation and page-level allocation from the designated node resources. In addition, it supports popular file systems such as ext3/ext4, and POSIX file access APIs as a traditional storage block driver. 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 PSD storage.

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 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 NVRAM resources from the PSD under memory pressure. Our prototype PSD device driver carries this out by hot-removing PSD region (which is a memory node) during its initialization. However, the offlined PSD region is logically removed from VM manager rather than physically from the system.

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 8 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:

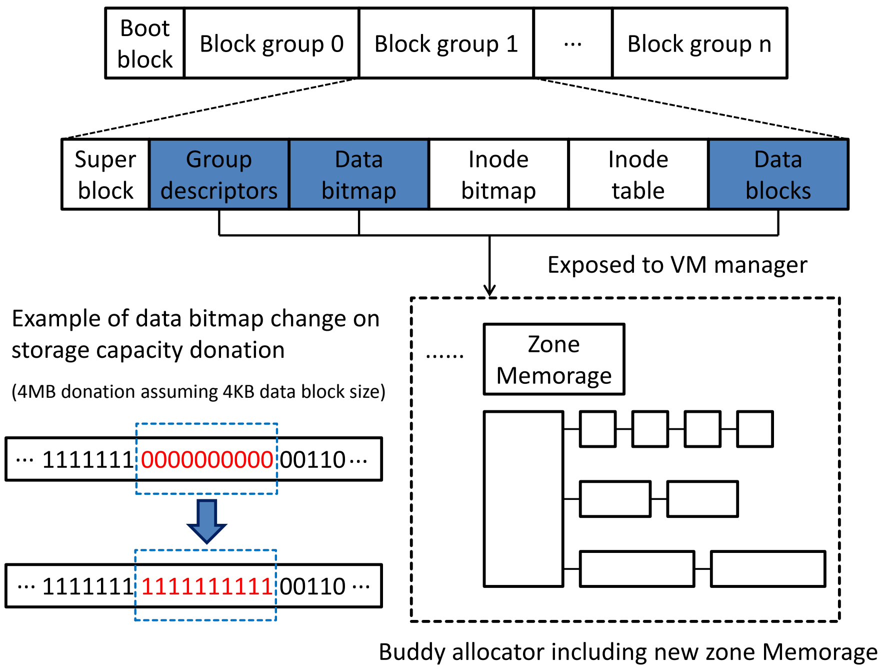


Figure 8: Memorage exposes free data blocks from a (mounted) file system to VM manager as a new zone. Buddy allocator treats the new Memorage zone the same way as other zones.

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 operating systems 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 a normal system shutdown request, Memorage mandates the file system to nullify the bitmap information previously marked for PSD donations because we assume memory 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, letting it invalidate the inconsistent bitmap information rather than trying to fix them up and slowing down the boot process.

3.2.3.2 Memory Expansion and Shrinkage When a system has few 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 that interferes normal storage access operation with the lock contention on common data structures.

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* (ie. *pages_high*, *pages_low*, and *pages_min*) used to control the invocation and sleeping of *kswapd*, the kernel swap daemon [17]. 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 9 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

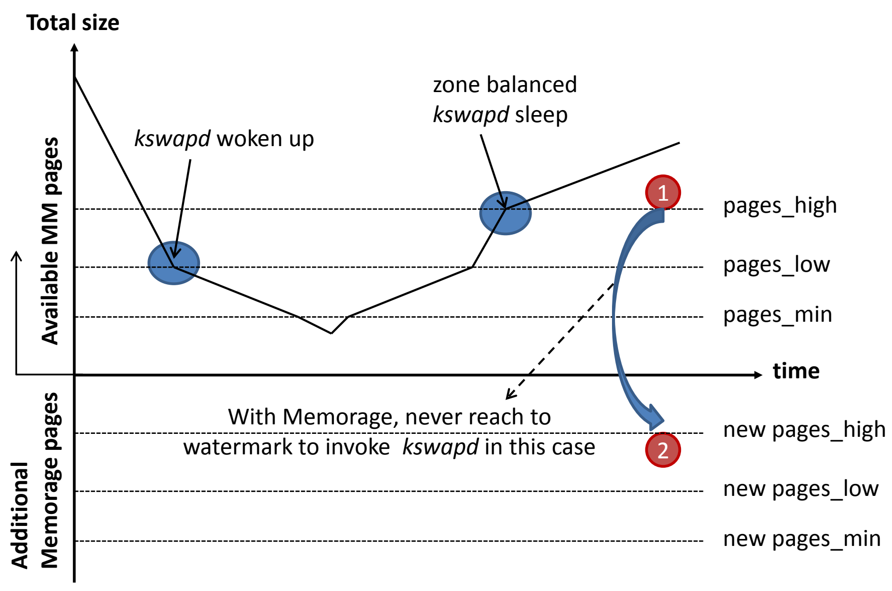


Figure 9: Memorage expands the margin to tolerate memory shortage, thus eludes a costly page swapping by the increased margin.

two ways. First, it adds a new watermark (WMARK_MEMORAGE) between *page_high* and *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 10 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 resource controller 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 excess PSD NVRAM resources (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

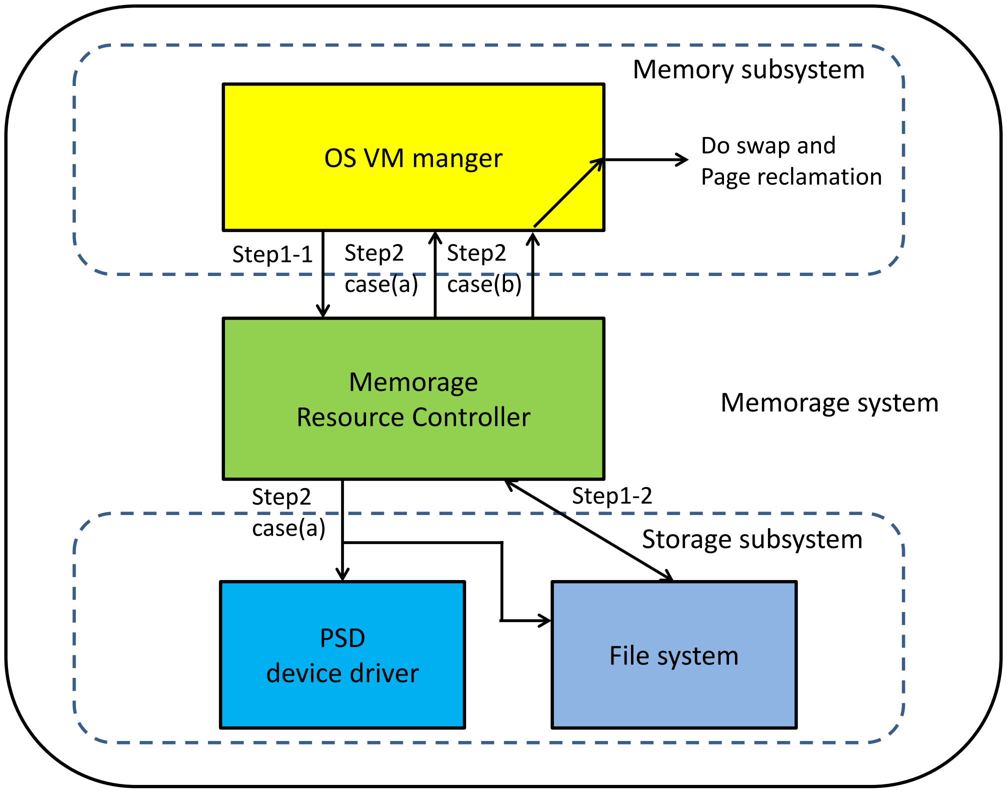


Figure 10: Cooperation between OS VM manager and storage subsystem to implement Memorage's flexible capacity sharing.

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 16 on page 47). 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 [44].

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.

3.2.4 Comparison with Possible Alternatives

Memorage provides a future NVRAM 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 NVRAM resources.

First, a platform may feature only fast SLC NVRAMs 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 NVRAMs which can store more information per cell than SLC. Moreover, when to control the growth of a particular space, especially the storage, is vague. By comparison, Memorage honors the traditional notion of the main memory and storage capacity, but manages the underlying NVRAM resources such that main memory lifetime issues influenced by NVRAM cell's wear-out as well as overheads for swapping 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 NVRAM 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, whereas Memorage can adapt to these changes.

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 may not achieve raw NVRAM performance of the PSD due to unnecessary file operations unrelated to page resource allocation.

3.2.5 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 are migratable. For example, direct mapped kernel pages are not. Therefore, it is important 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, 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 NVRAM, e.g., Wu et al. [113] 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 NVRAM main memory and the PSD. For example, if we treat all the NVRAM resources equally, 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 3.4 examines the effect of lifetime ratio of main memory to PSD.

3.3 EXPERIMENTAL RESULTS

In this section, we explain our evaluation methodology and experimental platform, and present the characterization results of our emulator and OS software overhead. Also, we give the performance evaluation results achieved by our Memorage prototype.

3.3.1 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 system implemented on a real platform. 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 running Linux kernel 2.6.35.2 with the `ext3` file system features an Intel Core i7 quad processor operating at 2.67 GHz, 8 MB L3 Cache, and 9 GB DDR3-1066 DRAM. Our Memorage prototype runs on a dual-socket Intel Xeon-based server platform and a newer 3.2 kernel. This platform has a large memory capacity organized in NUMA and eases emulating the PSD capacity. We perform application performance

System Component	Specification
Platform	HP Proliant SL160g6 Server
CPU	Two Intel Xeon E5620 (Sandy-bridge) processors 2.4 GHz base frequency, 2.66 GHz max turbo frequency 4 cores per processor
Hyper-threading	8 hardware threads/processor
Last-Level Cache (L3)	12 MB
Number of QPIs	2 links
Bus speed	5.86 GT/s QPI
Memory System	3 memory channels 192 GB DDR3-1066 SDRAM populated as 12 × 16 GB RDIMMs
Operating System	Linux kernel 3.2.8
File System	Ext3

Table 2: Experimental platform.

evaluation on this platform. Table 2 summarizes the platform's specification.

Figure 11 illustrates how we emulate a future NVRAM-based system with a commodity server. Figure 11a shows an initial status of two socket system which has two CPUs connected to memory of socket0 and socket1, respectively. Socket0 memory emulates NVRAM main memory of 4.4 GB capacity, and socket1 memory of 96 GB capacity emulates PSD. First, we offline CPU1 as shown in Figure 11b, thus the system has only one CPU. Then, we offline PSD resources and hide them from OS virtual memory manager (see Figure 11c). Therefore, OS memory manager cannot see PSD resources when it tries to allocate a memory page. When memory pressure occurs (Figure 11d), PSD resources are exposed to OS memory manager via memory hot-plugging operation (Figure 11e). Figure 11f depicts a system after memory pressure disappeared.

To form our workload, we select eight benchmarks from the SPEC CPU2006 suite because

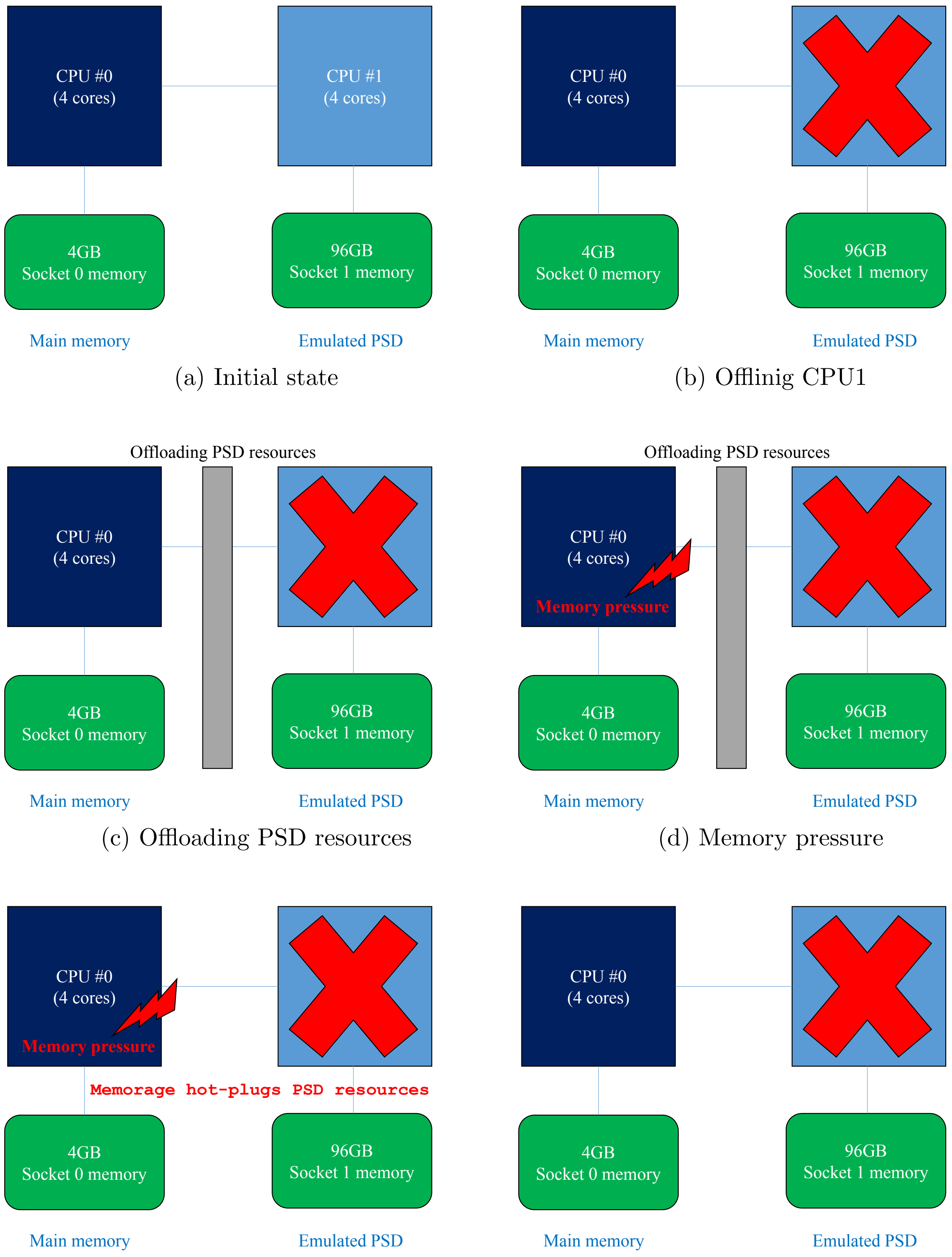


Figure 11: Illustration of emulation methodology.

they are memory-bound applications [91]. 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, 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* [14], 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 NVRAM whereas the PSD uses MLC NVRAM. This assumption implies that the target system is built cost-effectively—fast main memory with small capacity and slightly slow storage with large capacity. As discussed in Section 2.1, this construction does not necessarily imply much longer access latency to the PSD-provided memory capacity because MLC NVRAM can be read and programmed like SLC NVRAM. We assume that the NVRAM 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 NVRAM main memory and the NVRAM 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 [2]. 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 noise, we repeat experiments three times and average the

[2]We do not re-launch applications which complete earlier than others so that we can estimate a lower-bound of the performance gain from Memorage. However, re-launching early completed applications sustains memory pressure for a longer time, hence Memorage will achieve greater performance improvement.

Configuration	Description
Baseline	A system has 4.4 GB effective memory available for allocation. This size causes significant memory shortage.
Memorage	In addition to 4.4 GB (Baseline), Memorage provide a system with an additional 2 GB capacity from PSD 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 for page allocation.

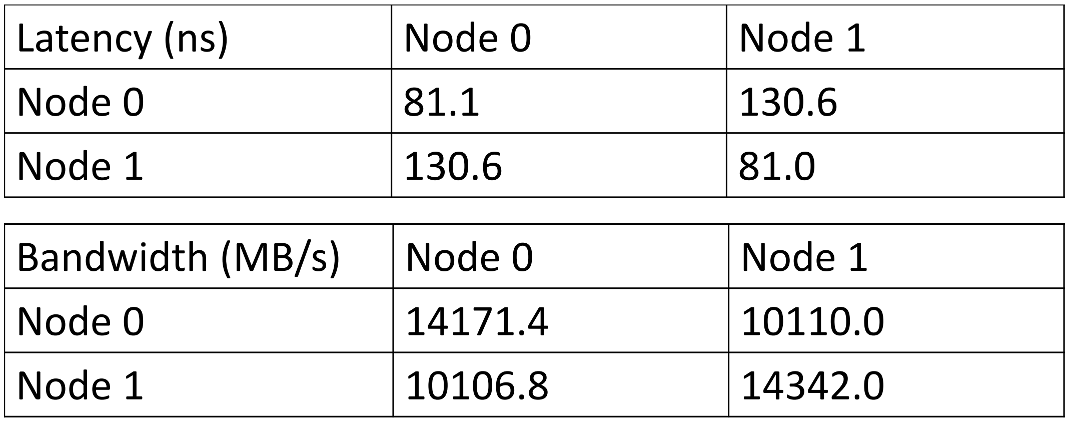
results. We also reboot the system after each experiment to keep the system state fresh and ensure that our workload runs under as much identical system conditions as possible.

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

3.3.2 Characterization of Platform Performance

We characterize an average memory access latency and memory bandwidth of our platform while consecutively issuing read operations. This bare-metal performance is important to understand how accurately our emulation vehicle based on a commodity server can emulate characteristics of the NVRAM main memory and PSD storage hardwares.

Figure 12 displays the memory system performance in terms of average latency and bandwidth measured when the system is otherwise idle. 2 CPU sockets in the platform are denoted by node 0 and node 1, respectively. In this matrix table, values on an intersection of an identical node number show local memory access latency and bandwidth of the node, whereas values between different node numbers represent the performance of remote



Latency (ns)	Node 0	Node 1
Node 0	81.1	130.6
Node 1	130.6	81.0

Bandwidth (MB/s)	Node 0	Node 1
Node 0	14171.4	10110.0
Node 1	10106.8	14342.0

Figure 12: Average memory read latency and bandwidth of the unloaded experimental platform. A node is a logical unit which OS manages CPU and memory resources associated with the CPU. Thus, the value on a cell crossed by the same node number means that the CPU in the node number accesses the memory in that node number.

memory access latency and bandwidth. The figure indicates that the emulated NVRAM main memory's access latency is faster than that of PSD by 61.2%. Also, the bandwidth of NVRAM main memory is higher than PSD by 41%. We presume that this performance difference reasonably emulates the performance disparity between the memory module and storage device built with SLC and MLC NVRAM technologies, respectively.

By contrast, Figure 13 shows the memory system performance measured when the system is busy. The x-axis shows an inject delay in cycles, which a load request is delayed by other application's memory requests. As system loads increase, the bandwidth degrades by about 30×, and average memory access latency decreases by 44%. The average memory latency is calculated from the bandwidth value measured over a constant time duration. As the load increases, the amount of data that a concerned thread can access decreases, and as a result, the average memory access latency decreases [3]. These measurements were performed by Intel's Memory Latency Checker tool [50] that is based on various hardware performance counters in a processor.

[3]See the manual from [50] for more details on the tool's capability.

