RAMinate: Hypervisor-based Virtualization for Hybrid Main Memory Systems

Takahiro Hirofuchi and Ryousei Takano

National Institute of Advanced Industrial Science and Technology (AIST)

t.hirofuchi@aist.go.jp

Abstract

In the future, STT-MRAM will achieve larger capacity and comparable read/write performance, but incur orders of magnitude greater write energy than DRAM. To achieve large capacity as well as energy-efficiency, it is necessary to use both DRAM and STT-MRAM for the main memory of a computer. In this paper, we propose a hypervisor-based hybrid memory mechanism (RAMinate) that reduces write traffic to STT-MRAM by optimizing page locations between DRAM and STT-MRAM. In contrast to past studies, our mechanism works at the hypervisor level, not at the hardware or operating system level. It does not require any special program at the operating system level nor any design changes of the current memory controller at the hardware level. We developed a prototype of the proposed system by extending Qemu/KVM and conducted experiments with application benchmarks. We confirmed that our page replacement mechanism successfully worked for unmodified operating systems and dynamically diverted memory write traffic to DRAM. Our experiments also confirmed that our system successfully reduced write traffic to STT-MRAM by approximately 70% for tested workloads, which results in a 50% reduction in energy consumption in comparison to a DRAM-only system.

Categories and Subject Descriptors D.4 [*Operating System*]: Storage Management—Main memory; B.3 [*Memory Structure*]: Design Styles

Keywords Hypervisor, Hybrid Memory, Virtual Machine, Non-volatile Memory, STT-MRAM

1. Introduction

Non-volatile memory (NVM) technologies[15, 26] are indispensable for future data centers. Server computer systems equipped with many-core processors need larger memory capacity so as to fully utilize processor resources. However, current and future DRAM technology is unlikely able to meet this growing memory demand. Instead, upcoming NVM technologies, now being intensively developed in the industry and academia, will play a key role in future computer systems.

Although DRAM is widely used for the main memory of computer systems, it is considered among researchers no longer scalable because the fundamental mechanism of a DRAM cell is a capacitor. A DRAM cell must be periodically refreshed to keep its data. Thus, as the capacity of a DRAM chip increases so does the energy consumption to refresh the memory cells. Current memory subsystems using DRAM contribute to more than 40% of the energy consumption of server computers[13]. Although in the past decades manufacturers succeeded in increasing the density of DRAM chips, reducing a cell size is becoming more difficult due to leak energy problems. As the scale of memory cells is reduced, more energy dissipation is inevitable. Leak energy, which raises the temperature of a chip, destabilizes electrical charge in memory cells.

In contrast to DRAM, NVM technologies will achieve more capacity with less energy consumption. These memory cells can maintain data without refresh operations. Manufacturers will be able to fabricate high-density NVM chips using a small scale manufacturing process. Among NVM technologies in the future around 2026, we believe that STT-MRAM (Spin Transfer Torque Magnetoresistive RAM) will have the greatest impact on the memory subsystems of computers. According to the technology roadmap forecasted by the semiconductor industry[23], STT-MRAM will achieve the same level of read/write latency as DRAM (both $<$ 10ns). Additionally, in contrast to the write endurance problems of PCRAM (Phase Change RAM or Phase Change Memory), STT-MRAM can rewrite memory cells without any practical degradation. A memory cell of STT-MRAM will accept rewrite operations more than 10^{15} times, while PCRAM will accepts only 10^{9}. Thus, given that the number of rewrite operations for STT-MRAM is a perfectly feasible number for main memory or CPU cache, STT-MRAM has the potential of serving as the main memory of computer systems like DRAM does today.

SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
© 2016 ACM. ISBN 978-1-4503-4525-5/16/10. . . $15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987570

However, one disadvantage of STT-MRAM is that the write energy will be 10^2 times larger than that of DRAM. Future STT-MRAM write energy is estimated at $1.5 * 10^{-13}$ J/bits while that of DRAM is $2 * 10^{-15}$ J/bits[23]. Just replacing all DRAM with STT-MRAM will likely increase the energy consumption of a memory subsystem. To address this problem a hybrid memory system that combines both large STT-MRAM modules and small DRAM modules for the main memory of a computer is necessary. To avoid large write energy of STT-MRAM, the system places memory pages having many write operations on DRAM. Although prior studies discussed the designs of hybrid memory systems, software-based mechanisms[3, 4, 6, 9, 25] need to update operating systems or application programs, and hardware-based mechanisms[5, 18, 19, 27] need to modify the memory subsystem of computers. Both approaches are not transparent to existing software and hardware components, which will incur huge migration costs to take advantage of emerging NVM technologies.

In this paper, we propose a hypervisor-based virtualization mechanism for hybrid memory systems, *RAMinate*. It dynamically optimizes memory mappings from guest physical pages to host physical pages, to reduce the energy consumption of the memory subsystem of an IaaS system. From the viewpoint of a virtual machine (VM), its RAM is virtually composed of a uniform type of memory (i.e., DRAM). In reality, guest memory pages are physically mapped to DRAM or STT-MRAM pages. RAMinate periodically detects write intensive pages in the guest RAM, and dynamically relocates them to DRAM on the fly without restarting the VM, taking into account the memory access trend of each page. Importantly, RAMinate works at the hypervisor level, not at the hardware or guest level, and does not require any special program at the guest level or changes in the design of the current memory controller at the hardware level.