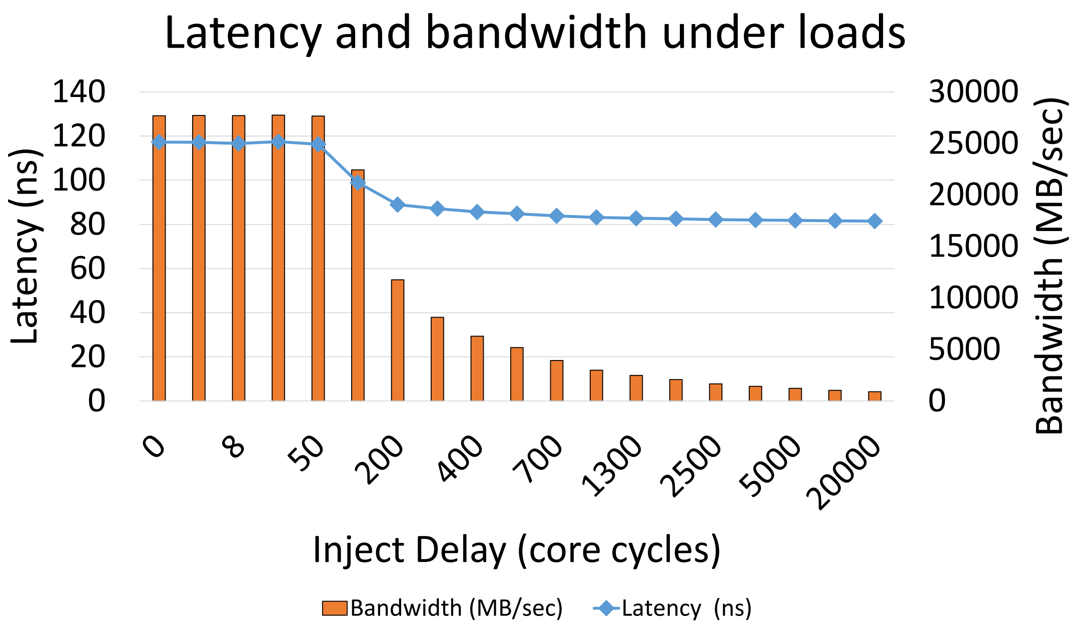


Figure 13: Average memory latency and bandwidth of the loaded experimental platform.

3.3.3 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 swap cache that contains the pages shared by several processes. 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 14 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.

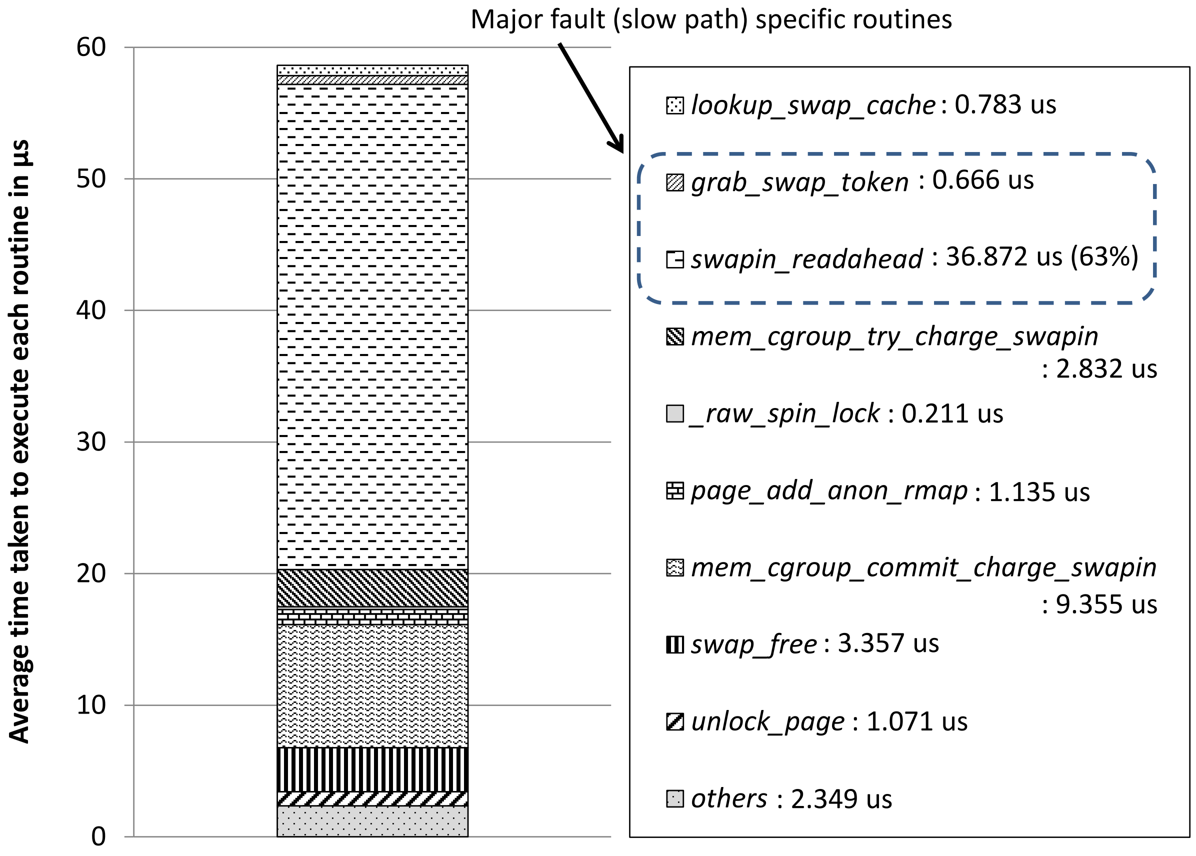


Figure 14: Breakdown of software latency of the Linux page fault handler. Routines in the legend are invoked in order. Recall that Memorage will eliminate this latency.

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 track the memory usage and limit of the specific group of processes, and reduce the chances of swap thrashing during page reclamations. The observed overhead is caused by the activities that VM manager updates the necessary page accounting metadata such as LRU data structure.

After all, given the long latency required on each page fault, a memory-bound appli-

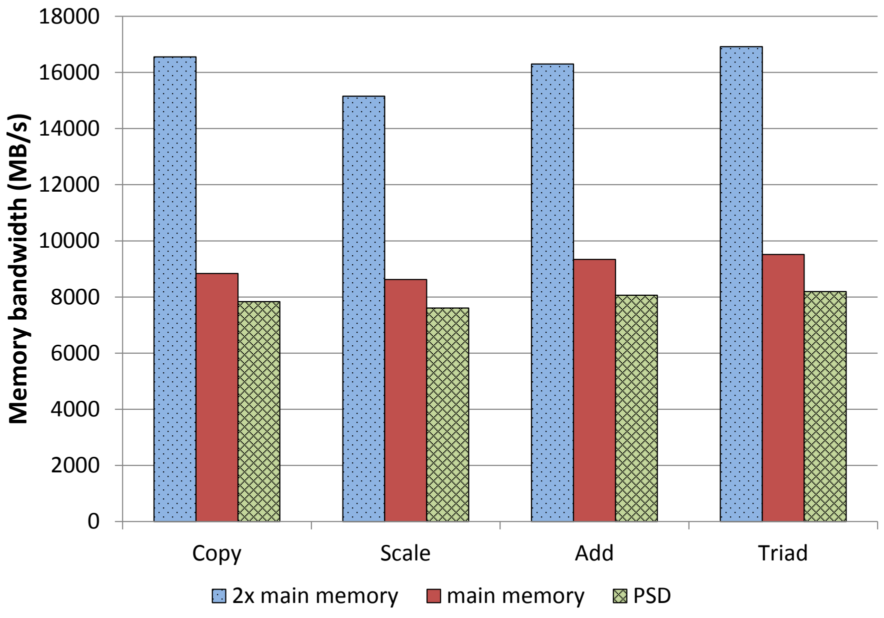


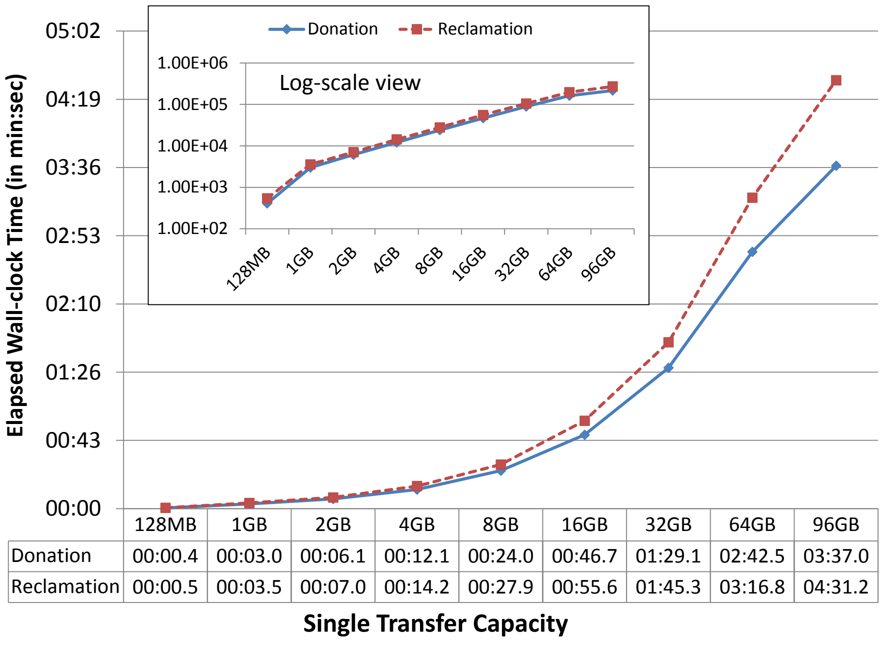
Figure 15: Performance difference between main memory and PSD on our experimental NUMA platform in terms of the peak bandwidth, running the STREAM benchmark. The “2x main memory” represents the bandwidth performance when both sockets are main memory.

cation’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. The larger the effective memory capacity is, the less frequently page faults would occur.

3.3.4 Application Performance

The result of the previous Section 3.3.3 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 [78]. This allows



	128MB	1GB	2GB	4GB	8GB	16GB	32GB	64GB	96GB
Donation	00:00.4	00:03.0	00:06.1	00:12.1	00:24.0	00:46.7	01:29.1	02:42.5	03:37.0
Reclamation	00:00.5	00:03.5	00:07.0	00:14.2	00:27.9	00:55.6	01:45.3	03:16.8	04:31.2

Figure 16: The latencies (in form of min:sec.ms) for Memorage to offer and retrieve PSD resources as the transfer size changes.

us to quantify the "application-level" performance difference between main memory (on the local node) and the emulated PSD (on the remote node). This is different from the bare-metal measurements presented in Section 3.3.2 in that the STREAM benchmark not only has both read and write operations, but also considers some compute time between two memory requests.

Figure 15 shows that main memory achieves higher bandwidth than the PSD by about 15% on average, and half bandwidth of 2x main memory that both sockets are main memory. We consider this difference to be a reasonable artifact of our experimental environment, as reading from and writing to MLC NVRAMs (even if they are operating in the SLC mode) could take slightly longer than SLC NVRAMs.

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

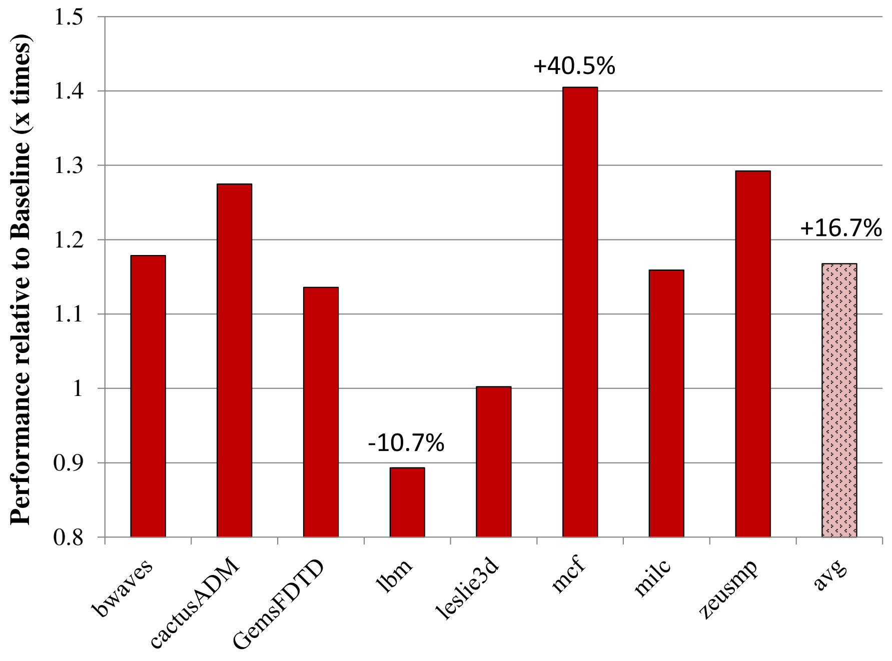


Figure 17: Relative performance of benchmarks with Memorage (based on wall clock time measurement). Performance improvement can be identified with a value greater than 1.

and the PSD, not on each (minor) page fault. Figure 16 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 17 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. The total execution time of Memorage are improved 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.

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

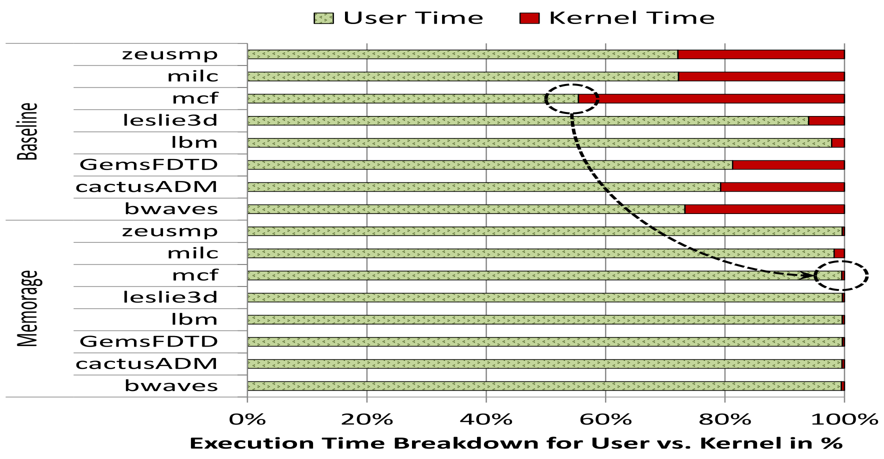
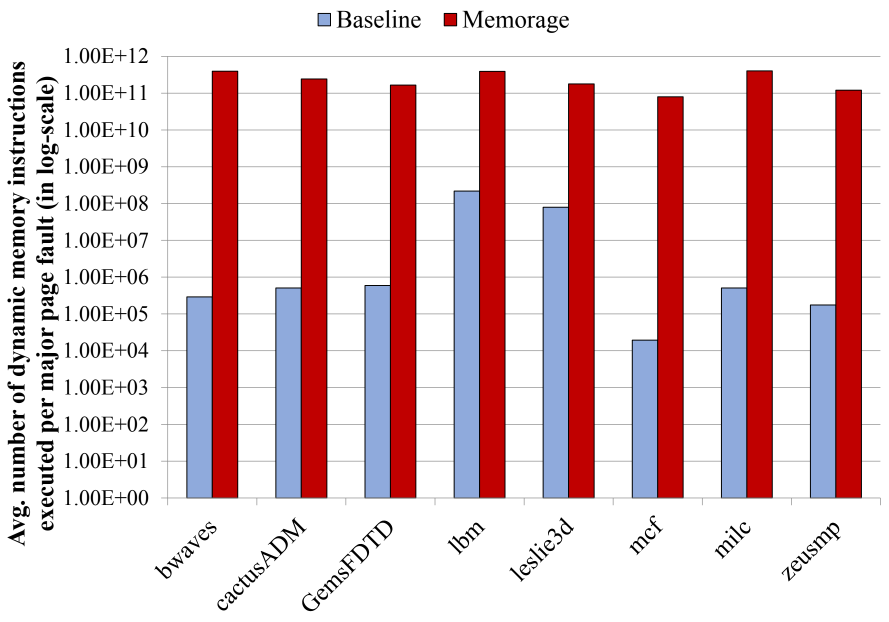


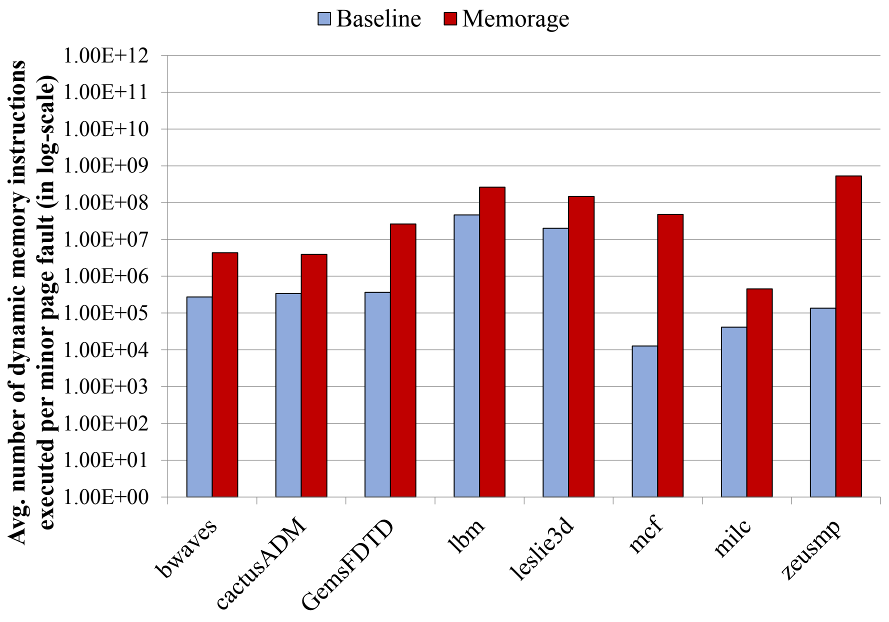
Figure 18: Total execution time is normalized to highlight the relative time spent in user applications and system routines.

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 can complete to execute faster. We also find that slight memory shortage (e.g. several dozens of MBs) is successfully handled by *kswapd*, whereas the huge 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 18 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.



(a) Average number of memory instructions per major fault



(b) Average number of memory instructions per minor fault

Figure 19: Memorage reduces the impact of major and minor page faults by increasing the average number of memory instructions between two faults. A larger value in the plot means more memory references are made without a page fault.

Finally, Figure 19 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 operating systems 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 due to a write on a shared page. Minor page faults have smaller impact on system performance compared to major faults because handling them merely require to create new page mappings without accessing storage. Hence, the performance of a system with Memorage is relatively insensitive to the number of minor 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 avoid them.

3.4 SYSTEM LIFETIME MODEL OF NVRAM RESOURCE SHARING

In this chapter, we develop a model to assess the system-level lifetime improvement of NVRAM resources by the proposed Memorage architecture.

3.4.1 Modeling 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 NVRAM 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 NVRAM main memory and PSD in the conventional system, respectively. They represent the time taken until all NVRAM 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 NVRAM 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 of the main memory. Ideally, Memorage could expose the whole NVRAM resource capacity to global wear-leveling because it manages all NVRAM 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 NVRAM 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:

$$\begin{aligned}
L_{new} &= \frac{E_m \cdot C_m + E_s \cdot C_s}{B_m + B_s} \\
&= \frac{E_m \cdot (C_m + \frac{\alpha}{\gamma} \cdot C_m)}{B_m + \frac{1}{\beta} \cdot B_m} \\
&= \frac{E_m \cdot C_m \cdot (1 + \frac{\alpha}{\gamma}) \cdot \beta}{B_m \cdot (1 + \beta)} \\
&= L_m \cdot \frac{(1 + \frac{\alpha}{\gamma}) \cdot \beta}{(1 + \beta)} \qquad (3.1)
\end{aligned}$$

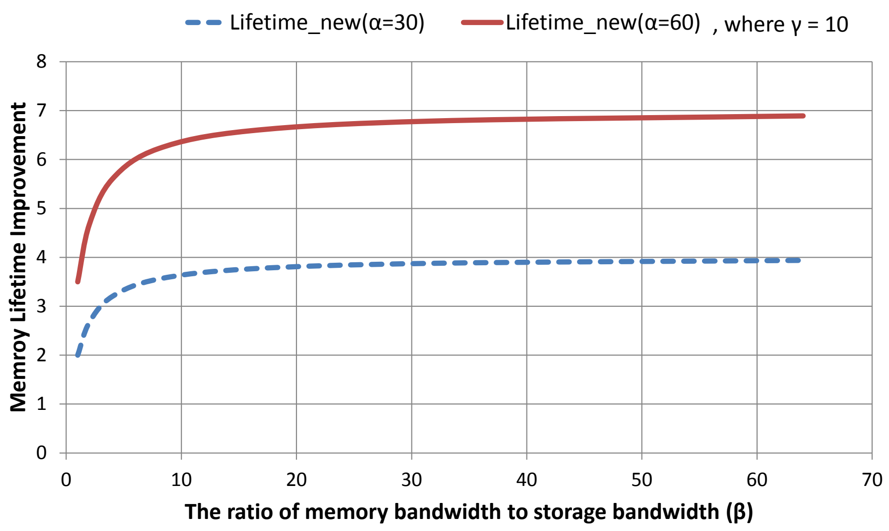


Figure 20: Main memory lifetime improvement.