We developed a prototype of the proposed mechanism by extending a widely-used hypervisor, Qemu/KVM[10]. We implemented 1) a lightweight trace mechanism of guest memory access that has no noticeable performance impact on VMs, 2) a page placement algorithm that dynamically optimizes page mappings, and 3) a page migration mechanism that swaps DRAM and STT-MRAM pages transparently to the running VM. To evaluate the hybrid memory system, which is not yet available on the market, we also developed 4) a new technique of performance evaluation using hardware performance monitoring units of recent CPUs. Thus, our evaluation was through our working prototype and not simulation-based.

To our knowledge, this is the first work fully implementing a hypervisor-based hybrid memory system. In the following parts of the paper, we first clarify the motivation and background of this project in Section 2. We detail the design of RAMinate in Section 3, and then evaluate it through various experiments in Section 4. In Section 5, we summarize related work. Finally, we conclude the paper in Section 6.

2. Background

In this section, we introduce a prospective memory architecture based on a combination of DRAM and STT-MRAM. We also discuss the requirements for applying hybrid memory architecture to IaaS data centers, where increased memory capacity would be highly beneficial.

2.1 Hybrid Memory System

The advent of NVM technologies has motivated researchers to envision new memory architectures. Because NVM technologies have not evolved to the point of being able to replace all the DRAM on a computer system, DRAM and NVM must be complementarily used to realize NVM-based main memory.

Most prior studies on hybrid memory systems discussed the combination of DRAM and PCRAM. In order to address its write endurance problems, these studies extended a memory controller to safely manage memory pages[5, 18, 19, 27]. In contrast, considering STT-MRAM does not have endurance problems, another design choice for a memory subsystem is possible. Indeed, we believe that it is possible to use the memory controller of a computer also for a hybrid memory system without any modification on it.

Today it is typical that DRAM is attached to a computer through a standard DIMM interface. Some memory vendors started shipping DIMM-based NVM products, e.g., NVDIMM (NAND flash backed DRAM) by Viking Technology. In these products, NVM is mapped to a particular address range of the main memory. From the viewpoint of the CPU, NVM and DRAM are both byte-addressable with different mapped address ranges. A software-based mechanism controls which data is stored in NVM or DRAM.

We assume that STT-MRAM will be attached to a computer in the same manner. Because its read/write performance will be comparable to that of DRAM, it is straightforward and reasonable to use the same interface as DRAM, and would be the most cost-effective design choice. It is not necessary to extend the current memory controller to support hybrid memory. Instead, a software mechanism would optimize the use of both types of memory modules for maximum energy efficiency of IaaS data centers.

2.2 IaaS Data Centers

The main targets of our project are future server platforms used in IaaS (Infrastructure-as-a-Service) data centers. Data centers using virtualization technologies would obtain great benefit from increased main memory capacities enabled by future NVM technologies. Larger memory capacities of physical machines (PMs) will allow IaaS service providers to consolidate more VMs onto a single PM, increasing the competitiveness of their commercial services.

Extending the current memory controller for page placement optimization is very costly and will not be feasible.

As discussed in the previous subsection, if STT-MRAM is considered, such a mechanism must be done at the software layer, not at the hardware layer. In academia, some prior work studied software mechanisms for NVM technologies requiring modifications on applications or operating systems[3, 4, 6, 9, 25]. For IaaS providers, it is difficult to force their customers to install such special mechanisms in their guest operating systems. Page placement must be done without impacting the customers' administrative domains.

We propose that the hypervisor layer, located between guest-OS software and physical hardware, is most suitable for implementing page placement optimization. However, existing hypervisors designed for DRAM-only memory architecture, do not have any support for hybrid memory systems; there is no mechanism to allocate guest pages from DRAM and NVM pages and dynamically optimize page mappings. This motivates us to explore the design of hybrid memory support at the hypervisor layer, which must provide transparency to guest operating systems, reduce the energy consumption of memory as much as possible, and minimize performance overhead of page migration.

2.3 Assumed Memory Architecture

To avoid misunderstanding in latter sections, we summarize here the memory architecture assumptions in this paper. As discussed above, if considering the performance of STT-MRAM predicted in 2026, these assumptions are valid.

- Both DRAM modules and STT-MRAM modules are connected to the byte-addressable memory bus of a computer (for example via DIMM interfaces).
- The memory pages of DRAM modules are mapped to a physical address range, and those of STT-MRAM modules are mapped to another address range. There are no channel and rank interleaving mechanisms between DRAM and STT-MRAM modules.
- BIOS provides system software with memory mapping information (e.g., the E820 memory map of x86 BIOS, and the UFEI GetMemoryMap interface). Through that interface, a system software program can identify which physical address range is mapped to which memory type.
- CPU cache (L1, L2, and LLC) works for both memory types. The CPU cache mechanism does not differentiate between them.

For I/O performance, we assume that the read/write latencies of DRAM are the same order of magnitude as STT-MRAM, unless otherwise noted. The energy consumption characteristics of DRAM are different from STT-MRAM in that the idle energy of STT-MRAM is nearly zero, and its write energy is some orders of magnitudes larger than that of DRAM. There are no endurance issues on STT-MRAM. Detailed parameter settings are given in the Evaluation section.

3. Hypervisor-based Page Replacement

We propose a hypervisor-based mechanism (*RAMinate*) transparently minimizing the energy consumption of a DRAM/STT-MRAM hybrid memory system. When creating a VM, our hypervisor allocates the RAM of the VM from both DRAM and STT-MRAM. While the VM is running, the hypervisor periodically monitors memory access of the VM and dynamically determines optimal guest page mappings. Then, it dynamically updates these mappings transparently to the guest operating system of the VM. In this section, we outline this mechanism step by step.

3.1 Memory Allocation

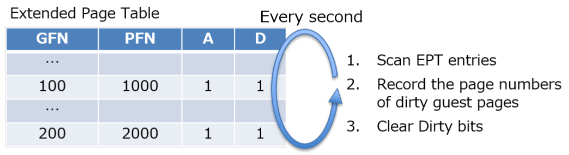
Upon the boot of a PM, the hypervisor obtains the memory mapping information of that machine through the BIOS interface. Then, it reserves a small amount of DRAM pages for the runtime memory of the hypervisor itself, and registers the rest of DRAM pages to the DRAM memory pool used for allocating RAMs of VMs. It also registers MRAM pages to the MRAM memory pool. These memory pools are used to manage free and in-use memory pages. From the memory pools, the hypervisor obtains defined amounts of DRAM and MRAM pages for creating the RAM of a VM. We bear in mind that service providers will predefine VM instance types for their customers. For example, a VM instance type will have 4GB virtual RAM comprising 1GB DRAM and 3GB MRAM; the guest operating system sees flat 4GB memory space, while in reality those memory pages are mapped to DRAM or MRAM. We assume that the DRAM/MRAM ratio predefined for an instance type does not dynamically change during the execution of a VM. Although DRAM and MRAM will have quite similar read/write performance, they will not be the exactly same. If we do inter-VMs page replacement, a VM may have a different DRAM/MRAM ratio than another VM of the same instance type, resulting in performance deviations among the same instance type of VMs. This will not be fair for customers. In order to assure the same performance for the same instance of VMs, we implement intra-VM page placement, which is performed between the DRAM and MRAM pages composing a single VM.

We extended Qemu/KVM by adding the support of hybrid memory systems. The extension allows specifying multiple physical memory address ranges to be used for composing a single view of VM RAM. The original Qemu has an abstraction class (Memory Backend) defining a common API for the management of memory regions of a VM. This framework is used to implement, for example, memory hotplug. It manages the RAM of a VM as a group of memory sub-regions, and it can dynamically add/remove a memory sub-region upon a hot-plug event.

We exploit this framework by adding new subclasses of Memory Backend, i.e., Memory Backend DevMem and Memory Backend Container. The former subclass allocates a RAM object using a given address range of physical memory. The latter subclass combines multiple RAM objects into a single pseudo RAM object. For example, when making a VM using both DRAM and STT-MRAM, the hypervisor creates two RAM objects of Memory Backend DevMem,

```
1   qemu-system-x86_64 -enable-kvm -hda image.qcow2 -m size=8G \
2   -object memory-backend-container,id=dmcon0,size=8G \
3   -object memory-backend-devmem,id=dram0,size=1G,phy-offset=0x1100000000,offset=0,parent=dmcon0 \
4   -object memory-backend-devmem,id=mram0,size=7G,phy-offset=0x3080000000,offset=0x40000000,parent=dmcon0 \
5   -numa node,nodeid=0,memdev=dmcon0
```

Figure 1: A command line example of Qemu/KVM with our hybrid memory support



GFN	PFN	A	D
…			
100	1000	1	1
…			
200	2000	1	1

Figure 2: The overview of memory access trace

i.e., a RAM object based on a free DRAM range, and a RAM object based on a free STT-MRAM range. Then, the hypervisor also creates a pseudo RAM object of Memory Backend Container that overlays the two RAM objects. The pseudo RAM object can be seen as a normal, single RAM object from other components in Qemu/KVM.

Figure 1 illustrates how free DRAM and STT-MRAM pages are combined for the RAM of a VM at a command line of Qemu/KVM. In this case, the VM virtually has 8GB RAM, which is mapped to 1GB DRAM and 7GB STT-MRAM. At the 2nd line, a pseudo RAM object of Memory Backend Container, `dmcon0`, is created. At the 3rd line, a RAM object of Memory Backend DevMem, `dram0`, is created and overlaid by `dmcon0`. It is mapped to a free DRAM space, the 1GB address range of physical memory starting from 0x1100000000. In the same manner, at the 4th line, a RAM object, `mram0`, is created from a free MRAM space. As specified in the `offset` option, `mram0` is placed after `dram0` in the Container object. At the 5th line, the created pseudo RAM object is assigned to NUMA domain 0 of the VM.

3.2 Memory Access Trace

The page placement mechanism needs to detect write-intensive memory pages and place them on DRAM, which consumes much less energy for write operation than MRAM. Some prior studies at the hardware level performed optimization at a cache-line granularity (e.g., typically 64 bytes in x86 platforms). Due to the lack of usable hardware interfaces, however, it is not possible to implement similar mechanisms for software-based optimization. For a memory access trace at the software level, a commonly-used method is binary instrumentation. It traps CPU instructions and records each memory operation. This method, however, incurs huge performance overheads and is not applicable to real-world IaaS data centers. Instead, we consider that a 4KB page will be the most feasible granularity to track memory access without serious overheads.

We developed a light-weight memory access trace mechanism exploiting the recent hardware support for virtualization (Figure 2). Recent Intel CPUs have a shadow page table mechanism (i.e., Extended Page Table, EPT) that helps hypervisors to execute guest operating systems. An EPT entry defines which guest physical page is mapped to which host physical page. A hypervisor maintains the EPT entries of a VM. Physical CPUs, executing its guest context, continuously refer the entries, and translate guest memory addresses to host physical ones. Notably, the processors of the Haswell generation or later possesses the Access and Dirty (A/D) bits of each EPT entry[8]. Once a processor has used an EPT entry (i.e., has accessed a guest physical page), it sets the Access bit. In the same manner, once the processor has used an EPT entry for write operation, it also sets the Dirty bit.