Equation (3.1) 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 NVRAM 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 20 illustrates how the lifetime improvement of NVRAM 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) [92, 2], and hence, we expect that the large main memory lifetime improvement of Memorage will be effectively achieved.

3.4.2 System Lifetime Analysis via Model

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.

$$\begin{aligned} L'_m &= (D_m \cdot C_m)/(B_m \cdot (C_m - \eta)), \\ L'_s &= D_s/(B_s + \frac{\eta}{C_m} \cdot B_m + \frac{2\eta}{\tau}) \end{aligned} \tag{3.2}$$

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 expressed by Equation 3.3.

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

$$\begin{aligned} L'_m/L_m &= \frac{1}{(1-h)}, \\ L_s/L'_s &= 1 + \beta \cdot h + \frac{2h \cdot C_m}{\tau \cdot B_s} \end{aligned} \tag{3.3}$$

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 write bandwidth difference between main memory and PSD storage).

Figure 21 plots Equation (3.3). 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 $2\times$ 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.

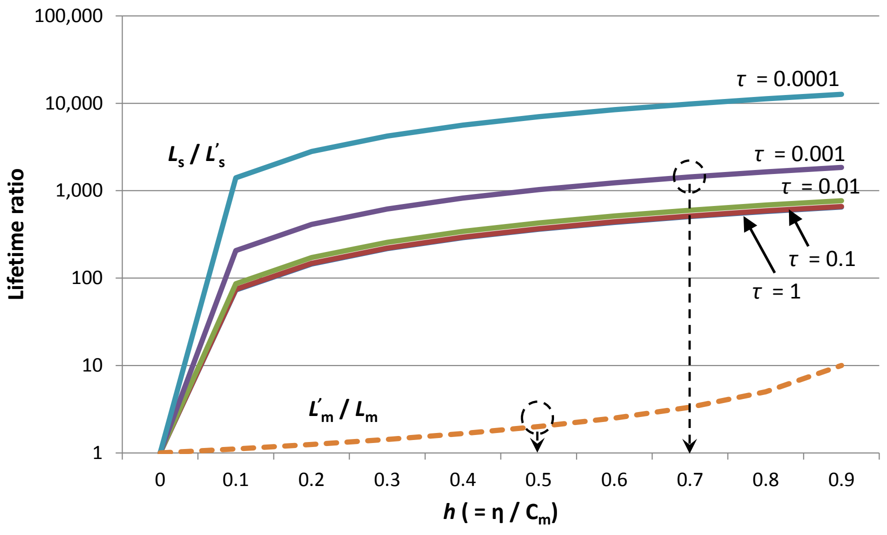


Figure 21: Main memory lifetime improvement and PSD lifetime degradation as a function of h (a ratio of the capacity borrowed from PSD to memory capacity) and τ (a time period for main memory to own the borrowed capacity. more frequent transfer with smaller τ).

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 NVRAM 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.

3.5 RELATED WORK

The work by Freitas et al. [37] gives an excellent overview of the NVRAM technologies. They suggest that a NVRAM is a "universal memory," providing capacity to both main memory and storage of a platform. They also suggest that the 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 [118, 68, 92]. 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, Jung et al. [57], Caulfield et al. [19, 20] and Akel et al. [6] evaluated the potential of NVRAM as a storage medium. They studied the performance of a block storage device that employs NVRAM 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 NVRAM.

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 [54, 99, 100]. 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 NVRAM technology.

Badam et al. [9] 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. [91] and Dong et al. [32] 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 [109] 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. [44] 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 NVRAM. A Memorage system collapses the traditional main memory and the storage resource management and efficiently utilizes the available NVRAM resources of the entire system to realize higher performance and longer lifetime of the MVRAM resources.

3.6 SUMMARY

Emerging NVRAM technologies have the potential to find their place in a computer system and replace DRAM and (a portion of) traditional rotating storage medium. This work 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 NVRAM. 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 NVRAM resource exchange between main memory and NVRAM storage is shown to improve the lifetime of the NVRAM main memory (by up to 6.9 $\times$) 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.

4.0 HARDWARE-DRIVEN PAGE SWAP IN HYBRID MAIN MEMORY

In this chapter, we develop an analytical model to assess the profitability of in-memory page swap between DRAM and NVRAM in a hybrid main memory subsystem, and evaluate the effectiveness of a hardware-driven page swap mechanism which is guided by that model.

4.1 MOTIVATION

Over the last three decades, DRAM has been a de-facto standard memory technology to build system's main memory. However, as semi-conductor process technology continuously advances at the pace of *Moore's law* [101], DRAM is approaching its scaling wall due to manufacturing constraints which cannot further shrink a DRAM cell in a functionally correct and cost-effective way [61]. Moreover, current leakages are accelerated with a smaller transistor feature size, thus data retention time decreases. As a result, the relative fraction of wasteful power consumption dissipated by DRAM refresh operations is growing to compensate for weaker retention capability [13].

Due to two critical system-level problems caused by the above DRAM challenges, system designers cannot stay with a traditional DRAM-only approach to build a system memory. First, since DRAM is not scalable, the main memory capacity of a future platform is insufficient to accommodate an ever-increasing memory footprint that is required from applications running on a much larger number of cores. After all, the deficit of memory capacity becomes a major system bottleneck that considerably degrades the application performance. Second, a system power budget is prone to being violated by the addition of excessive wasteful refresh power consumption to that dissipated by demand memory references. Though putting memory devices in power-down mode may delay the violation, a DRAM cell must be refreshed before losing the stored data.

To address these problems, a variety of emerging memory technologies such as PCM (phase change memory) [7], ReRAM (Resistive RAM) [71], and STT-MRAM (spin-transfer torque magneto-resistive RAM) [103] have been considered for a DRAM surrogate or extension in system memory. However, emerging NVRAM technologies have drawbacks as well as advantages compared to DRAM. For instance, PCM achieves higher density than DRAM, but the latter is faster than the former. For that reason, it is a promising design choice to have both DRAM and NVRAM collaboratively form a system main memory [31] until NVRAMs become as fast as DRAM. There are two main design approaches to build a DRAM+NVRAM system memory. One is a *DRAM cache* that uses DRAM as an OS-invisible caching store of NVRAM main memory, and another is a *hybrid memory* which makes both DRAM and NVRAM visible to OS [68, 91, 94, 118]. A hybrid memory allows more flexible resource management than DRAM cache because a system can address the entire memories across both.

In a hybrid memory, it is important to maximize synergy from both types of memories by hiding downsides of DRAM with advantages of NVRAM or vice versa. The synergistic effect can be enabled by a *page swap* which allows a system to dynamically place frequently accessed (hot) pages in DRAM and less frequently referenced (cold) pages in NVRAM [88],[94]. A sound page swap can boost the performance of applications running on a system with a hybrid memory by timely relocating NVRAM resident pages to DRAM and by servicing most memory accesses from DRAM, as if applications run with a fast and colossal DRAM-only memory system.

For a page swap to be feasible, a system should be able to classify pages into hot and cold ones, and fulfill the required page swaps with low overhead. Unfortunately, in modern systems, OS-managed page LRU lists are too dormant to classify hot and cold pages. Figure 22 displays OS LRU lists' length changes measured from a real server system while running *429.mcf*, one of the most memory-intensive programs from SPEC CPU2006 [105]. The program starts at ❶ and finishes at ❹. The x-axis shows a wall-clock time of program execution and y-axis represents the number of pages in each LRU page list. From the figure,

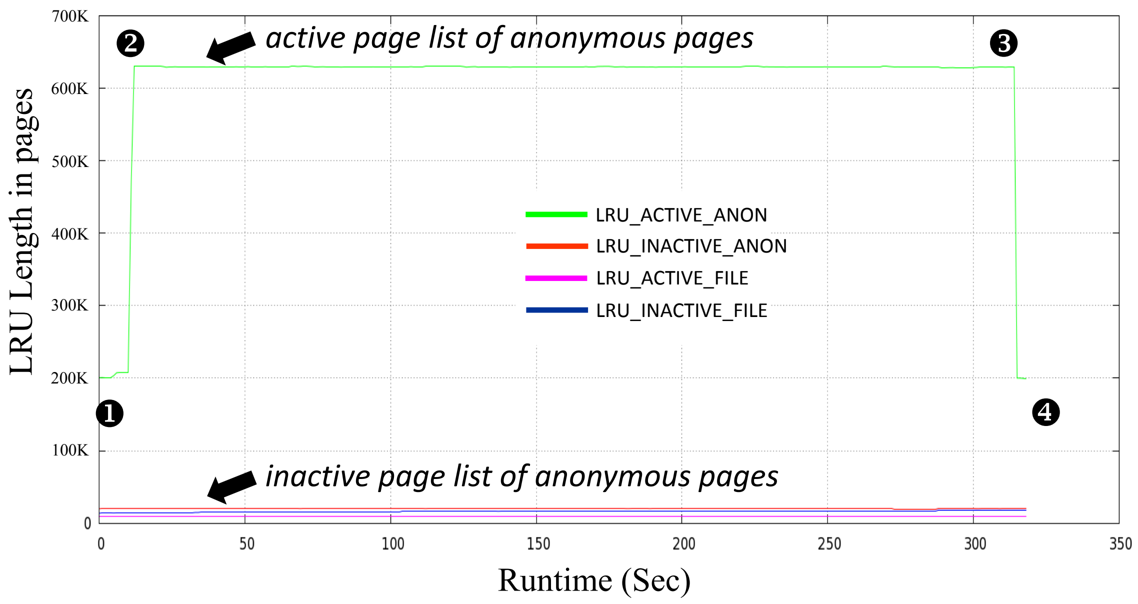


Figure 22: Change of OS LRU list length while running *429.mcf*.

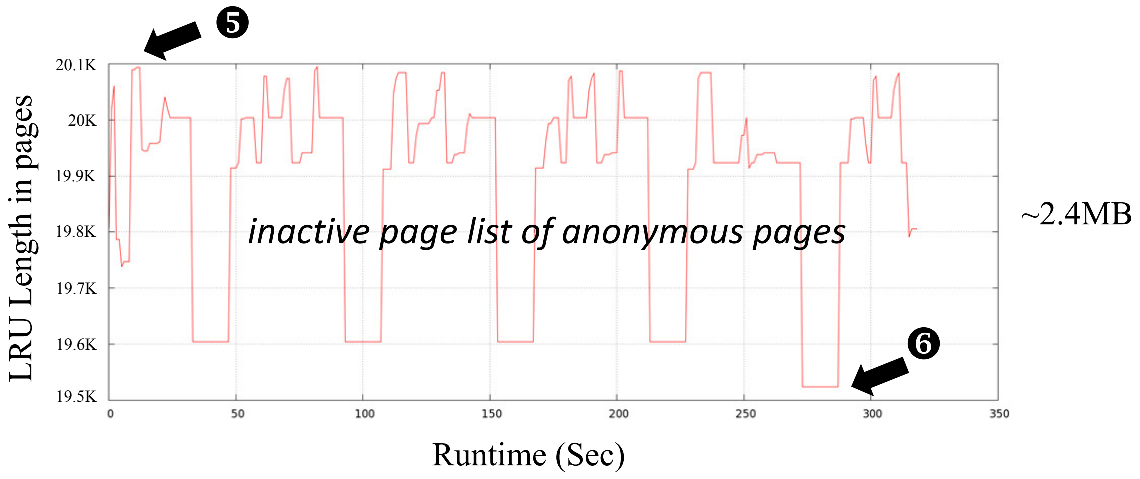


Figure 23: Change of OS LRU list length for inactive pages while running *429.mcf*.

we can observe that as soon as the program starts running, an LRU list of active anonymous pages [1] rapidly grows up to its memory footprint of about 1.7GB as measured with `top`. Thereafter, all of LRU lists remain stationary without a noticeable change until the program

[1]An anonymous page in Figure 22 denotes a page which is not backed by a file. In this work, we define a memory-intensive application as a program using many anonymous pages (e.g. heap or stack pages), not file-backed pages. Hence, if a program is dominated by the file-backed pages, it is not a memory-intensive application, but an I/O-intensive application. Note that we study memory-intensive applications here.

exits (see ❷ to ❸). Interestingly, an LRU list of inactive pages to which a cold page belongs does not vary, erroneously meaning that no page is cold! Zooming-in inactive page list at Figure 23 shows that its maximum variation between peak (❺) and valley (❻) is at most 2.4 MB. We have consistently observed this phenomenon for all memory-intensive programs in the benchmark suite. Different from a conventional wisdom, OS's LRU lists rarely update their length throughout the entire program execution. Accordingly, one cannot resort to page reference information from OS to decide a page swap. Instead, a system must maintain a hardware-driven page reference history for a page swap to be feasible. From this observation, we study a hardware-driven page swap, assuming that a hardware-managed page reference history is available to a future hybrid memory system.

4.2 ANALYTICAL MODEL OF HARDWARE-DRIVEN PAGE SWAP IN HYBRID MAIN MEMORY

In this section, we identify key parameters in which a hybrid memory system is affected by a page swap, and develop an analytical model to investigate the effect of a hardware-based page swap in the context of the flow control.

4.2.1 A Studied System Architecture and Model Parameters

In this chapter, we develop a model to assess the system-level lifetime improvement of NVRAM resources by the proposed Memorage architecture. A studied system architecture which features a hybrid main memory is shown in Figure 24. We assume that a hybrid memory system consists of DRAM and PCM currently available, but hereafter we refer to them as FM and SM (Faster and Slower Memory) for notational simplicity and generality.

Table 4 describes key model parameters used in our model. Our model is based on a simple queuing theory, specifically *Little's law* [70],[43] which relates the average request arrival rate, the average number of requests in memory system, and the average service time of a request. From Figure 24, we can see that memory requests arrive at a rate of λ_0, and a

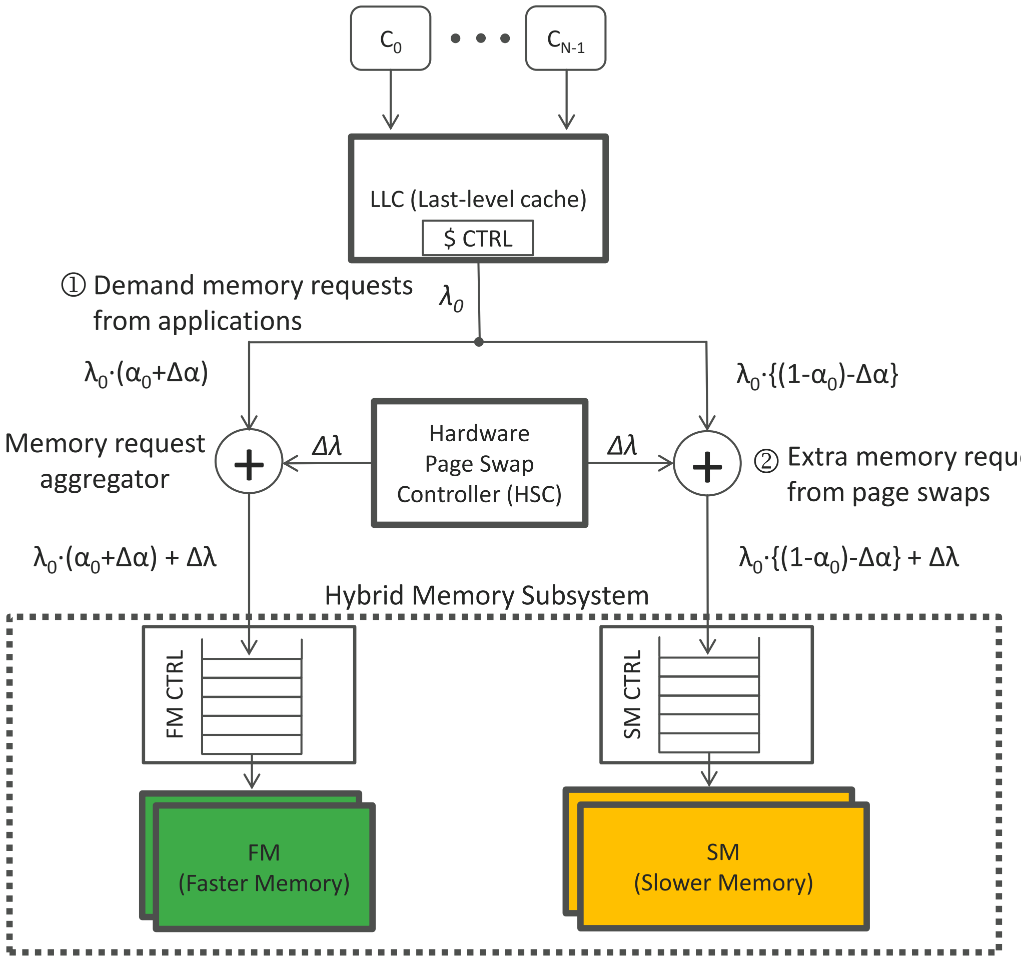


Figure 24: System architecture with hybrid main memory.

fraction of α_0 to total memory requests accesses FM while the remaining fraction of $(1-\alpha_0)$ to total requests is serviced by SM. When page swap is enabled, swap operations not only increase memory request rate by $\Delta\lambda$ to each memory region, but also change a fraction of FM references by $\Delta\alpha$. These two changes have an opposite effect on the application performance. In this work, we assume that HSC (Hardware Page swap Controller) chooses and swaps a hot page from SM with a cold page from FM pairwise instead of moving a page in only one direction.

Since we aim at improving the performance of memory-intensive workloads via page swap, we employ a metric of CPI (cycles per instruction) [33] or IPC (instruction per cycle) to compare the application performance improvement. Note that an average access time improvement of a memory subsystem is not always directly proportional to the application performance improvement. For instance, an application with high instruction-level parallelism may effectively hide long-latency memory access penalties. In this case, the application performance may not be sensitive to small improvement of memory access time.