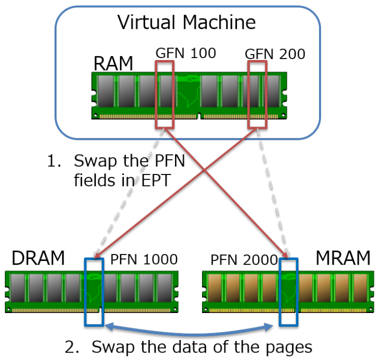


Figure 3: The overview of page remapping

The proposed mechanism scans EPT entries every second, records guest page numbers with the active Dirty bit, and clears the Dirty bit. The operation of setting the A/D bits is performed at the hardware level. As far as we observed, there are no noticeable performance overheads due to this operation. We carefully designed this trace mechanism to minimize its performance impact on running VMs. The mechanism creates a dedicated thread for the scan-record-clear steps of each VM. We can assign the scan-record-clear thread to another CPU than the CPUs executing the VM so as to avoid CPU resource contention. To manipulate EPT entries, the Compare-and-Swap (CAS) instruction is used. This allows us to avoid having a lock between the scan-record-clear thread and the CPU threads of the VM.

The trace mechanism creates a shared memory space to communicate with the page optimization daemon. It reports to the daemon the page frame numbers of dirty guest physical pages every second.

3.3 Page Remapping

As mentioned above, a write operation to an MRAM page consumes much more energy than writing to a DRAM page. Using an optimization algorithm (detailed in the next subsection), our system swaps an MRAM page having many write operations with a DRAM page having less write operations.

To implement this, we have developed a page remapping mechanism that swaps the mappings of two guest physical pages transparently to a VM (Figure 3). It exchanges the EPT entries of the guest physical pages as well as the data of these memory pages. For example, consider exchanging two guest physical pages, guest frame number (GFN) 100 and GFN 200, which are mapped to physical frame number (PFN) 1000 and PFN 2000, respectively. First, the remapping mechanism temporarily stops the VM. Then, it rewrites the PFN field in the EPT entry of GFN 100 with PFN 2000; it also rewrites the PFN field of GFN 200 with PFN 1000. Next, it also copies the 4KB data of PFN 1000 to a temporary buffer, copies that of PFN 2000 to PFN 1000, and copies the temporary buffer to PFN 2000. Finally, it restarts the VM.

We modified Qemu/KVM to support page remapping. The KVM driver of Linux maintains a mapping table from GFNs to host virtual addresses[1]. We added a new mapping table from GFNs to real (i.e., swapped) host virtual addresses, which enables KVM to refer to the contents of guest physical pages even after swapping them. Qemu, the userland code of Qemu/KVM, also refers to the contents of guest memory pages. In our extension, the information in the mapping table from GFNs to real host virtual addresses is cached in Qemu, which reduces context switches between the userland and the kernel space. A new control command, SwapPages, is added to the monitor interface of Qemu, enabling external programs to control page remapping.

The VM must be stopped temporarily to ensure that processors do not see inconsistent mapping information. The duration of this stop (i.e., the downtime of the VM), however, potentially impacts on the performance of the VM. When temporarily stopping the VM, Qemu/KVM waits for the next VM EXIT event and synchronizes all the threads comprising the VM (e.g., vCPU threads and I/O threads). The cost of this operation is not zero, therefore, our remapping mechanism is designed to allow swapping multiple guest physical pages at once during one temporary pause of the VM.

Table 1 shows a period of time elapsed for different numbers of page swaps being performed at once. We measured elapsed time 100 times for each configuration, and calculated their mean and standard deviation. We consider that approximately 30 ms is an acceptable downtime time for guest operating systems. This is the same as the default setting of Qemu/KVM for the maximum downtime of live VM migration. Thus, according to these results, our replacement system uses 1000 as the default value for the maximum number of simultaneous page swaps.

[1] In Qemu/KVM, a guest physical page is located at a virtual address of the host operating system.

Table 1: Elapsed time for simultaneous page swaps

# of Page swaps	Elapsed time in ms (mean $\pm$ SD)
0	28.3 ± 10.1
100	27.9 ± 11.9
1000	29.8 ± 10.0
10000	44.3 ± 14.5

3.4 Optimization Algorithm

3.4.1 Design Considerations

The optimization algorithm that determines which pages to swap between MRAM and DRAM should increase the hit ratio of write operations to DRAM under a limitation on the number of page migrations that can be executed at once without noticeable performance impact. The algorithm should also account for the recency and frequency of page write operations by a VM. These two concepts are used in MQ (Multi Queue) data caching algorithms, which were originally developed for proxy servers of network file systems[29]. It determines the priority of data blocks to be cached by using both the recency and frequency of data access from client nodes. Prior studies on hardware-based page placement[19, 27] used MQ, which motivates us to exploit it for our optimization algorithm at the hypervisor level.

Recency, as previously defined[29], is the temporal proximity between the time when a write operation to a page is invoked and the time when the next write request to the same page is invoked. A guest physical page that recently accepts a write operation, is likely to have another write operation in the near future. If such a page exists on MRAM, it should be migrated to DRAM so as to avoid further write operations to MRAM. Conversely, a guest physical page on DRAM that has no recent write operation, should be migrated to MRAM. An example of recency-based algorithms is LRU (Least Recently Used), which selects the least-recently-used guest physical page on DRAM as a victim page to be swapped with another page on MRAM.