4.2.2 Baseline CPI Model without In-Memory Page Swap

Let us consider an application running on a system with a hybrid main memory. The CPI of the application can be formulated as the sum of the CPI achieved with an ideal LLC and the memory access penalty per instruction. An ideal LLC can serve all LLC references without accessing main memory. However, a non-ideal LLC has a limited capacity, and a missed LLC reference costs memory access penalty. We can consider an average memory access penalty as AMAT (Average memory access time) in cycles. Therefore, an application performance without page swap can be expressed as follows:

$$\begin{aligned} CPI_{base} = CPI_{ideal} + MPI_{base} \times \{\alpha_0 \cdot (T_{FM} + T_{queue,FM}) + \\ (1-\alpha_0) \cdot (T_{SM} + T_{queue,SM})\} \end{aligned} \tag{4.1}$$

Considering $\lambda_0 = MR_{LLC} \times AF_{LLC}$ and $MPI_{base} = \lambda_0 \times \frac{CPI_{base}}{f_{core}}$, we can rearrange equation 4.1 by applying the above relations to MPI_{base} and solving it for CPI_{base}. Then, we get the following equation 4.2 to represent the baseline CPI performance for an application running on a system without page swap.

$$\begin{aligned} CPI_{base} = \frac{CPI_{ideal}}{1 - \frac{\lambda_0}{f_{core}} \cdot F_0}\ , \\ where\ F_0 = \{\alpha_0 \cdot (T_{FM} + T_{queue,FM}) + (1-\alpha_0) \cdot (T_{SM} + T_{queue,SM})\} \end{aligned} \tag{4.2}$$

F_0 in equation 4.2 includes queuing delay terms, $T_{queue,FM}$ and $T_{queue,SM}$ for FM and SM, respectively, thus we further need to derive the queuing delays which a memory request experiences in a memory controller's request queue on average. According to *Little's law*,

Parameters	Description of model parameters
f_{core}	CPU clock frequency (Hz = cycles/sec)
T_{FM}	Average FM access latency (cycles)
T_{SM}	Average SM access latency (cycles)
$T_{queue,FM}$	Average FM access queuing delay without page swap
$T_{queue,SM}$	Average SM access queuing delay without page swap
$T_{pqueue,FM}$	Average FM access queuing delay with page swap
$T_{pqueue,SM}$	Average SM access queuing delay with page swap
CPI_{ideal}	Cycles per instruction with 100% LLC hit
CPI_{base}	CPI without page swap
CPI_{pg}	CPI with page swap
MPI_{base}	LLC misses per instruction without page swap
MPI_{pg}	LLC misses per instruction with page swap
MR_{LLC}	LLC miss rate (misses/access)
AF_{LLC}	LLC access frequency (accesses/sec)
α_0	Fraction of FM accesses without page swap
α	Fraction of FM accesses with page swap
$\Delta\alpha$	Changed fraction of FM accesses by swap: $\alpha - \alpha_0$
λ_0	Memory access rate without page swap (accesses/sec)
λ	Memory access rate with page swap (accesses/sec)
$\Delta\lambda$	Increased memory access rate by page swap: $\lambda - \lambda_0$
F_0	Average memory access time without page swap
F	Average memory access time with page swap
x	Relative ratio of access latency of FM and SM: $x = \frac{T_{SM}}{T_{FM}}$

Table 4: Performance Model Parameters

the length of queue and the expected queuing delay of a memory request in each type of memory region can be expressed as follows:

$$\begin{aligned}
Len_{q,FM} &= \lambda_{0,FM} \cdot \frac{T_{FM}}{f_{core}} = \alpha_0 \cdot \lambda_0 \cdot \frac{T_{FM}}{f_{core}} \\
T_{queue,FM} &= Len_{q,FM} \cdot T_{FM} = \alpha_0 \cdot \lambda_0 \cdot \frac{{T_{FM}}^2}{f_{core}} \\
Len_{q,SM} &= \lambda_{0,SM} \cdot \frac{T_{SM}}{f_{core}} = (1-\alpha_0) \cdot \lambda_0 \cdot \frac{x \cdot T_{FM}}{f_{core}} \\
T_{queue,SM} &= Len_{q,SM} \cdot T_{SM} = (1-\alpha_0) \cdot \lambda_0 \cdot \frac{x^2 \cdot {T_{FM}}^2}{f_{core}}
\end{aligned} \tag{4.3}$$

Note that $T_{queue,SM}$ in equation 4.3 has been expressed with T_{FM} by relating T_{SM} to T_{FM} with the assumption that FM is x times faster than SM (i.e. $x = T_{SM}/T_{FM}$ and $x > 0$). This allows us to investigate the effect of various relative latency performance differences between FM and SM.

For this model, we employ *m/m/1* queuing model to simplify our model and the hardware implementation. Although it is possible to extend our model to use *m/m/c* or non-memoryless queuing model to take the device-level parallelism or request burstiness into account, we show that the simple queuing model is accurate enough to determine the profitability of page swap in section 4.3.

4.2.3 New CPI Model with In-Memory Page Swap

Similar to the baseline CPI model, we can derive a CPI performance model of a system with page swap as follows:

$$\begin{aligned}
&CPI_{pg} = \frac{CPI_{ideal}}{1 - \frac{\lambda_0}{f_{core}} \cdot F}\ , \\
&where\ F = \{\alpha \cdot (T_{FM} + T_{pqueue,FM}) + (1-\alpha) \cdot (T_{SM} + T_{pqueue,SM})\}, \\
&and\ \alpha = \alpha_0 + \Delta\alpha
\end{aligned} \tag{4.4}$$

Now let us formulate the queuing delay terms, $T_{pqueue,FM}$ and $T_{pqueue,SM}$ in equation 4.4. Using *Little's law* again and memory access rates seen by FM, SM, and the overall hybrid memory (i.e. $\lambda_{FM}, \lambda_{SM},$ and $\lambda = \lambda_{FM} + \lambda_{SM}$), we finally get the formula of per-region queuing delays in equation 4.5.

$$
\begin{aligned}
Len_{pq,FM} &= \lambda_{FM} \cdot \frac{T_{FM}}{f_{core}} = \{\alpha \cdot \lambda_0 + \Delta\lambda\} \cdot \frac{T_{FM}}{f_{core}} \\
T_{pqueue,FM} &= Len_{pq,FM} \cdot T_{FM} = \lambda_{FM} \cdot \frac{{T_{FM}}^2}{f_{core}} \\
Len_{pq,SM} &= \lambda_{SM} \cdot \frac{T_{SM}}{f_{core}} = \{(1-\alpha) \cdot \lambda_0 + \Delta\lambda\} \cdot \frac{x \cdot T_{FM}}{f_{core}} \\
T_{pqueue,SM} &= Len_{pq,SM} \cdot T_{SM} = \lambda_{SM} \cdot \frac{x^2 \cdot {T_{FM}}^2}{f_{core}} \ , \\
&\mathit{where}\ \lambda_{FM} = \alpha \cdot \lambda_0 + \Delta\lambda,\ \mathit{and}\ \lambda_{SM} = (1-\alpha) \cdot \lambda_0 + \Delta\lambda
\end{aligned}
\tag{4.5}
$$

From Figure 24, equation 4.4, and 4.5, we observe that a hardware-based page swap allows an application's demand memory access rate, λ_0 to be independent of an additional memory request rate, $\Delta\lambda$ induced by page swap because LLC is oblivious to swap activities generated by a hardware page swap controller beneath it.

4.2.4 Profitability of page swap

For a page swap to be justified, the CPI of an application with page swap must be smaller than or equal to CPI_{base}. That is, the advantage of $\Delta\alpha$ (i.e. having a larger proportion of access to the faster memory) should outweigh the disadvantage of $\Delta\lambda$ (i.e. additional load on the memory system due to page swaps). From this simple axiom, we can derive a formula to determine whether or not a page swap operation is justifiable. A page swap may harm the application performance unless the inequality in equation 4.6 holds.

$$
\begin{gathered}
CPI_{pg} \le CPI_{base} \\
\frac{CPI_{ideal}}{1 - \frac{\lambda_0}{f_{core}} \cdot F} \le \frac{CPI_{ideal}}{1 - \frac{\lambda_0}{f_{core}} \cdot F_0}
\end{gathered}
\tag{4.6}
$$

By eliminating common variables from both sides in equation 4.6, we notice that examining $F \le F_0$ has the same effect as checking $CPI_{pg} \le CPI_{base}$. In other words, F, an average memory access penalty of a page swap-enabled system should be smaller than or equal to F_0 in order to improve application performance via page swap.

4.3 COMPARING MODEL WITH SIMULATION

In this section, we compare the outcome of our page swap model with the result of an architecture simulation, and make observations of page swap behavior in a hybrid memory.

4.3.1 Experimental Setup and Simulation Methodology

Simulator and configuration

We employ the *Ramulator* [63] as a reference architecture simulator over which the outcome of the model is compared. The simulator has been validated against Micron Technology's hardware-level DRAM device model as well as JEDEC DDR specification. However, we extended the simulator to support a hardware page swap controller, *HSC* and a hardware page table. Also, each memory controller has a global request queue. Table 5 details our simulation configurations.

Benchmark applications

For the performance evaluation, we use instruction traces of 21 memory-intensive SPEC CPU2006 benchmarks (configured with reference input data) as input to *Ramulator*. The instruction traces were collected using Pin [76] and SimPoint [89], and filtered through 512KB cache. Using these traces, we form a total of 11 multiprogrammed, memory-intensive workloads, each of which is comprised of 6 applications. Table 6 shows the workloads used for our study [2]. Note that an average memory footprint of our workloads for single core is 1.93× larger than that from recent research [25].

Simulation methodology

We assume other workloads in a system already occupy a portion of FM memory capacity, so our workload is not allowed to allocate its entire pages to FM. In particular, we assume that our workload could get allocated just a half of its footprint on FM due to other workloads. For that, we first allocate the same number of pages to each memory region via a pre-

[2]Since SimPoint helps Pin tool collecting traces only for representative phases of a program execution through a sampling, the memory footprint size of our workloads is smaller than that observed during the complete execution of the applications.

[3]We simulate a scenario that memory footprint size of our workloads entirely fit in a hybrid memory and perform page swappings only within the given memory footprint without an additional page allocation. Therefore, having smaller SM device than FM device does not harm our simulation purpose.

Parameters	Configuration
CPU	3.2GHz, out-of-order, 4-wide issue superscalar processor, 128-entry ROB
LLC	512KB L2 cache
FM Controller	128-entry request queue, FR-FCFS scheduling, closed page mode, row-col-rank-bank addressing, 1 channel, 1 rank
SM Controller	128-entry request queue, FR-FCFS scheduling, open page mode, row-col-rank-bank addressing, 1 channel, 1 rank
FM	DDR4-2400 DRAM, 8 banks, 1KB page size, tCL-tRCD-tRP-tRAS=16-16-16-39, tREFI=7.8us, 4GB
SM	DDR3-1600 PCM-like, 8 banks, 1KB page size, tCL-tRCD-tRP-tRAS=11-44-0-59, tREFI=∞ (no refresh), 512MB [3], slower than FM (R: 4$\times$, W: 10$\times$)

Table 5: Simulation configuration

execution before measuring page swap performance. This allows us to decouple the effect of a page swap from a disputable side-effect caused by the memory capacity available in each region. For example, if FM has a sufficient memory capacity and bandwidth to serve the entire memory footprint of a workload, we hardly find a reason to harness SM or move any page from FM to SM. After a pre-execution, since a page swap always deals only with already memory-resident pages which may be recycled as destination pages for a page swap, it is possible to evaluate the usefulness of a page swap for a specific memory footprint size.

To do a page swap, HSC monitors page access frequency in each region for an observation interval or epoch. Then, HSC classifies pages by the frequency at the end of the interval, and performs zero or some number of page swaps for a subsequent interval. We repeat this "monitoring-and-swap" process.

For this simulation, we set an observation interval to one full execution of a workload trace, and iterate the execution 5 times after a pre-execution. Since a collected instruction trace is a representative phase of execution in an application, the phase is executed multiple

WL	Six applications in a workload (WL)	Footprint
mix1	cactusadm, gemsfdtd, soplex, libquantum, lbm, mcf	577 MB
mix2	bzip2, namd, hmmer, leslie3d, cactusadm, leslie3d	120 MB
mix3	omnetpp, lbm, astar, zeusmp, h264ref, sjeng	671 MB
mix4	libquantum, lbm, wrf, zeusmp, h264ref, sjeng	435 MB
mix5	soplex, lbm, wrf, libquantum, dealII, sphinx3	270 MB
mix6	lbm, cactusadm, gcc, dealII, gemsfdtd, xalancbmk	537 MB
mix7	milc, lbm, h264ref, wrf, cactusadm, libquantum	634 MB
mix8	mcf, zeusmp, astar, sphinx3, gromacs, omnetpp	419 MB
mix9	libquantum, leslie3d, cactusadm, sjeng, gobmk, soplex	281 MB
mix10	gemsfdtd, omnetpp, namd, hmmer, gromacs, milc	719 MB
mix11	mcf, bzip2, gcc, sjeng, leslie3d, milc	587 MB

Table 6: Multiprogrammed workloads

times throughout the program's execution. Therefore, it makes sense to repeat a chosen phase of run. Meanwhile, we statically set the maximum number of page swaps within an iteration to 500. This static swap capability eases a comparison of the behavior of page swap between simulation and model.

4.3.2 Evaluation Results

In this section, we demonstrate that the developed page swap model is valuable as an off-line analysis tool for page swap. Our model is useful for studying how a page swap affects the application performance.

Let us compare the AMAT of an analytical model over that measured and reported by the architecture simulation, because F_0 and F are key factors of assessing the profitability of a page swap in our model. To compute the AMAT from the analytical model, we measure the proportion of FM access (α_0 and $\Delta\alpha$), memory request rate (λ_0 and $\Delta\lambda$), runtime latency

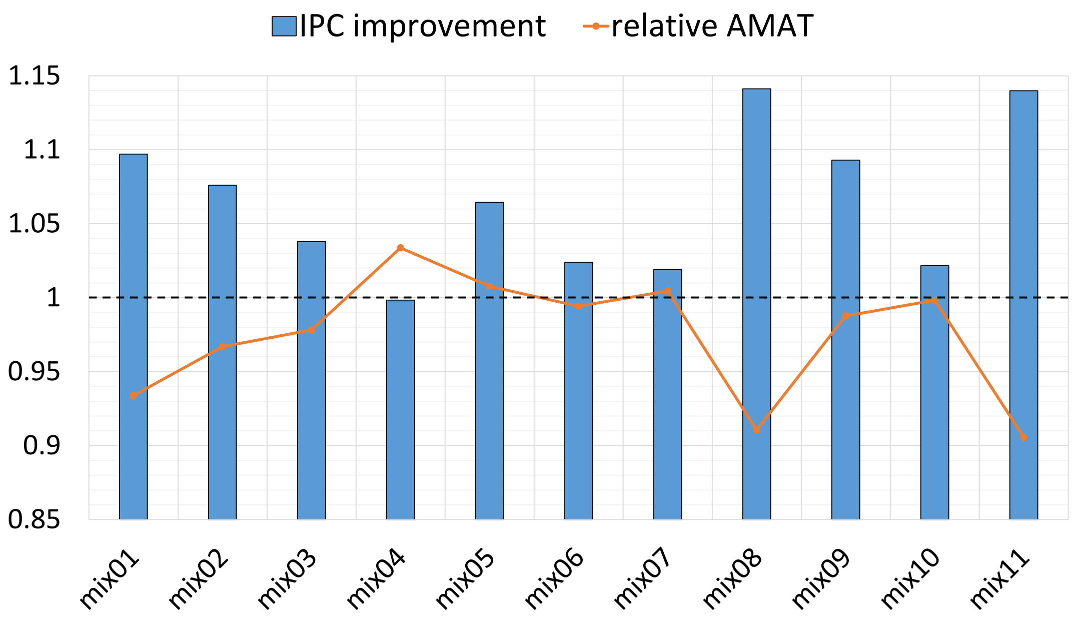


Figure 25: IPC improvement and AMAT decrease of a static page swap over the baseline.

difference between memory regions (x), and memory access counts and cycles per region, then apply them to equations 4.2 and 4.4. Figure 25 shows simulation results of the IPC improvement and the AMAT decrease over the baseline (i.e. no swap). From the figure, we can observe that the page swap achieves 6.5% performance improvement on average, and that the AMAT improvement of a workload results in the IPC performance improvement in most cases. For example, the workloads `mix01`, `mix08`, and `mix11` achieve the largest AMAT improvement with page swaps, and they achieve the largest IPC improvement thanks to the AMAT improvement. On the contrary, the AMAT of `mix04` becomes worse, thus its IPC performance also degrades. We found that a page swap rarely increases FM use for `mix04` and `mix07`. Thus, their resulting performance degrades or improves little. However, we note that the resulting performance improvement across all workloads is somewhat small, considering the media access latency difference of FM and SM in Table 5. This happens because the access latency of FM and SM as a memory module is different from their cell-level latencies due to the system-level effect (e.g. parallelism).

Observation1: "dynamic access latency gap of two memory regions is small." Figure 26 depicts that a dynamic latency gap (x) between FM access and SM access is

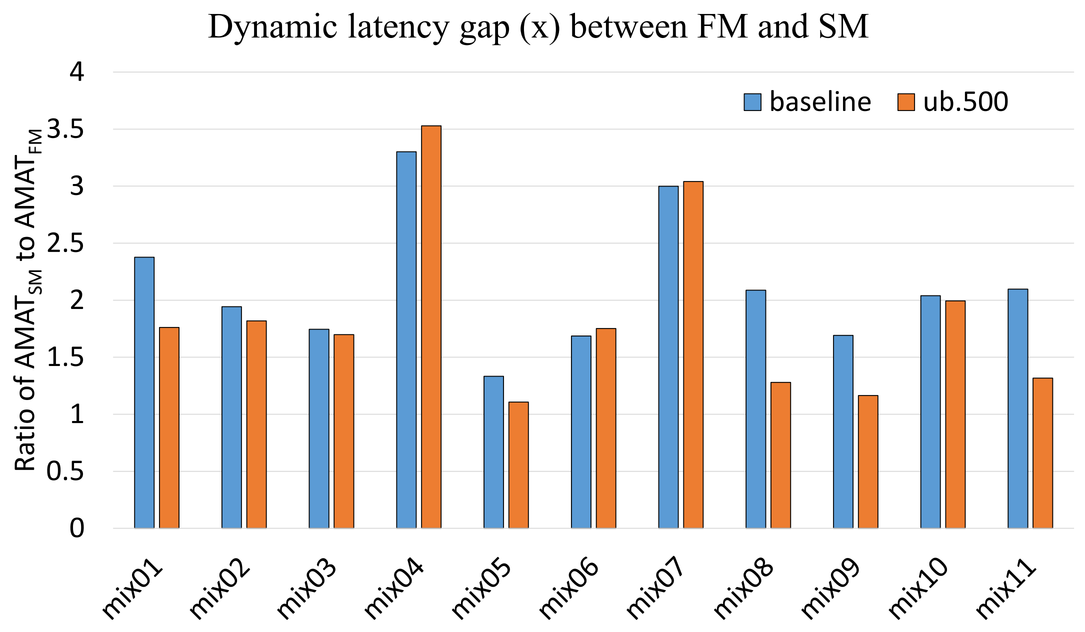


Figure 26: Dynamic access latency gap between two memory regions, FM and SM.

smaller than the media latency configured in Table 5. Different from 4× and 10× gap for read and write respectively, the actual average memory access latency gap at a runtime for both reads and writes is 2.11× which is much smaller than the media latency. This is mainly attributed to the convoluted effect of three factors. First, the write-back policy of LLC effectively alleviates the direct impact of an inferior SM write performance on the application performance. Second, a memory request is decomposed into multiple media specific sub-commands inside a memory controller, then these commands are executed with ones from other memory requests in parallel [13]. This further reduces the latency gap between FM and SM. Third, workloads hardly saturate the provisioned memory system [11],[60], so the application performance is not proportional to a given device-level latency gap. Since the dynamic latency gap between two regions is small, the gain achieved by a page swap becomes small as well.

Observation2: "direction of AMAT variation is closely correlated."