Frequency, as previously defined[29], is spatial locality of issued write operations. The guest operating system is likely to write a particular range of memory more frequently than other ranges. These *hot* guest physical pages should have higher priority to be placed on DRAM. An optimization algorithm using only recency information can possibly migrate a cold guest physical page to DRAM that has only one rare write operation to it. Because the number of possible page migrations is limited, it is necessary to avoid as many ineffective page migrations as possible.

These prior studies extended a memory controller to implement page placement and assumed relatively low latency of page migration. Their MQ algorithms do not allow precise control of the amount of page migrations. Our hypervisor-based mechanism needs to carefully adjust the intensity of page migrations, otherwise it will experience noticeable performance overhead, because it temporarily stops the VM during page relocation.

3.4.2 Corked Multi Queue

We developed an optimization algorithm for hypervisor-based page replacement, Corked MQ (CMQ), that enables us to control the timing and number of page migrations. CMQ maintains the priority order of guest physical pages by receiving the input of dirty GFNs every second and updating the priority order. In parallel to this input operation, CMQ outputs migration plans asynchronously as pairs of GFNs to be swapped at any given interval. It is possible to also specify the maximum number of page pairs swapped at once.

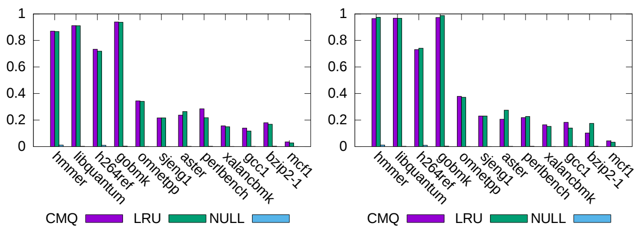
Similar to MQ, CMQ is composed of multiple queues. An element of the queues is an object representing a guest physical page; in short, we call it a guest page. A guest page object has 4 properties: 1) guest frame number, 2) physical memory type (i.e., DRAM or MRAM), 3) number of past detected updates, and 4) time of the last update. Generally, the queue level relates to the number of page updates from the past and within a queue, the tail of the queue has pages that experienced an update more recently.

At the initial state, all the guest pages on DRAM are located at the lowest level of queue, Q_0. When the update of a guest page is detected, CMQ moves it to the tail of a queue. Given the number of past detected updates of the page is n, CMQ moves it to the tail of the $\lfloor log(n) \rfloor$-th level of queue, $Q_{\lfloor log(n) \rfloor}$. This means that if the number of past detected updates of a page exceeds the threshold value, the page is *promoted* to a higher level of queue. When the update of a guest page on MRAM is detected, if the page already exists in a queue, CMQ moves it to the tail of $Q_{\lfloor log(n) \rfloor}$. If the page does not exist in any queue, CMQ appends it to the tail of Q_0.

CMQ refers to the first element of each queue (i.e., the oldest guest page in that queue) every second, and checks when the last update of the page happened. If the elapsed time from the last update exceeds a threshold value (i.e., life time), CMQ moves the page to the tail of the queue one level lower, i.e., the page is *demoted*. It repeats this check for the new first element of the queue, until its elapsed time from the last update is under the threshold value. In the case of demotion from the lowest queue Q_0, if the page exists on DRAM, CMQ moves it to the tail of Q_{victim}, which keeps candidates for page swaps. If the page exists on MRAM, CMQ removes it and clears the number of past detected updates to zero.

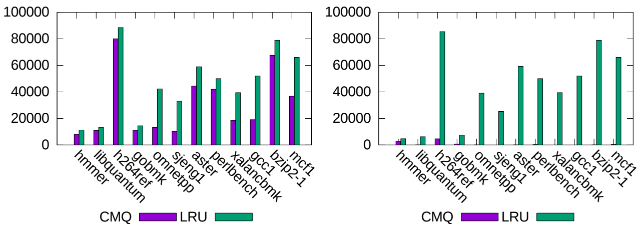
In the default setting, CMQ outputs the pairs of GFNs to be swapped every 5 seconds. CMQ scans elements from the highest queue to the lowest, from the tail of each queue to the head. If it finds a guest page on MRAM, it picks up the first element of Q_{victim} and outputs a pair of these GFNs. It repeats making the GFN pairs for page swaps, until the number of the pairs reaches the maximum number of page swaps at once, or until there are no more elements for page swap candidates.

The page remapping mechanism swaps guest physical pages according to the output of CMQ. If the swapping of a pair of GFNs succeeds, CMQ toggles the physical memory type of the guest pages. The guest page picked up from Q_{victim} is now on MRAM and removed from the queue.



(a) 1st iteration (b) 2nd iteration

Figure 4: The DRAM hit ratios with CMQ, LRU and the null algorithm. Note that the Y-axis values of the null algorithm are nearly zero in these graphs.



(a) 1st iteration (b) 2nd iteration

Figure 5: The numbers of page swaps with CMQ and LRU

3.4.3 Simulation

Before applying CMQ to dynamic page placement on a real-world system, we confirmed the advantages of CMQ by simulation. First, we launched a VM with 1 vCPU and 4GB RAM on a physical host, and recorded its dirty GFNs every second by using our memory access trace mechanism. To obtain memory access traces under various workloads, we ran the SPEC CPU 2006 benchmark on the guest operating system. A memory access trace file of each SPEC CPU workload is the sequence of wall clocks and dirty GFNs. Next, we developed a system to simulate dynamic page placement and implemented CMQ, a LRU-based algorithm, and a null algorithm (i.e., one that does not perform any replacement) for comparison. The simulation system reads a memory access trace file, and virtually optimizes page locations in response to dirty GFNs every second. Finally, it outputs simulation results including the DRAM hit ratio of write operations and the number of performed page migrations. It should be noted that this DRAM hit ratio is estimated on a per-page basis, not a write operation basis; therefore, when a dirty page on DRAM is detected the number of DRAM hits is incremented.