Figure 27 shows the comparison of the AMAT change of the model to the simulation. A total of 10 workloads as much as 90.9% have a positive correlation, which suggests that if the

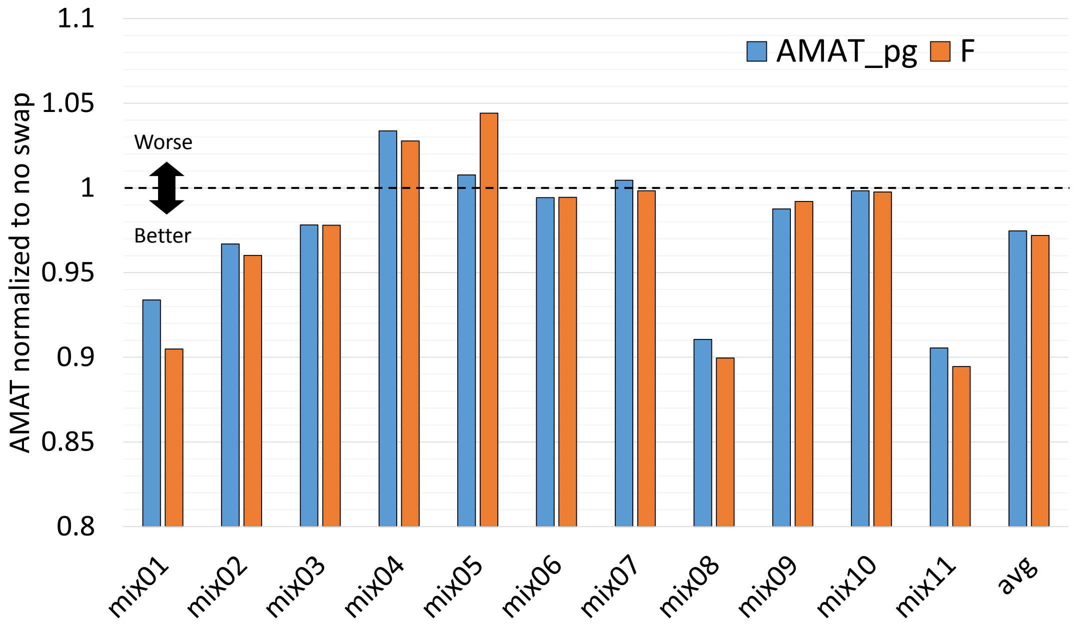
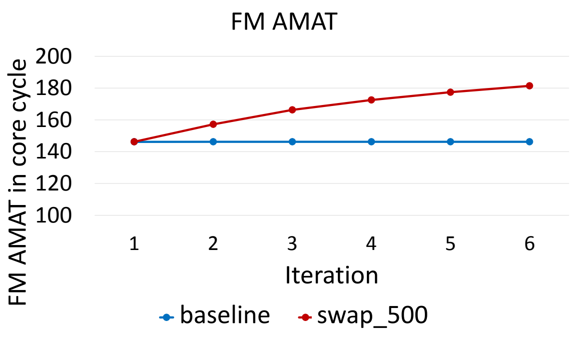


Figure 27: AMAT comparison of simulation ($AMAT_{pg}$) over model (F)

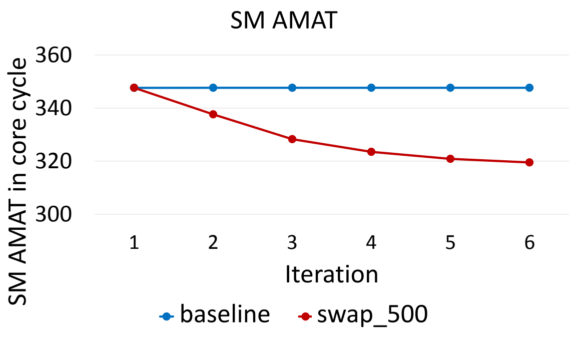
AMAT calculated from our model improves over the baseline, the AMAT from the simulation also shows the improvement, and vice versa. However, `mix07` has a negative correlation that AMAT change direction of the simulation mismatches with the model. That is, the model indicates the AMAT improvement for mix07, but the AMAT measured from the simulation deteriorates. This discrepancy happens because `mix07` is a write-heavy workload and our model does not differentiate the queuing effect of read and write requests. As a result, our model trades accuracy for model simplicity. Although there may exist a disagreement on AMAT increment or decrement between the model and the simulation, it is important to note that our "simple" model can effectively evaluate the profitability of a page swap only with modest error (9.1%) based on the measured values of α and λ. Note that knowing the direction of AMAT change, not its magnitude, suffices to evaluate the profitability of a page swap.

Figure 28 comparatively depicts AMAT changes of a system-wide as well as each memory region for `mix01` and `mix04` [4]. The *baseline* in the figure denotes a scheme that has no swap,

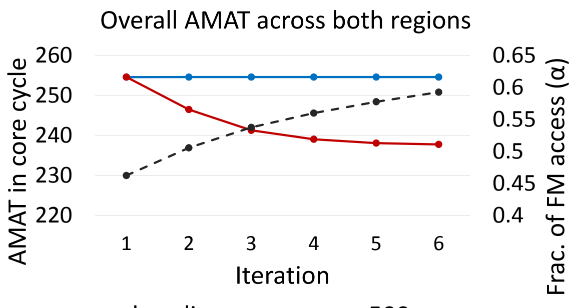
[4]We compare `mix01` and `mix04` to examine cases in which their AMAT improves and deteriorates, respectively.



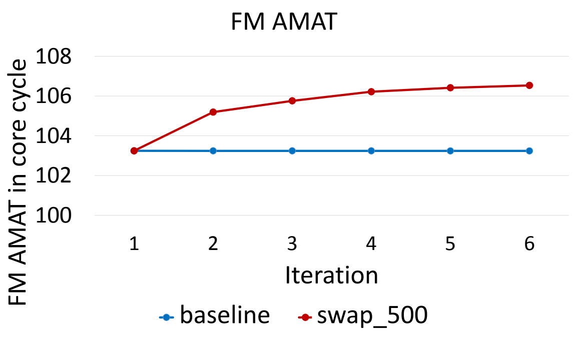
(a) mix1 $AMAT_{FM}$



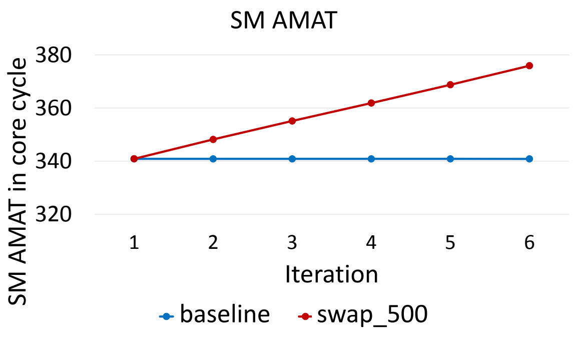
(b) mix1 $AMAT_{SM}$



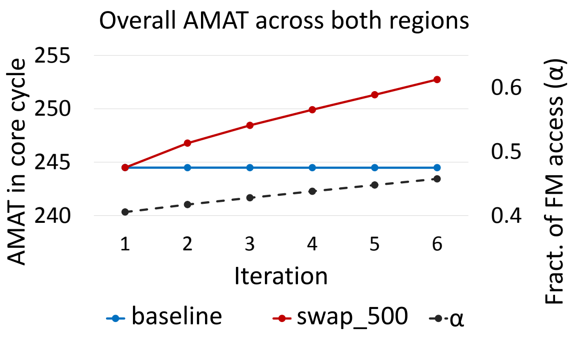
(c) mix1 $AMAT$



(d) mix4 $AMAT_{FM}$



(e) mix4 $AMAT_{SM}$



(f) mix4 $AMAT$

Figure 28: AMAT and α change for mix01 and mix04.

while the *swap_500* represents a scheme which performs 500 page swaps per iteration. From Figure 28a and 28b of `mix01`, we see the AMAT of FM region degrades with 500 page swaps, whereas SM region's AMAT improves over the baseline. Since the page swap has FM

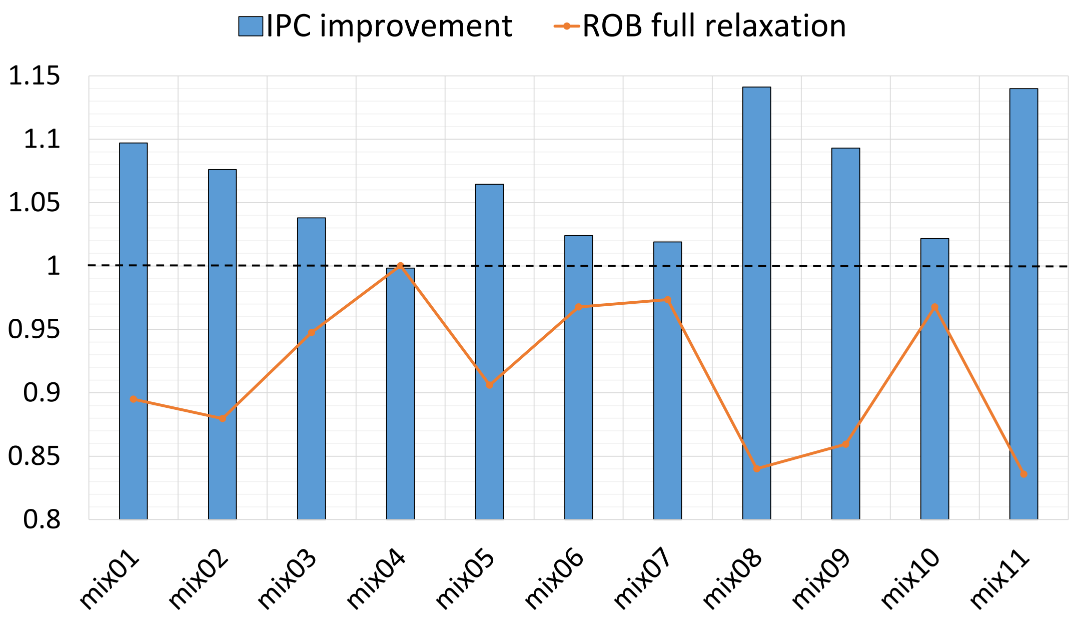


Figure 29: IPC improvement and ROB fullness decrease over baseline.

serve 13% more memory requests than the baseline, the overall AMAT of `mix01` improves. Note that the gain from AMAT improvement of SM region outweighs the loss by AMAT degradation in FM region, because SM is slower than FM by 2.4 times at runtime in the baseline (see Figure 26). On the contrary, Figure 28d and 28e of `mix04` show the AMAT of both regions deteriorates after page swaps. In particular, we can observe that SM region does not benefit from page swaps due to its high utilization. After all, page swaps add more requests to the overloaded SM region, and the overall AMAT of `mix04` degenerates. Also, it is noteworthy that page swap could redirect just 5% more memory requests to FM.

Observation3: "ROB fullness affects a page swap."

`mix05` and `mix07` in Figure 25 seemingly show counter-intuitive results because even if the AMAT degrades by a page swap, a system achieves better IPC performance. This happens because the bottleneck of a hybrid memory system for these workloads is at ROB (Re-Order Buffer) in a processor front-end. Figure 29 depicts the relationship of the IPC improvement to the decrease of ROB fullness. We observe that both `mix05` and `mix07` experience less number of ROB fullnesses with a page swap. Due to the in-order retirement property of ROB entries, entries occupied by memory instructions which require long-latency SM access

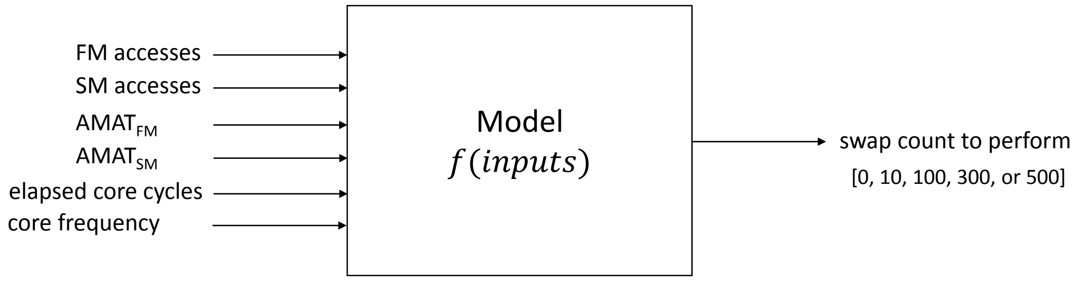


Figure 30: Illustration of model-guided page swap. The model f determines the number of page swaps to perform at the end of current epoch τ_i, expecting to improve the application performance in next epoch τ_{i+1}.

cause a HOL (Head-Of-Line) blocking problem, which prevents subsequent non-memory instructions from being retired and releasing their ROB entries. Accordingly, ROB resources become a bottleneck for issuing new instructions in the baseline. The page swap mitigates the problem by making these non-memory instructions own ROB entries for a shorter time, thus a system achieves better performance. This implies that a hybrid main memory sometimes needs to perform a page swap in order to relax ROB bottleneck even if the overall AMAT slightly gets worse.

4.4 MODEL-GUIDED HARDWARE-DRIVEN PAGE SWAP

The model in Section 4.2 can also serve as an online page swap tool to regulate both rate and volume of a page swap on-the-fly, adapting to the change of a program execution phase. In this chapter, we assess the usefulness of our model by applying the model to the architecture simulator and observing a dynamic behavior of application performance.

Figure 30 depicts what the model takes as an input and yields as an output. The inputs of the model-guided page swap are straightforward and simple enough to derive from hardware performance counters in modern processors. To calculate F and F_0 per epoch, the model in HSC takes FM access count, SM access count, and total elapsed cpu cycles from a previous epoch τ_{i-1}, and uses them to derive α_0 and λ_0. Likewise, HSC newly counts FM and SM

Configuration	Description
baseline	This scheme does no page swap.
ub.c	This scheme does page swap with static, upper-bound c that performs at most [5] c ($\in \{10, 100, 300, 500\}$) page swaps in an epoch.
model	This scheme does a model-guided page swap. A page swap count to take is dynamically chosen from above c ranges by our model.
RaPP [6]	This scheme does a rank-based page swap from previous work [94]. A page swap is determined by multi-queue based page ranking system.

Table 7: The considered page swap schemes

access during current epoch τ_i. In addition, it continuously measures the average memory access time to FM and SM, which enables us to derive a dynamic latency gap x. Meanwhile, instead of deriving $\Delta\alpha$ in F, HSC needs to *predict* $\Delta\alpha$ for next epoch τ_{i+1} at the end of the current epoch because it is nearly impossible to know how many more requests will be served by FM after performing a current page swap decision. As a proxy to approximate $\Delta\alpha$, we use the difference between total access counts of the considered hottest SM pages and total reference counts of the considered coldest FM pages. The number of hottest or coldest pages to consider is decided by iteratively checking values in a finite set of upper-bounds, which are the number of page swaps allowable for an epoch. The model can derive the estimated $\Delta\alpha$ for each upper-bound value, and in turn derive $\Delta\lambda$ because we know that a page swap incurs 128 page references on each region. Then, the model is ready to assess the profitability of a page swap using the derived parameter values.

[5]These schemes try to do at most c page swappings in an epoch, even if more than c pages in SM have greater page access frequencies than cold pages in FM. For instance, when 513 hottest pages in SM region have larger access counts than 513 coldest pages in FM region, ub.500 scheme performs only 500 page swaps instead of 513. This static upper bound can simplify hardware implementation.

[6]The **Ra**nk-based **P**age **P**lacement (RaPP) leverages hardware multi-queue (MQ) to not only rank pages on the basis of the page access frequency but also determine when and what pages are replaced between FM and SM. For this simulation, we faithfully configure MQ parameters as described in Ramos et al. [94] (e.g. the number of rank queues, a threshold queue, the page liftime, and a filter threshold). Refer to [94] for other details of RaPP-specific parameters and their algorithm.

4.4.1 Simulation Methodology

To evaluate the effectiveness of our model-guided page swap, we continue to use an experimental setup from previous Section 4.3.1. However, for this simulation, we continuously switch applications in a workload every 128 instruction execution that an application can fill up ROB once, until the workload completes one run. This has an effect similar to running a multiprogrammed workload on a multi-core processor. Even if we rapidly switch an application with another to alternate the ownership of a processor, note that each application maintains its locality and memory reference order.

Meanwhile, we set an epoch to a time span to execute 12.8 million instructions, which corresponds to 4 milliseconds in our processor clock speed. As before, we perform an epoch-based page swap at the end of an epoch, and the memory requests by a page swap compete for a request queue in a memory controller with demanding requests from applications. However, this simulation additionally takes into account a rank-based page swap scheme (RaPP) from previous page migration research so that we can compare our scheme over the existing related work. This RaPP scheme continuously determines the need of page swap based on a page ranking system, and performs page swaps on-the-fly throughout an epoch, not just at the end of epoch. In addition, different from others, RaPP scheme has no upper-limit on the number of pages that can swap in an epoch as original authors assumed. Table 7 specifies the page swap schemes that we consider, all of which use the access frequency for hot and cold page classification. However, RaPP scheme additionally leverages a time-based information for that classification to consider a page decaying interval.

4.4.2 Evaluation Results

In this section, we present the simulation results of the proposed model-guided, hardware page swap. Figure 31 compares the overall IPC performance improvement of our model-guided swap to baseline, static schemes, and RaPP policy. From the figure, we find that the model-guided page swap improves application performance by 28.9%, 13.3%, and 26.1% compared to the baseline, static page swap cases, and RaPP, respectively. In contrast to the static schemes that simply rely on page access frequency, our model-guided scheme carries

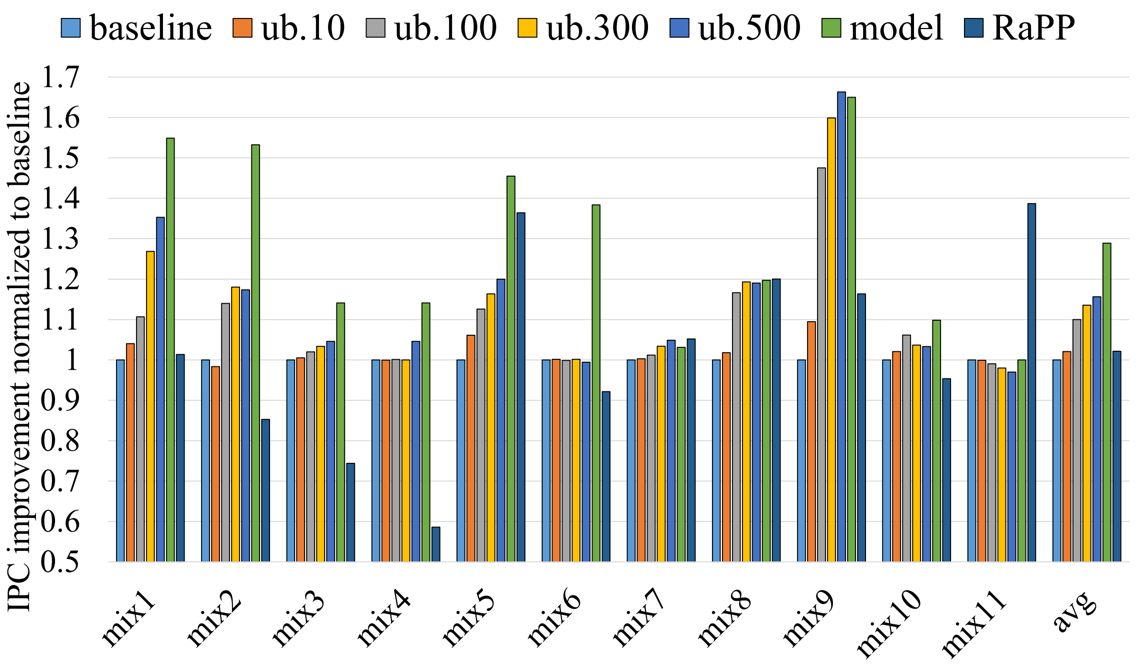


Figure 31: Comparison of IPC performance improvement.

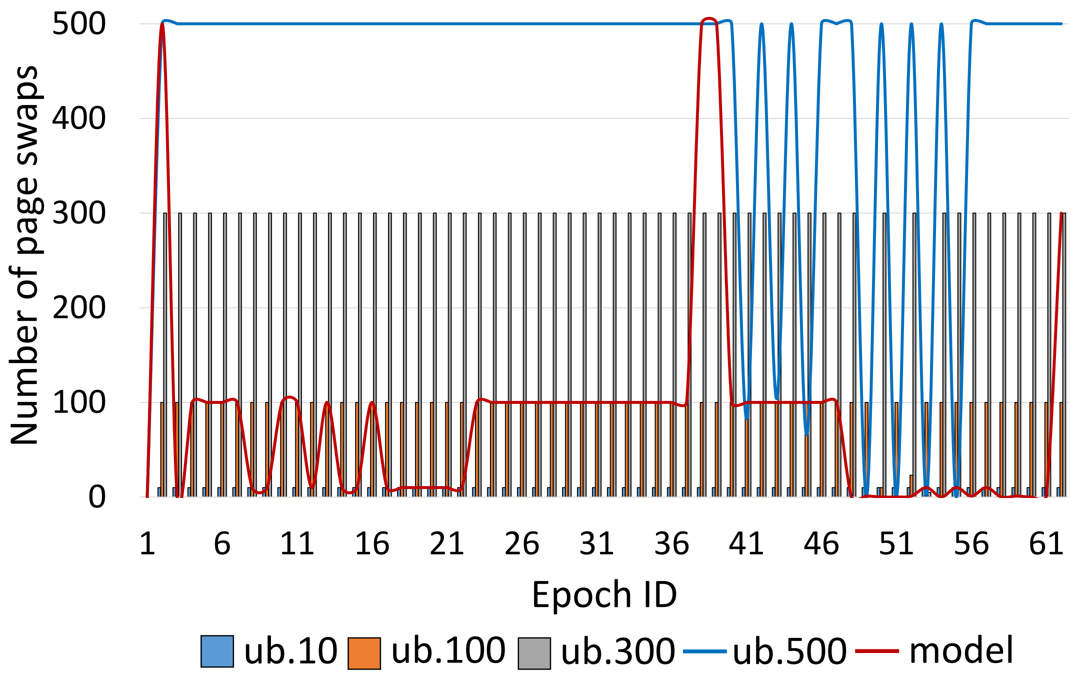
out a page swap only when the model predicts that the page swap can improve performance by considering the memory request arrival rate as well as the service rate of FM and SM region. Figure 32 shows how our model-guided page swap adaptively selects the number of page swaps to regulate page swap rate so that an application can benefit from page swaps. Meanwhile, Figure 33 displays how RaPP scheme performs the page swap over epoch. We only explain the most interesting, representative workloads.

In Figure 31, `mix1` benefits from static page swap schemes. Specifically, it achieves higher IPC performance improvement with a larger number of page swaps. However, we also observe that our model-guided scheme outperforms the *ub.500* scheme even if the number of page swaps of the model-guided scheme is less than that of the *ub.500*. While the *ub.500* resorts only to page hot and cold information and continues to swap pages, the model-guided scheme selectively accomplishes page swaps, depending on the condition of each memory region. Moreover, the *ub.10* and *ub.100* try to swap even when there are insufficient swappable pages around the 50th epoch in Figure 32a. These unjustifiable page swaps hurt the performance. The negative performance impact caused by bad page swaps can be clearly observed in RaPP scheme, which achieves only 1.3% performance improvement over baseline even if it performs 18.7× more page swaps than the model-guide scheme (see Figure 33). This implies that RaPP scheme, which does not evaluate the achievable gain from page swaps, produces a significant amount of non-profitable page swaps and they offset the performance improvement achieved by page swap.

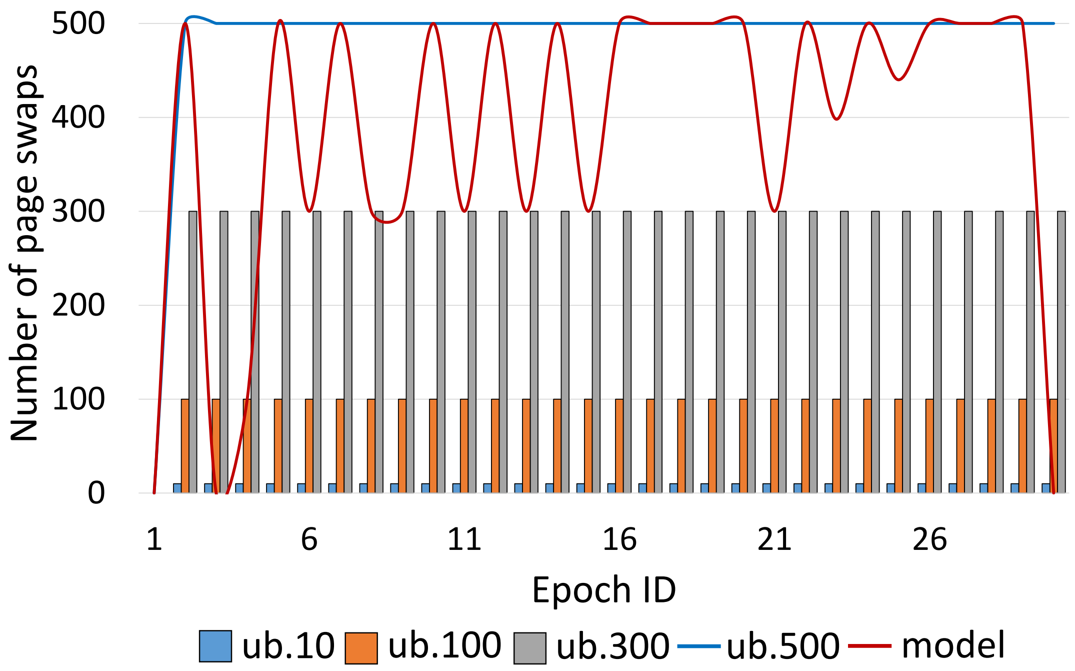
For `mix6`, our model-guided scheme greatly outperforms other schemes, and the gain comes from ameliorating heavy SM utilization. According to page access frequency, the workload has many hot and cold pages appropriate for swap, so static schemes continuously swap pages, although the actual gain achieved by swap is small. However, since the SM region has high utilization, a careless addition of write traffic by a page swap to the region offsets the benefit of swap. Due to the same reason, we found that RaPP scheme underperforms the baseline by 7.9% with 3× more page swaps than the model-guided scheme. In contrast, our model-guided scheme throttles page swaps if the profitability condition is not met, though swappable pages exist. Note that our model-guide scheme makes a difference on the application performance with the capability that swaps the right number of pages at the right moment.

For `mix10`, static schemes *ub.10* and *ub.100* boost the IPC performance. Afterward, the application performance degrades as the number of page swaps increases. Specifically, we observe that later epochs commit the mistake of executing non-profitable page swaps and thus servicing demand memory requests from applications is delayed. We also observe a similar but worse situation with RaPP scheme, whose performance is lower than the baseline by 4.7% due to more unjustifiable page swaps. Compared to this, the model-guided page swap scheme forbids page swap in most epochs immediately after a system accomplishes profitable page swaps in the early stages of execution.

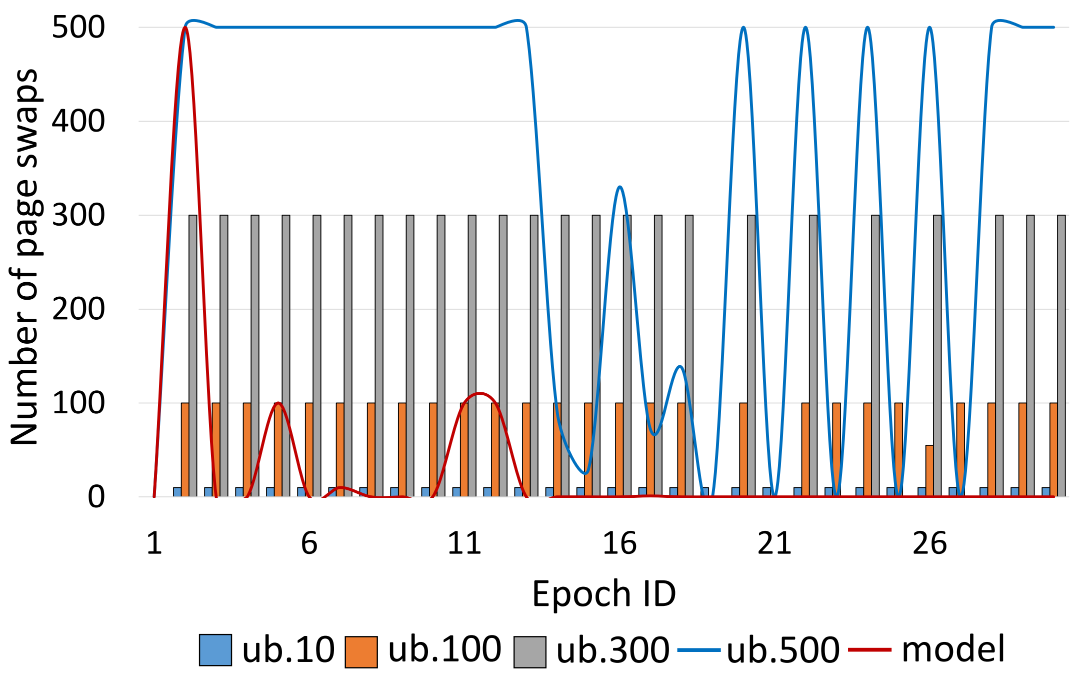
As for `mix11`, all static schemes underperform the baseline. This implies that suppressing any page swap achieves the best performance for this workload. The access pattern of pages in `mix11` is mostly uniform in terms of access frequency, and thus there are not many pages to swap. For example, we can see that the number of page swaps with the *ub.100* scheme goes to zero more frequently compared to `mix6`. Different from static schemes which do not consider the expected profit or loss, our model-guided scheme is capable of identifying


500
400
300
200
100
0
Number of page swaps
1
6
11
16
21
26
31
36
41
46
51
56
61
Epoch ID
ub.10
ub.100
ub.300
ub.500
model

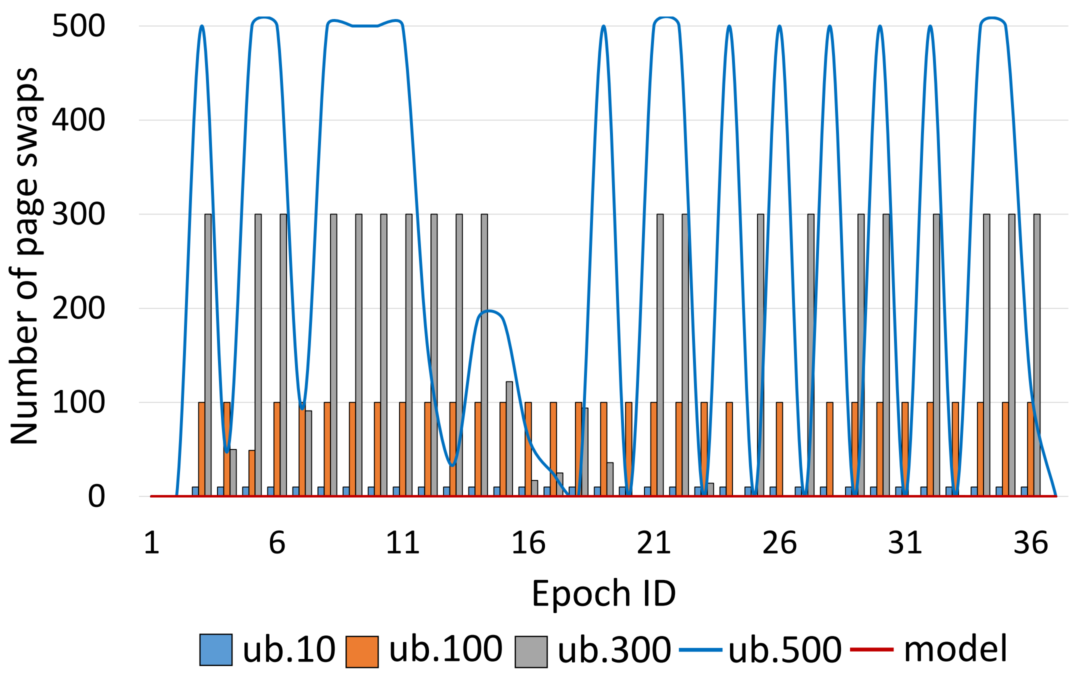
(a) mix1


500
400
300
200
100
0
Number of page swaps
1
6
11
16
21
26
Epoch ID
ub.10
ub.100
ub.300
ub.500
model

(b) mix6



(c) mix10



(d) mix11

Figure 32: Number of page swaps over epoch.

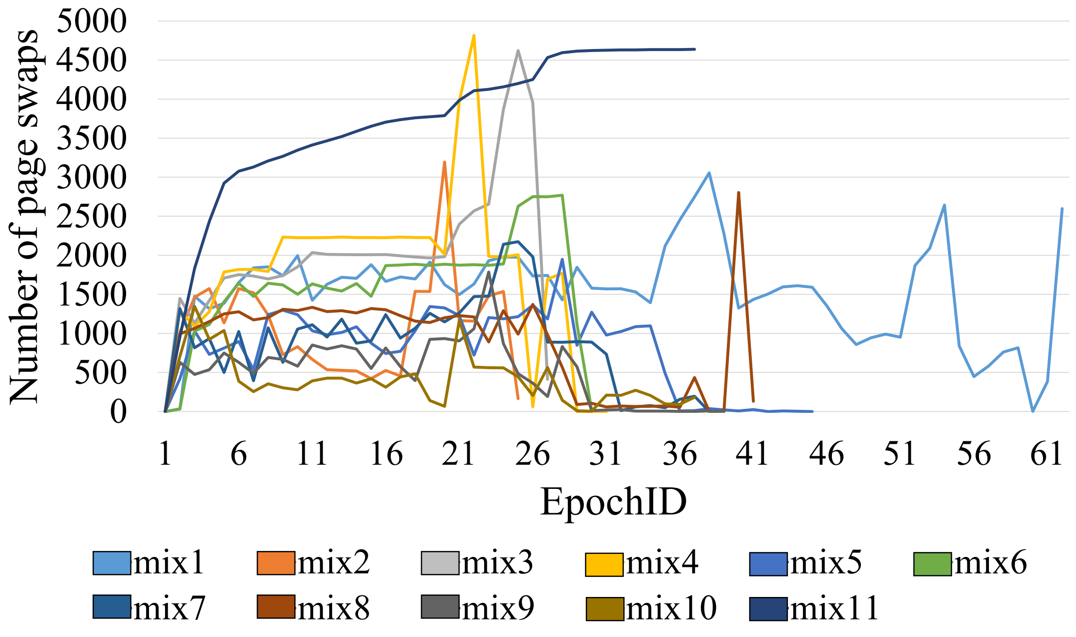


Figure 33: Number of page swaps over epoch with RaPP [94].

and suppressing detrimental page swaps. As a result, the number of page swaps with our model-guided scheme dramatically decreases by 17.3× over the *ub.500*. For NVRAMs with the limited write endurance, this is another important factor to consider. Meanwhile, it is notable that RaPP scheme outperforms our model-guided scheme by 38.6%. Figure 33 shows that RaPP scheme always generates high volume of page swaps throughout `mix11`'s entire execution, and on average, executes 3.68K page swaps per epoch. On the other hand, other schemes including the model-guided scheme have a constraint that they can do only maximum 500 page swaps per epoch. The upper-limit of 500 page swaps per epoch prevents these schemes from achieving the performance improvement. In addition, it is another factor for them to underperform RaPP that they perform page swap following a "monitoring-and-swap" rule which initiate page swaps at the end of an epoch. We expect that having a shorter epoch than 12.8M instructions as well as a broader range of the number of page swaps to select will ameliorate this performance issue for `mix11`.

In summary, our model-guided scheme outperforms both static schemes and the existing RaPP scheme by 13.3% and 26.1%, respectively by filtering out non-profitable page swaps.

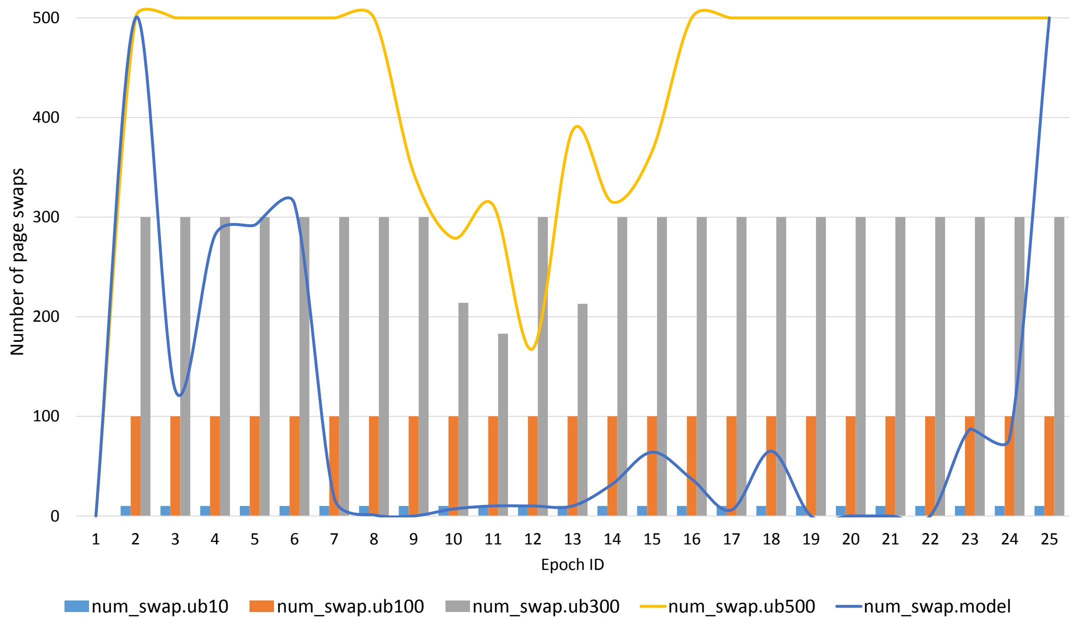
4.5 RELATED WORK

Ramos et al. [94] studied page ranking and migration policies in a hybrid memory. Using hardware multi-queues (MQ), they ranked pages based on the access frequency and decaying period, and moved a page to another region when it reached a specific MQ level, which represents a migration threshold. However, the overhead of extra traffic by page migrations has not been addressed in their works, and various heuristic (so, hard to tune) knobs governs a process of deciding a page decaying interval and suppressing a harmful migration. Unlike their work, our approach considers the overhead of the swap-induced extra requests, and does not rely on tunable parameters for page swap decision.

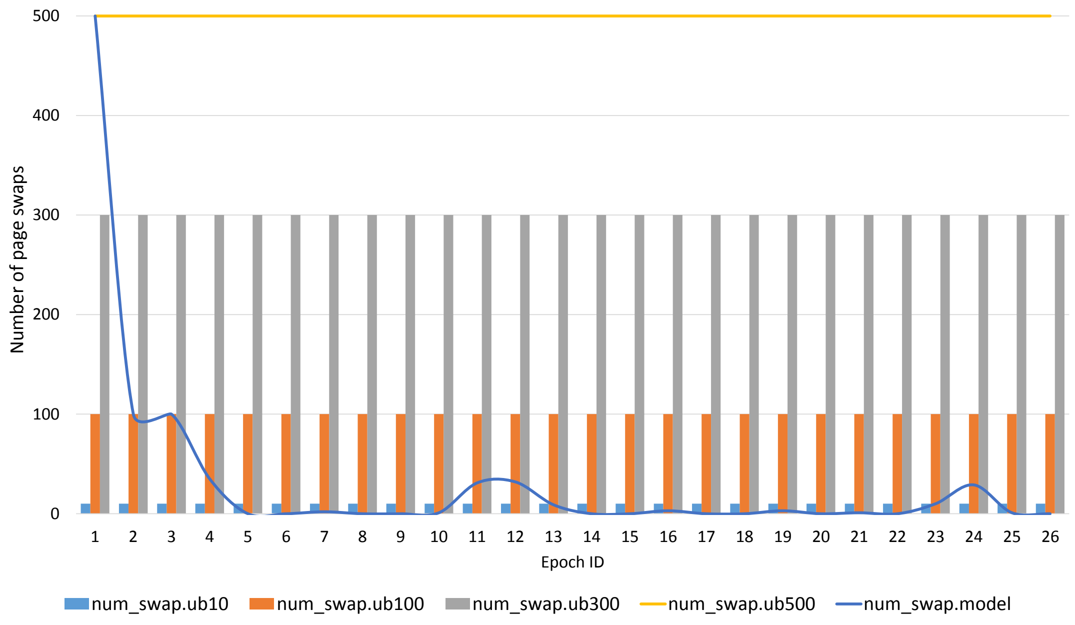
Pavlovic et al. [88] studied the effect of data placement and migration policies in HPC architectures with a hybrid memory. By running HPC applications whose memory footprints were much larger than ours, they made a similar observation to ours that a DRAM+PCM hybrid memory with a dynamic migration policy was several dozen percentages slower than DRAM-only systems. However, they did not study why a hybrid memory system achieves a performance comparable to a DRAM-only system, while our work answers the question. In addition, they used a migration parameter whose value should be changed and tuned whenever the running application changes.