We repeated a simulation of each trace file two times sequentially. The first iteration is intended to simulate a situation where a memory access pattern has drastically changed. At the beginning of the first iteration, all DRAM pages are mapped to the lower address range of the guest memory. The placement system will aggressively migrate guest pages from their initial locations. After that, without clearing guest page locations, we start the second iteration with the same trace file. The second iteration is intended to simulate a quasi-steady state where a particular type of workload is continuously running.

Considering the memory usage of each SPEC CPU workload, we mapped 1% of the 4GB guest RAM (i.e., 40MB) to DRAM and the rest to MRAM for simulations. This enables us to observe that working sets of memory overflows DRAM in some workloads. The time interval of replacement is set to 5 seconds. The life time of a guest page is also set to 5 seconds. The number of levels in CMQ is set to 8. These are adequate default values determined through experiments. Intuitively, a guest page on the top queue may be evicted from DRAM if it does not have a write access within 40 seconds.

Figure 4 shows the DRAM hit ratios during the first and second iteration, respectively. Obviously, the dynamic replacement mechanism is necessary as the DRAM hit ratios with the null algorithm were nearly zero in any workloads. Although there were minor differences, the DRAM hit ratios of CMQ and LRU were comparable. In `perlbench` at the 1st iteration, the hit ratio of CMQ was slightly higher than that of LRU. On the other hand, in `aster` at the 2nd iteration, that of CMQ was slightly smaller. The mean value of these differences was only 1.75%. When comparing the 1st and 2nd iterations, we observed that all the DRAM hit ratios in the 2nd iteration were higher. In `hmmer`, `libquantum` and `gobmk`, the DRAM hit ratios reached more than 95%; since most of their working sets of memory were placed on DRAM in the 1st iteration and only few memory writes were issued to MRAM.

Figure 5 shows the numbers of page swaps during the first and second iteration, respectively. In comparison to LRU, CMQ drastically reduced the number of guest page swaps during each benchmark. In the 1st iteration, the mean number of guest page swaps with CMQ was 66% of the LRU case. In the 2nd iteration, CMQ had a mean number of guest page swaps 1.7%. Although CMQ and LRU achieved the same level of DRAM hit ratios, CMQ outperformed LRU in page swap efficiency. CMQ, taking into account the past page access, successfully avoided useless page swaps that did not contribute to the increase of the DRAM hit ratio. Given these results as a whole, we used CMQ in the following experiments using real hardware.

4. Evaluation

4.1 Methodology

The first step of evaluation is to confirm that our replacement mechanism successfully reduces write operations to MRAM. Next, we integrate those results with energy models to provide insight on the amount of energy reduction. Past studies typically used memory access trace and simulation. Using a binary instrumentation technique or a cycle-accurate CPU simulator, memory access trace of workloads was obtained. Then, this trace data was passed to a simulation system[17, 21] implementing a new idea of memory management. The problem with this approach is execution overhead, i.e., extremely severe slowdown of a running workload. For example, although depending on the environment, we experienced that a widely-used cycle accurate CPU simulator, GEM5[2], executed a workload 10^4 times slower than a real machine. For system software studies, a new memory management mechanism that is implemented at the operating system or hypervisor layer would require a full-system (not userland-only) simulation for evaluation. This is exacerbated with the long computation times these full-system simulations require making them unfeasible for most projects. This is a lack of an easy-to-use, reliable method to evaluate the performance of hybrid memory systems.

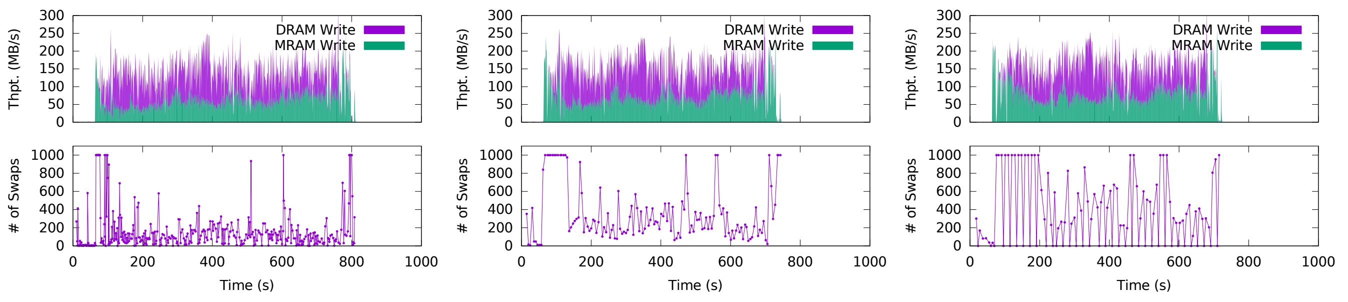
Table 2: The DRAM address mapping of the PM used for experiments

Channel	Start Offset	Size	Usage
0	0x0	2GB	Host OS
0	0xd00000000	14GB	Host OS
1	0x100000000	16GB	Host OS
2	0x500000000	16GB	DRAM Pool
3	0x900000000	16GB	MRAM Pool