Shen et al. [102] investigated the data placement of parallel benchmarks in a software-managed hybrid memory, using an in-house profiling tool and emulator. A profiler named `DataPlacer` instrumented a program on every function call to collect a per-function memory access pattern, then used the information for guiding a data placement. However, a software-based profiling that requires the per-function instrumentation is inapplicable to programs whose source codes are not public. Different from their work, our hardware based approach is transparent to user applications and does not require any code modification.

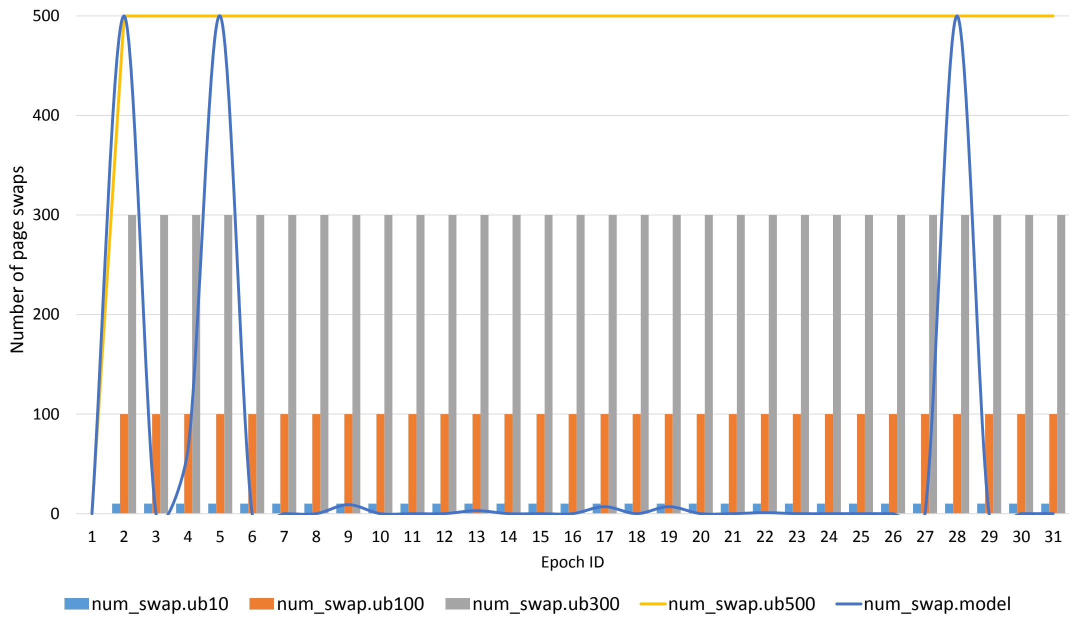
Bock et al. evaluate the impact of page migration in DRAM+PCM hybrid memory on system performance [15]. By breaking down various types of overhead in the memory hierarchy experienced by a memory request, they found the queuing delay at NVM banks as well as an external memory bus is a key limiting factor that hinders performance improvement by a software-managed hybrid memory. Different from our work, they assumed a software-


Number of page swaps
500
400
300
200
100
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
Epoch ID
num_swap.ub10
num_swap.ub100
num_swap.ub300
num_swap.ub500
num_swap.model

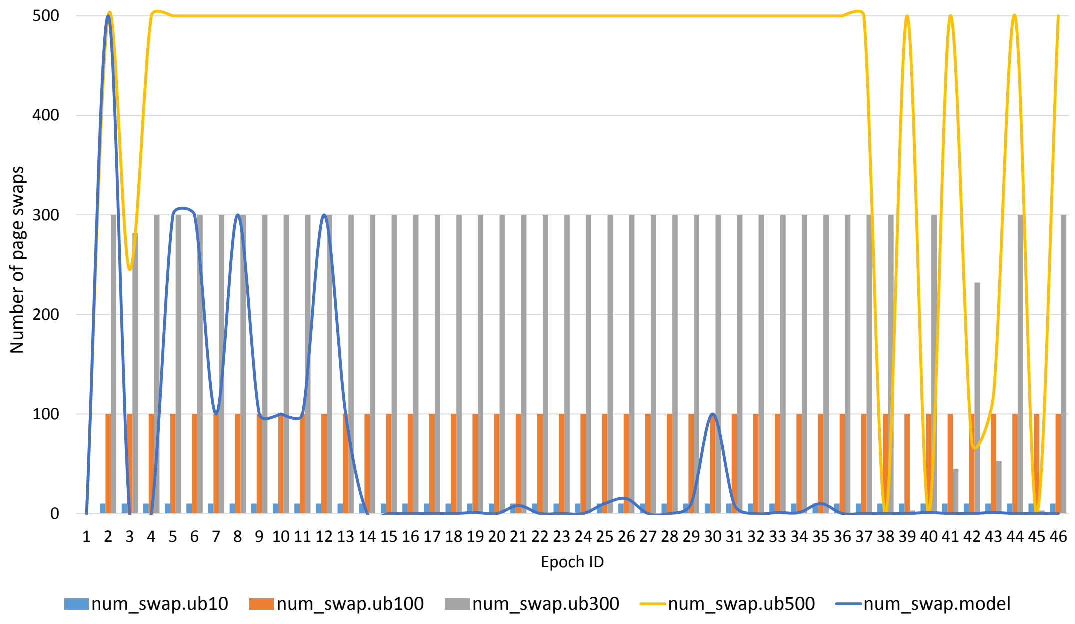
(a) mix2


Number of page swaps
500
400
300
200
100
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
Epoch ID
num_swap.ub10
num_swap.ub100
num_swap.ub300
num_swap.ub500
num_swap.model

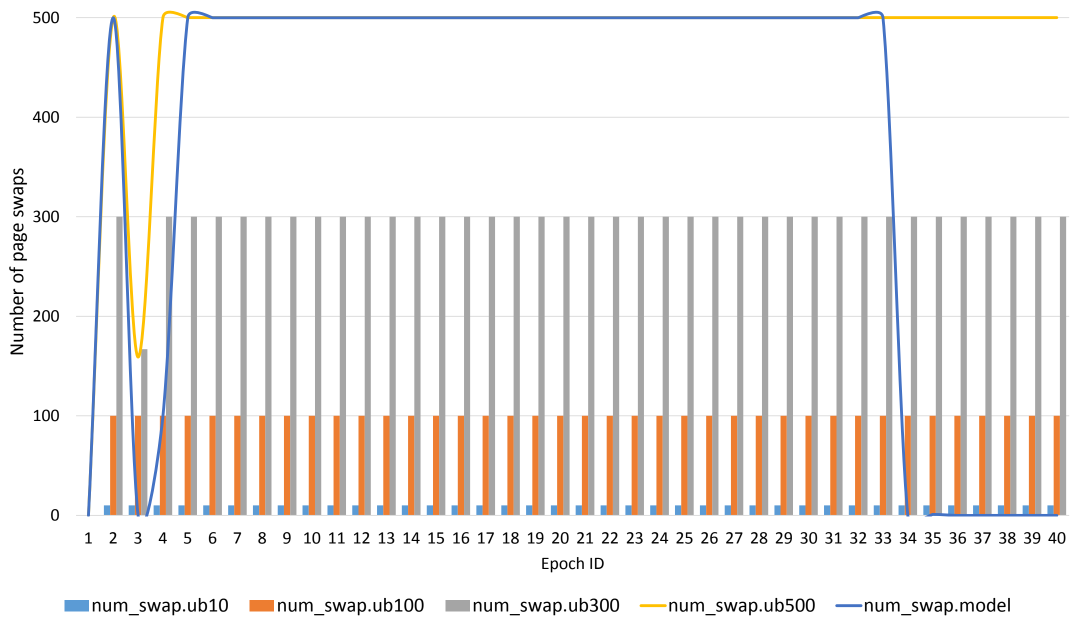
(b) mix3


500
400
300
200
100
0
Number of page swaps
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
Epoch ID
num_swap.ub10
num_swap.ub100
num_swap.ub300
num_swap.ub500
num_swap.model

(c) mix4

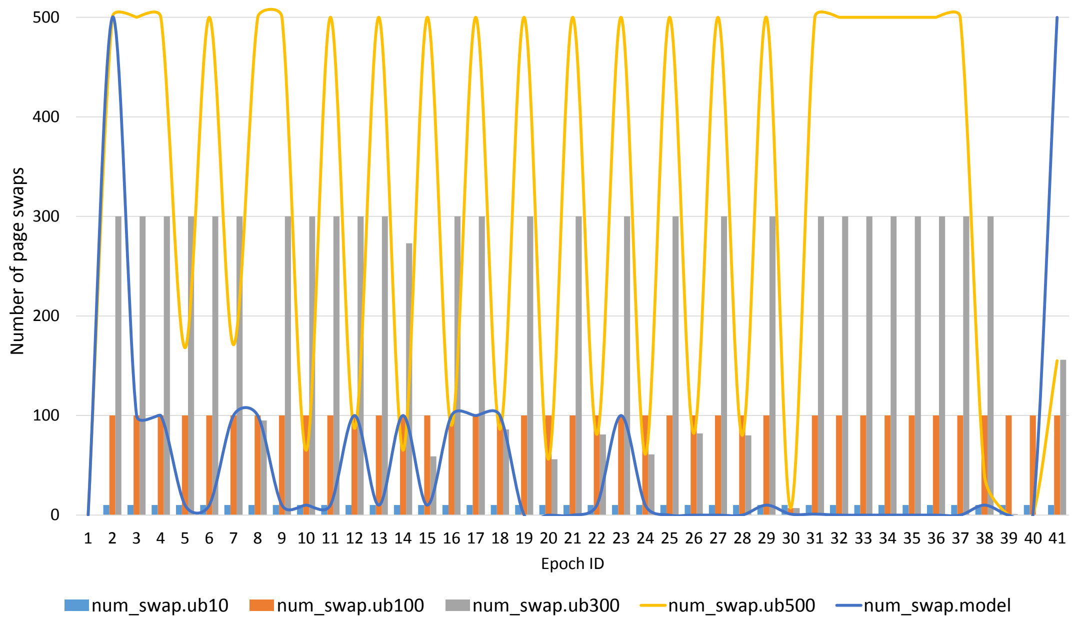

500
400
300
200
100
0
Number of page swaps
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
Epoch ID
num_swap.ub10
num_swap.ub100
num_swap.ub300
num_swap.ub500
num_swap.model

(d) mix5



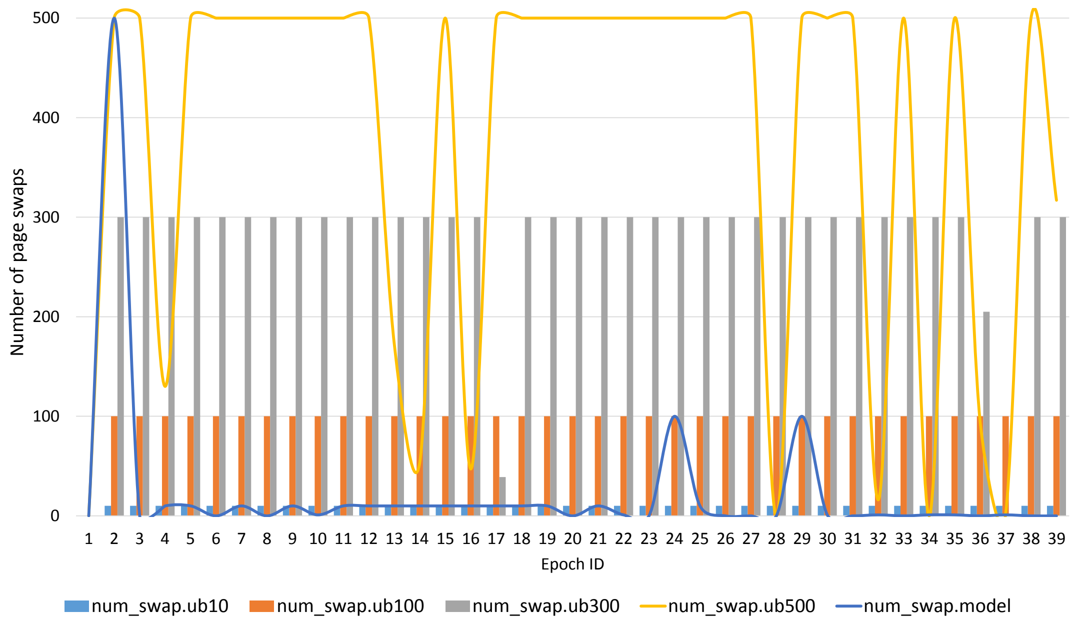
500
400
300
200
100
0
Number of page swaps
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40
Epoch ID
num_swap.ub10
num_swap.ub100
num_swap.ub300
num_swap.ub500
num_swap.model

(e) mix7



500
400
300
200
100
0
Number of page swaps
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41
Epoch ID
num_swap.ub10
num_swap.ub100
num_swap.ub300
num_swap.ub500
num_swap.model

(f) mix8



(g) mix9

Figure 34: Number of page swaps over epoch for other workloads.

managed hybrid memory which is dependent on OS-level page usage information, and they did not provide an exact analytical model to consider the queuing effect within a memory controller while making a page migration decision.

4.6 SUMMARY

In this work, we present an analytical model to investigate the effect of hardware-based page swap in a hybrid memory, and evaluate the effectiveness of a model-guided, hardware-driven page swap using the proposed model. Our approach to address a page swap in a hybrid memory is novel in that the effect of page swap is understood from a perspective of the flow control. Instead of just considering page references for a swap decision, the proposed model explicitly evaluates the profitability of a page swap at a system-level by observing the varying latency gap and traffic ratio between memory regions.

Our simulation results show that the proposed model is correlated to the architecture simulation as close as 90.9% with regard to the direction of AMAT variation. Meanwhile, the model-guided page swap improves IPC performance, on average, by 28.9% and by 13.3% over no page swapping and the studied static page swap schemes. In addition, the model-guided page swap considerably reduces the number of page swaps by up to 17.3× compared to the static schemes while achieving better performance. In summary, the model-guided, hardware-driven page swap is a viable choice for improving the performance of a system with a hybrid memory.

The following highlights some of our interesting findings.

- On a system that can run a memory-intensive workload in-memory, page reference history in OS's page LRU list is coarse-grained and a system should not rely on the software information to identify pages for a page swap.
- A page swap model built upon simple queuing theory can precisely appraise the profitability of a page swap, although the absolute CPI from the model may deviate from architecture simulation results. This is due to an accurate prediction of a performance direction change which enables assessing the profitability of a page swap.
- The access latency ratio of NVRAM to DRAM seen at program runtime [7] is much smaller than that shown at a device-level. This implies that naively considering a transistor-level latency ratio to evaluate the profitability of a page swap may result in detrimental page swaps at a system-level due to an incorrect expectation of attainable profits.
- The model-guided, hardware-driven page swap scheme which captures a dynamic variation of access latency and traffic distribution in a hybrid memory significantly outperforms static swap schemes as well as no swap (baseline) via its systematic and adaptive page swap decision.

[7]we call this runtime access latency ratio as a dynamic latency gap.

5.0 CONCLUSIONS AND FUTURE WORK

In this chapter, we summarize key contributions that this dissertation work made, and describes our future works.

5.1 CONCLUDING REMARKS

In this dissertation, we studied effective methods to integrate an emerging NVRAM into the memory hierarchy of an existing system platform. Since the main memory and secondary storage of current systems have not taken emerging NVRAM technologies into account, the lack of understanding the difference and uniqueness of NVRAMs causes the inefficient system designs.

To address the inefficiencies and fully exploit the advantages of emerging NVRAM technologies, we proposed and evaluated two new ideas, which are the Memorage architecture, and the model-guided, hardware-driven page swap mechanism.

The former improves the system performance by dynamically sharing emerging NVRAM resources across main memory and storage system when a system is under high memory pressure. Compared to the conventional methods to handle high memory pressure, our experimental results indicate that the proposed Memorage architecture not only achieves the large application performance improvement, but also considerably extends the system-level NVRAM lifetime. We expect that this work will spur researchers to reconsider the system-level implications of the relationship between memory and storage in future NVRAM-based systems. In addition, the relevant industries may benefit from our experimental methodology which allow them to evaluate NVRAM-based systems at the early system development stage.

Meanwhile, the later improves the performance of applications running with a hybrid main memory by judiciously swapping pages from the faster memory with those from the slower memory. We showed that examining the profitability of a page swap while making page swap decisions is very important to better utilize the faster memory and improve the performance. Also, we observed the a conventional OS is incapable of identifying hot/cold pages of memory-intensive applications, suggesting that the page classification for page swapping should be performed in hardware. We believe that our analytical model to evaluate the profitability of a page swap and our findings of page swap behaviors will inspire other researchers to address this in-memory page swap from a perspective of the system-level flow control, not the traditional approaches based only on page access frequency.

5.2 FUTURE WORK

In this dissertation, we have successfully evaluated the Memorage architecture through a working prototype system, and demonstrated that dynamic NVRAM resource sharings across the conventional memory hierarchy could considerably boost the application performance under heady memory pressure. However, the current prototype system can be further improved by introducing better policy modules in order to adaptively determine the capacity of the memory and storage, respectively. When a high-level policy maker to control Memorage operations has been integrated into Memorage architecture, a system administrator can more flexibly change the behavioral rules of Memorage (e.g. a ratio of capacity partitioning between memory and storage) according to a recent system usage pattern. Therefore, we plan to conduct a research on Memorage's policy module.

As for in-memory page swap, we have dealt with only anonymous pages allocated by the non-file oriented applications such as SPEC CPU2006 to evaluate our hardware-driven in-memory page swap in a hybrid main memory. However, the OS VMM manages not only anonymous pages but also file pages. These file object-oriented applications which frequently issue access requests to NVRAM storage devices will have different page reference patterns from that of heap and stack-oriented applications. Hence, a future work is to investigate the potential of our model-guided page swap for those storage applications. In addition, the proposed models can be extended to consider the impact of page swaps between heap and file pages on system-wide performance.

Lastly, we plan to explore the value of Memorage architecture and in-memory page swap by running recent commodity applications on a large-scale system. For example, it will be interesting to evaluate how the studied system-level techniques influence the performance of IMDB (In-Memory Database) or Spark's clustered data processing applications.

We expect this dissertation work to inspire other researches on the integration of various emerging NVRAM technologies into current NVRAM-unaware memory hierarchy, and we aspire to see our proposed architecture/techniques applied to the design of a future commodity computing system.

BIBLIOGRAPHY

[1] Linux memory hotplug. http://tinyurl.com/memory-hotplug, 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://tinyurl.com/micron-PCM, 2012.

[4] Nidhi Aggarwal, Jason F. Cantin, Mikko H. Lipasti, and James E. Smith. Power-efficient DRAM speculation. *HPCA*, pages 317–328, 2008.

[5] Nitin Agrawal, William Bolosky, John Douceur, and Jacob Lorch. A five-year study of file-system metadata. *TOS*, 3(3):1553–3077, October 2007.

[6] Ameen Akel, Adrian M. Caulfield, Todor I. Mollov, Rajesh K. Gupta, and Steven Swanson. Onyx: A protoype phase-change memory storage array. *HotStorage*, pages 2–2, 2011.

[7] A. Athmanathan and et al. Multilevel-cell phase-change memory: A viable technology. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 6(1):87–100, March 2016.

[8] Sven Auhagen, Lars Bergstrom, Matthew Fluet, and John Reppy. Garbage collection for multicore numa machines. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 51–57, New York, NY, USA, 2011. ACM.

[9] Anirudh Badam and Vivek Pai. Ssdalloc: hybrid ssd/ram memory management made easy. *NSDI*, pages 16–16, 2011.

[10] Luiz Andr Barroso, Jimmy Clidaras, and Urs Hlzle. *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. 2013.

[11] Scott Beamer, Krste Asanovic, and David Patterson. Locality exists in graph processing: Workload characterization on an ivy bridge server. In *Proceedings of the 2015 IEEE International Symposium on Workload Characterization*, IISWC '15, pages 56–65, Washington, DC, USA, 2015. IEEE Computer Society.