Therefore, we developed an evaluation method for hybrid memory studies based on real hardware with minimal effort. The key idea is to emulate a hybrid memory system by using multiple memory channels and measure the number of read/write operations in a per-channel basis. By disabling the channel interleaving of the memory controller via the BIOS interface, the memory pages of each memory channel can be mapped to an individual range of physical address space. For example, as shown in Table 2, the physical address range of Channel 2 starts from 0x500000000. For experiments, we use this address range for the DRAM pool of the proposed hypervisor. In the same manner, we use Channel 3 for the MRAM pool. We also exploit the hardware mechanism of recent Intel Xeon CPUs to obtain the amount of read/write operations for the DRAM and MRAM pools. The recent Intel Xeon CPUs have a performance monitoring unit for each memory controller[7]. It can count the number of the DRAM CAS (Column Address Select) READ/WRITE operations performed at a memory channel. By multiplying them by the maximum burst size of DDR3 and DDR4 (i.e., 64 bytes, the same as the cache line size), we obtain memory read/write throughput in bytes. We input these results into the energy models of future memory devices to approximate the energy consumption of hybrid memory systems.

4.2 Experimental Setup

We used a PM equipped with an Intel Xeon Processor E5-2618L v3 (Haswell, 8 CPU cores and 20MB Last Level Cache) and DDR4 64GB DRAM (1866MHz). As mentioned in Section 4.1, the memory channel interleaving was disabled. To avoid performance fluctuation during experiments, the Dynamic Voltage and Frequency (DVFS) control of the processor was also disabled at the BIOS level. The host operating system was Debian GNU/Linux 8.3 (Jessie) with Linux Kernel 3.18.5 extended by our hypervisor-based page replacement mechanism. As shown in Table 2, 32GB of physical memory on Channel 0 and 1 was assigned for the host operating system, and the remaining memory was used for the DRAM and MRAM pools.



(a) Interval 1s (b) Interval 5s (c) Interval 9s

Figure 6: DRAM/MRAM write throughput (above) and the number of page swaps (below) under different page swap intervals. The workload is kernel compile. The DRAM size is 1% of VM memory (40MB).

We launched a VM with 1 vCPU and 4GB RAM on the host operating system. The guest operating system was standard Debian GNU/Linux without any special modifications. During and immediately after the boot of the guest operating system, the hypervisor performs page replacement aggressively. In each experiment, after we confirmed that page replacement became idle, we started a benchmark program on the guest operating system. It should be noted that although we present results only from Debian GNU/Linux, the guest operating system agnostic nature of our page replacement mechanism was confirmed by verifying that our mechanism works correctly for a VM running standard Microsoft Windows XP.

In order to evaluate our system under various conditions, we used 5 application programs having different memory access patterns. **Kernel compile** is a chain of tasks to build a Linux kernel. It first extracts its source code tree from a compressed archive, generates a configuration file, compiles the kernel and device drivers, and finally creates a compressed package file. We observed fluctuating, but continuous write traffic around 200MB/s. **MCF** is one of the SPEC CPU 2006 benchmark programs, which requires the largest memory (i.e., 1.7GB) among the benchmark programs. It is a memory-intensive program solving single-depot vehicle scheduling problems. We observed sporadic bursty write traffic up to 1-2GB/s. **Graph 500** is a data-intensive benchmark program that searches a large-scale, undirected graph tree. We adjusted the size of the graph tree to approximately 1.5GB. It first generates the graph tree and then performs graph searches. In addition to large memory read traffic, we observed that the graph search phase also causes periodic write traffic up to 50MB/s. **Apache** is a widely-used web server program. We configured it to serve 1000 files of 1MB each (i.e., 1GB in total). We used another PM to send HTTP requests to it. Both PMs are connected with Gb Ethernet. A benchmark program (`siege`)[1] randomly accessed served files. 4 client threads were launched; each client thread sent 8000 requests (i.e., 32000 requests in total). For approximately 30 seconds from beginning, we observed bursty memory write traffic (up to 150MB/s), during which most web contents were loaded to the page cache of the guest operating system. After that, memory write traffic was approximately at 50MB/s, which would be caused by HTTP request handling. **Memcached** is a key-value-store (KVS) server program caching data objects in memory, widely used for many IT systems to improve throughput of their services. We set the size of its memory pool to 2GB. On another PM, a benchmark program (`memtier`)[20] was configured to use the Memcached server to cache 1000 data objects of 1MB each. It generated 40000 read/write requests to these objects, conforming Gaussian distribution. First, memory write traffic went up to approximately 80MB/s. After 20 seconds, it kept fluctuating around 50MB/s. Memcached uses its own memory allocator that splits a large object into small memory chunks and tracks the use of those small chunks. It updates the metadata of memory chunks even for read requests for cached objects, which resulted in continuous write traffic in steady state.

4.3 Overview of Page Replacement

To illustrate how RAMinate dynamically optimizes page locations, we plot time series graphs for the results of the kernel compile experiments with different page swap intervals. In Figure 6, the top graph represents the write throughput of DRAM and MRAM, respectively. The lower graph is the number of page swaps performed by the replacement mechanism. For these experiments, the DRAM size is set to 1% (i.e., 40MB) of the 4GB VM RAM.

Careful inspection of the results for a page swap interval of 5 seconds (Figure 6b) revealed that at the beginning of kernel compile (approximately 50 seconds), most memory write traffic went to MRAM, which was approximately 200MB/s. Up to 150 seconds, the hypervisor aggressively optimized guest pages, periodically relocating 1000 pairs of guest pages, the maximum number of page swaps allowed to be performed at once. As a result, approximately 50% of write traffic was diverted to DRAM starting at about 200 seconds. Although the assigned DRAM size is only 1% of VM memory, the hypervisor successfully determined write-intensive guest pages and placed them in DRAM. Because hot memory locations were ever changing during kernel compiling, the hypervisor continuously performed page replacement until the end of each experiment.