[12] F. Bedeschi, R. Fackenthal, C. Resta, E. M. Donze, M. Jagasivamani, E. C. Buda, F. Pellizzer, D. W. Chow, A. Cabrini, G. Calvi, R. Faravelli, A. Fantini, G. Torelli, D. Mills, R. Gastaldi, and G. Casagrande. A bipolar-selected phase change memory featuring multi-level cell storage. *JSSC*, 44:217–227, 2009.

[13] I. Bhati, M. T. Chang, Z. Chishti, S. L. Lu, and B. Jacob. Dram refresh mechanisms, penalties, and trade-offs. *IEEE Transactions on Computers*, 65(1):108–121, Jan 2016.

[14] Tim Bird. Measuring function duration with Ftrace. *Japan Linux Symposium*, 2009.

[15] S. Bock, B. R. Childers, R. Melhem, and D. Mosse. Characterizing the overhead of software-managed hybrid main memory. In *Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS), 2015 IEEE 23rd International Symposium on*, pages 33–42, Oct 2015.

[16] Santiago Bock, Bruce R. Childers, Rami Melhem, and Daniel Mossé. Concurrent page migration for mobile systems with os-managed hybrid memory. In *Proceedings of the 11th ACM Conference on Computing Frontiers*, CF '14, pages 31:1–31:10, New York, NY, USA, 2014. ACM.

[17] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[18] Ravi Budruk, Don Anderson, and Ed Solari. *PCI Express System Architecture*. Pearson Education, 2003.

[19] Adrian Caulfield, Arup De, Joel Coburn, Todor I. Mollow, Rajesh K. Gupta, and Steven Swanson. Moneta: A high-performance storage array architecture for next-generation, non-volatile memories. *MICRO*, pages 385–395, 2010.

[20] Adrian Caulfield, Todor Mollov, Louis Alex Eisner, Arup De, Joel Coburn, and Steven Swanson. Providing safe, user space access to fast, solid state disks. *ASPLOS*, pages 387–400, 2012.

[21] R. Chen, Z. Shao, C. L. Yang, and Tao Li. Mcssim: A memory channel storage simulator. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 153–158, Jan 2016.

[22] Yiran Chen, Weng-Fai Wong, Hai Li, and Cheng-Kok Koh. Processor caches with multi-level spin-transfer torque ram cells. *ISLPED*, pages 73–78, 2011.

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

[24] Youngdon Choi, Ickhyun Song, Mu-Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, Younghoon Oh, Dukmin Kwon, Jung Sunwoo, Junho Shin, Yoohwan Rho, Changsoo Lee, Min Gu Kang, Jaeyun Lee, Yongjin Kwon, Soehee Kim,

Jaewhan Kim, Yong jun Lee, Qi Wang, Sooho Cha, Sujin Ahn, Hideki Horii, Jaewook Lee, KiSeung Kim, Han-Sung Joo, KwangJin Lee, Yeong-Taek Lee, Jei-Hwan Yoo, and Gitae Jeong. A 20nm 1.8V 8gb PRAM with 40MB/s program bandwidth. *ISSCC*, pages 46–48, 2012.

[25] Chia-Chen Chou, Aamer Jaleel, and Moinuddin K. Qureshi. Batman: Maximizing bandwidth utilization for hybrid memory systems. In *Proceedings of the 42th Annual International Symposium on Computer Architecture*, ISCA '15, pages 72–83, New York, NY, USA, 2015. ACM.

[26] Chih-Hsun Chou, Daniel Wong, and Laxmi N. Bhuyan. Dynsleep: Fine-grained power management for a latency-critical data center application. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, ISLPED '16, pages 212–217, New York, NY, USA, 2016. ACM.

[27] G. F. Close, U. Frey, J. Morrish, R. Jordan, S. Lewis, T. Maffitt, M. Breitwisch, C. Hagleitner, C. Lam, and E. Eleftheriou. A 512mb phase-change memory (pcm) in 90nm cmos achieving 2b/cell. *VLSIC*, pages 202–203, 2011.

[28] Elliott Cooper-Balis, Paul Rosenfeld, and Bruce Jacob. Buffer-on-board memory systems. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, pages 392–403, Washington, DC, USA, 2012. IEEE Computer Society.

[29] Dell. Solid state drive vs. hard disk drive price and performance study. http://tinyurl.com/Dell-SSD-Price-Performance, 2011.

[30] Advanced Micro Devices. High bandwidth memory, reinventing memory technology. http://www.amd.com/en-us/innovations/software-technologies/hbm, 2015. (Last accessed October 12th, 2016).

[31] Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. Pdram: a hybrid pram and dram main memory system. In *Proceedings of the 46th Annual Design Automation Conference*, DAC '09, pages 664–469, New York, NY, USA, 2009. ACM.

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

[33] P. G. Emma. Understanding some simple processor-performance limits. *IBM J. Res. Dev.*, 41(3):215–232, May 1997.

[34] A. Foong and F. Hady. Storage as fast as rest of the system. In *2016 IEEE 8th International Memory Workshop (IMW)*, pages 1–4, May 2016.

[35] International Technology Roadmap for Semiconductors. 2011 edition. http://public.itrs.net, 2011.

[36] International Technology Roadmap for Semiconductors. 2013 edition. http://public.itrs.net, 2013.

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

[38] F. Furuta and K. Osada. 6 tbps/w, 1 tbps/mm2, 3d interconnect using adaptive timing control and low capacitance tsv. In *3D Systems Integration Conference (3DIC), 2011 IEEE International*, pages 1–4, Jan 2012.

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

[40] M. Gries, U. Hoffmann, M. Konow, and M. Riepen. Scc: A flexible architecture for many-core platform research. *Computing in Science Engineering*, 13(6):79–83, Nov 2011.

[41] Laura Grupp, Adrian Caulfield, Joel Coburn, Steven Swanson, Eitan Yaakobi, Paul H. Siegel, and Jack K. Wolf. Characterizing flash memory: anomalies, observations, and applications. *MICRO*, pages 24–33, 2009.

[42] Tae Jun Ham, Bharath K. Chelepalli, Neng Xue, and Benjamin C. Lee. Disintegrated control for energy-efficient and heterogeneous memory systems. In *Proceedings of the 2013 19th IEEE International Symposium on High-Performance Computer Architecture*, HPCA '13, pages 101–112, Shenzhen, China, 2013. IEEE Computer Society.

[43] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2011.

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

[45] J. Howard, S. Dighe, Y. Hoskote, S. Vangal, D. Finan, G. Ruhl, D. Jenkins, H. Wilson, N. Borkar, G. Schrom, F. Pailet, S. Jain, T. Jacob, S. Yada, S. Marella, P. Salihundam, V. Erraguntla, M. Konow, M. Riepen, G. Droege, J. Lindemann, M. Gries, T. Apel, K. Henriss, T. Lund-Larsen, S. Steibl, S. Borkar, V. De, R. V. D. Wijngaart, and T. Mattson. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *2010 IEEE International Solid-State Circuits Conference - (ISSCC)*, pages 108–109, Feb 2010.

[46] H. F. Huang and T. Jiang. Design and implementation of flash based nvdimm. In *Non-Volatile Memory Systems and Applications Symposium (NVMSA), 2014 IEEE*, pages 1–6, Aug 2014.

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

[48] J. W. Im, W. P. Jeong, D. H. Kim, S. W. Nam, D. K. Shim, M. H. Choi, H. J. Yoon, D. H. Kim, Y. S. Kim, H. W. Park, D. H. Kwak, S. W. Park, S. M. Yoon, W. G. Hahn, J. H. Ryu, S. W. Shim, K. T. Kang, S. H. Choi, J. D. Ihm, Y. S. Min, I. M. Kim, D. S. Lee, J. H. Cho, O. S. Kwon, J. S. Lee, M. S. Kim, S. H. Joo, J. H. Jang, S. W. Hwang, D. S. Byeon, H. J. Yang, K. T. Park, K. H. Kyung, and J. H. Choi. 7.2 a 128gb 3b/cell v-nand flash memory with 1gb/s i/o rate. In *2015 IEEE International Solid-State Circuits Conference - (ISSCC) Digest of Technical Papers*, pages 1–3, Feb 2015.

[49] INCITS. Technical Committee T10 SCSI Storage Interfaces. http://www.t10.org, 2013.

[50] Intel. Intel memory latency checker v3.1a. https://software.intel.com/en-us/articles/intelr-memory-latency-checker, October 2016. (Last accessed October 23rd, 2016).

[51] Intel. Persistent memory programming. http://pmem.io, October 2016. (Last accessed October 23rd, 2016).

[52] Intel and Micron. Intel and micron produce breakthrough memory technology. http://tinyurl.com/3DXpoint2015, July 2015.

[53] J. Jeddeloh and B. Keeth. Hybrid memory cube new dram architecture increases density and performance. In *VLSI Technology (VLSIT), 2012 Symposium on*, pages 87–88, 2012.

[54] William Josephson, Lars Bongo, Kai Li, and David Flynn. Dfs: A file system for virtualized flash storage. *TOS*, 6(3):14:1–14:25, September 2010.

[55] Madhura Joshi, Wangyuan Zhang, and Tao Li. Mercury: A fast and energy-efficient multi-level cell based phase change memory system. *HPCA*, pages 345–356, 2011.

[56] Ju-Young Jung and Sangyeun Cho. Dynamic co-management of persistent ram main memory and storage resources. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 13:1–13:2, New York, NY, USA, 2011. ACM.

[57] Ju-Young Jung and Sangyeun Cho. Prism: Zooming in persistent ram storage behavior. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 22–31, April 2011.

[58] Ju-Young Jung and Sangyeun Cho. Memorage: Emerging persistent ram based malleable main memory and storage architecture. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 115–126, New York, NY, USA, June 2013. ACM.

[59] Ju-Young Jung and Rami Melhem. Empirical, analytical study of hardware-based page swap in hybrid main memory system. In *Computer Architecture and High Performance*

Computing (SBAC-PAD), 2016 28th International Symposium on, pages 82–89, Oct 2016.

[60] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. Profiling a warehouse-scale computer. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 158–169, New York, NY, USA, 2015. ACM.

[61] Kimberly Keeton. The machine: An architecture for memory-centric computing. In *Proceedings of the 5th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '15, pages 1:1–1:1, New York, NY, USA, 2015. ACM.

[62] Dong Kim, Kwanhu Bang, Seung-Hwan Ha, Sungroh Yoon, and Eui-Young Chung. Architecture exploration of high-performance PCs with a solid-state disk. *Trans. Computers*, 59(7):878–890, 2010.

[63] Y. Kim, W. Yang, and O. Mutlu. Ramulator: A fast and extensible dram simulator. *Computer Architecture Letters*, PP(99):1–1, 2015.

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

[65] Mark Kryder and ChangSoo Kim. After hard drives–what comes next? *Trans. Magnetics*, 45:3406 – 3413, 2009.

[66] Karthik Kumar, Kshitij Doshi, Martin Dimitrov, and Yung-Hsiang Lu. Memory energy management for an enterprise decision support system. In *Proceedings of the 17th IEEE/ACM international symposium on Low-power electronics and design*, ISLPED '11, pages 277–282, Piscataway, NJ, USA, 2011. IEEE Press.

[67] Stefan Lankes, Boris Bierbaum, and Thomas Bemmerl. Affinity-on-next-touch: An extension to the linux kernel for numa architectures. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I*, PPAM'09, pages 576–585, Berlin, Heidelberg, 2010. Springer-Verlag.

[68] Benjamin Lee, Engin Ipek, Onur Mutlu, and Doug Burger. Architecting phase change memory as a scalable DRAM alternative. *ISCA*, June 2009.

[69] Kevin Lim, Jichuan Chang, Trevor Mudge, Parthasarathy Ranganathan, Steven K. Reinhardt, and Thomas F. Wenisch. Disaggregated memory for expansion and sharing in blade servers. *SIGARCH Comput. Archit. News*, 37(3):267–278, June 2009.

[70] John D. C. Little. Or forum—little's law as viewed on its 50th anniversary. *Oper. Res.*, 59(3):536–549, May 2011.

[71] T. Y. Liu and et al. A 130.7mm2 2-layer 32gb reram memory device in 24nm technology. In *2013 IEEE International Solid-State Circuits Conference Digest of Technical Papers*, ISSCC '13, pages 210–211, Feb 2013.

[72] Wenjie Liu, Ping Huang, Kun Tang, Ke Zhou, and Xubin He. Car: A compression-aware refresh approach to improve memory performance and energy efficiency. *SIGMETRICS Perform. Eval. Rev.*, 44(1):373–374, June 2016.

[73] X. Liu, D. Buono, F. Checconi, J. W. Choi, X. Que, F. Petrini, J. A. Gunnels, and J. A. Stuecheli. An early performance study of large-scale power8 smp systems. In *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 263–272, May 2016.

[74] Robert Love. In *Linux Kernel Development, 2nd Edition*. Novell Press, 2005.

[75] Jason Lowe-Power, Mark D. Hill, and David A. Wood. When to use 3d die-stacked memory for bandwidth-constrained big data workloads. *CoRR*, abs/1608.07485, 2016.

[76] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[77] Wolfgang Mauerer. In *Professional Linux Kernel Architecture*. Wrox, 2008.

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

[79] Lucas Mearian. Flash memory's density surpasses hard drives for first time. http://tinyurl.com/FlashDensitySurpassHDD.

[80] David Meisner, Brian T. Gold, and Thomas F. Wenisch. Powernap: eliminating server idle power. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 205–216, New York, NY, USA, 2009. ACM.

[81] Jie Meng and A.K. Coskun. Analysis and runtime management of 3d systems with stacked dram for boosting energy efficiency. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2012*, pages 611–616, 2012.

[82] Dutch Meyer and William Bolosky. A study of practical deduplication. *TOS*, 7(4):14, 2012.

[83] H. Midorikawa, H. Tan, and T. Endo. An evaluation of the potential of flash ssd as large and slow memory for stencil computations. In *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pages 268–277, July 2014.

[84] G. E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[85] D. Nagaraj and C. Gianos. Intel xeon processor d: The first xeon processor optimized for dense solutions. In *2015 IEEE Hot Chips 27 Symposium (HCS)*, pages 1–22, Aug 2015.

[86] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguade. User-level dynamic page migration for multiprogrammed shared-memory multiprocessors. In *Parallel Processing, 2000. Proceedings. 2000 International Conference on*, pages 95–103, 2000.

[87] S. Park, A. Wang, U. Ko, L. S. Peh, and A. P. Chandrakasan. Enabling simultaneously bi-directional tsv signaling for energy and area efficient 3d-ics. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 163–168, March 2016.

[88] M. Pavlovic, N. Puzovic, and A. Ramirez. Data placement in hpc architectures with heterogeneous off-chip memory. In *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pages 193–200, 2013.

[89] Erez Perelman, Greg Hamerly, Michael Van Biesbrouck, Timothy Sherwood, and Brad Calder. Using simpoint for accurate and efficient simulation. *SIGMETRICS Perform. Eval. Rev.*, 31(1):318–319, June 2003.

[90] H. Pozidis, N. Papandreou, A. Sebastian, A. Pantazi, T. Mittelholzer, G. F. Close, and E. Eleftheriou. Enabling technologies for multi-level phase change memory. *E\PCOS*, 2011.

[91] Moinuddin Qureshi, Michele Franceschini, Luis Alfonso Lastras-Montano, and John P. Karidis. Morphable memory system: A robust architecture for exploiting multi-level phase change memories. *ISCA*, pages 153–162, June 2010.

[92] Moinuddin Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. Scalable high performance main memory system using phase-change memory technology. *ISCA*, June 2009. IBM T.J. Watson RC.

[93] Moinuddin K. Qureshi, Dae-Hyun Kim, Samira Khan, Prashant J. Nair, and Onur Mutlu. Avatar: A variable-retention-time (vrt) aware refresh for dram systems. In *Proceedings of the 2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, DSN '15, pages 427–437, Washington, DC, USA, 2015. IEEE Computer Society.

[94] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page placement in hybrid memory systems. In *Proceedings of the international conference on Supercomputing*, ICS '11, pages 85–95, New York, NY, USA, 2011. ACM.

[95] Simone Raoux, Geoffrey W. Burr, Matthew J. Breitwisch, Charles T. Rettner, Yi-Chou Chen, Robert M. Shelby, Martin Salinga, Daniel Krebs, Shih-Hung Chen, Hsiang-Lan Lung, and Chung Hon Lam. Phase-change random access memory: A scalable technology. *IBM J. of R & D*, 52(4-5):465–480, 2008.

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

[97] Samsung. Samsung starts mass producing industry's first 128-gigabyte ddr4 modules for enterprise servers. http://www.samsung.com/semiconductor/about-us/news/24441, November 2015.

[98] SATA-IO. The Serial ATA International Organization. http://www.sata-io.org, 2013.

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

[100] Eric Seppanen, Matthew O'Keefe, and David Lilja. High performance solid state storage under linux. *MSST*, pages 1–12, 2010.

[101] J. M. Shalf and R. Leland. Computing beyond moore's law. *Computer*, 48(12):14–23, Dec 2015.

[102] Du Shen, Xu Liu, and Felix Xiaozhu Lin. Characterizing emerging heterogeneous memory. In *Proceedings of the 2016 ACM SIGPLAN International Symposium on Memory Management*, ISMM 2016, pages 13–23, New York, NY, USA, 2016.

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

[104] A. Sodani, R. Gramunt, J. Corbal, H. S. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. C. Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, Mar 2016.

[105] Cloyce D. Spradling. Spec cpu2006 benchmark tools. *SIGARCH Comput. Archit. News*, 35(1):130–134, March 2007.

[106] Dmitri Strukov, Gregory Snider, Duncan Stewart, and Stanley Williams. The missing memristor found. *Nature*, 453(7191):80–83, 2008.

[107] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Hardware monitors for dynamic page migration. *J. Parallel Distrib. Comput.*, 68(9):1186–1200, September 2008.

[108] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. Didi: Mitigating the performance impact of tlb shootdowns using a shared tlb directory. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on*, pages 340–349, Oct 2011.

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

[110] D. Wei, L. Deng, L. Qiao, P. Zhang, and X. Peng. Peva: A page endurance variance aware strategy for the lifetime extension of nand flash. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(5):1749–1760, May 2016.

[111] Z Wei, Y Kanzawa, K Arita, Y Katoh, K Kawai, S Muraoka, S Mitani, S Fujii, K Katayama, M Iijima, , and et al. Highly reliable taox reram and direct evidence of redox reaction mechanism. *IEDM*, pages 1–4, 2008.

[112] NVM Express Workgroup. Nvm express - the optimized pci express ssd interface. http://www.nvmexpress.org/, June 2016. (Last accessed October 23rd, 2016).

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

[114] Yun-Chao Yu, Chih-Sheng Hou, Li-Jung Chang, Jin-Fu Li, Chih-Yen Lo, Ding-Ming Kwai, Yung-Fa Chou, and Cheng-Wen Wu. A hybrid ecc and redundancy technique for reducing refresh power of drams. In *VLSI Test Symposium (VTS), 2013 IEEE 31st*, pages 1–6, 2013.

[115] Wangyuan Zhang and Tao Li. Exploring phase change memory and 3d die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, PACT '09, pages 101–112, Washington, DC, USA, 2009. IEEE Computer Society.

[116] Yuang Zhang, Li Li, Zhonghai Lu, Axel Jantsch, Minglun Gao, Hongbing Pan, and Feng Han. A survey of memory architecture for 3d chip multi-processors. *Microprocess. Microsyst.*, 38(5):415–430, July 2014.

[117] Jishen Zhao, Sheng Li, Jichuan Chang, John L. Byrne, Laura L. Ramirez, Kevin Lim, Yuan Xie, and Paolo Faraboschi. Buri: Scaling big-memory computing with hardware-based memory expansion. *ACM Trans. Archit. Code Optim.*, 12(3):31:1–31:24, October 2015.

[118] Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. A durable and energy efficient main memory using phase change memory technology. *ISCA*, 2009.