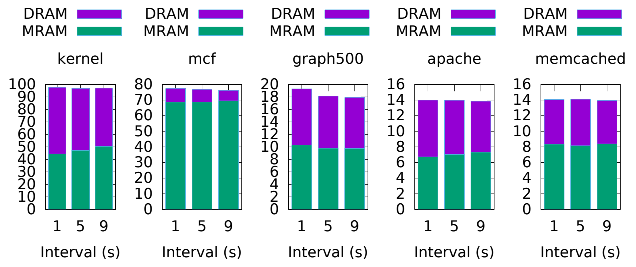


Figure 7: The total amount of written data (in GB) to DRAM and MRAM. The DRAM size is 1% of VM memory (40MB).

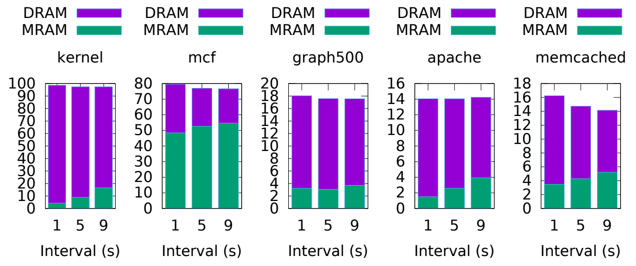


Figure 8: The total amount of written data (in GB) to DRAM and MRAM. The DRAM size is 10% of VM memory (400MB).

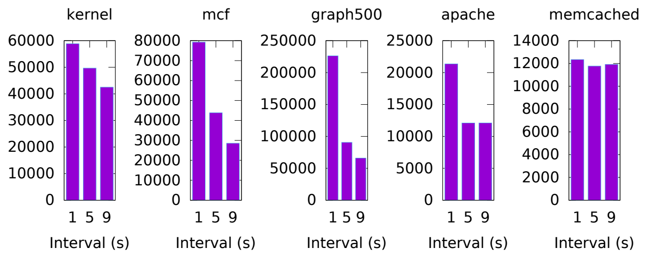


Figure 9: The number of page swaps during each experiment. The DRAM size is 1% of VM memory (40MB).

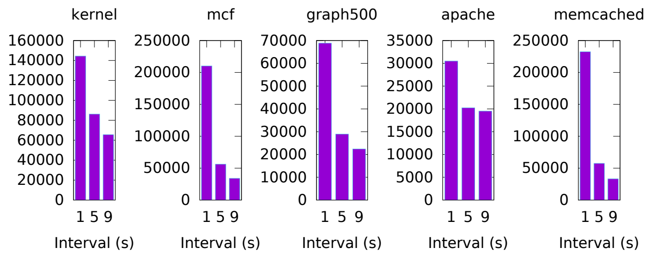


Figure 10: The number of page swaps during each experiment. The DRAM size is 10% of VM memory (400MB).

For a page swap interval of 1 second, as shown in Figure 6a, the hypervisor performed page replacement more frequently. The number of page swaps performed at once was smaller than that of the 5 second interval case; although, the number of page swaps performed during the experiment was larger ($\approx$ 59000 in the 1 second interval case versus $\approx$ 50000 in the 5 second interval case). The more frequent optimization reduced 6% more MRAM write traffic but degraded system performance, resulting in a 10% longer elapsed time for kernel compile.

4.4 System Performance Details

In addition to kernel compile, we have confirmed that RAMinate successfully worked for the other application programs. Figure 7 and Figure 8 show the total amount of written data to DRAM and MRAM respectively for different application programs. The former is the case that the DRAM size is 1% of VM memory and the latter is 10%. Even though the DRAM size is just 1%, the hypervisor successfully concentrated approximately half of memory write traffic at DRAM for kernel compile, Graph 500, Apache and Memcached. When the size of DRAM was increased to 10%, the hypervisor reduced the percentage of written data to MRAM for all tested application programs. For kernel compile, the percentage of memory write traffic to MRAM was below 20%. For MCF, however, it was still high (i.e., more than 60%) even with 400MB DRAM. The working set of memory in MCF is approximately 1.7GB, which far exceeds the DRAM sizes. Also, MCF rather uniformly writes this working set of memory. For such applications, the DRAM ratio in VM memory should be set to a higher value to efficiently cover hot memory pages.

For all tested applications, in a shorter page swap interval, the amount of write traffic to MRAM was smaller; however, the total amount of write traffic to both DRAM and MRAM slightly increased, due to the memory copy cost of page swap operations. Figures 9 and 10 show the total number of page swap performed during each experiment. When the DRAM size was 1% and the page swap interval was 1 second, the total number of page swaps in Graph 500 was approximately 230000, which was 140000 higher than the case of the 5 seconds swap interval. This resulted in the approximately 1GB higher total amount of written data. Because one page swap operation involves 2 page writes to the DRAM and MRAM pools[2], the increased size of written data can be roughly estimated as 4 (KB/page) * 2 (pages) * 140000 = 1 (GB).

A shorter page swap interval can also impact performance as shown in Figures 11 and 12, which depict normalized application performance in each experiment. For each benchmark program, we used the inverse of the time required for completing it as an indicator of application performance. We also obtained the performance of each application program without performing any page relocation, and used it as the reference performance value of each application for normalization. RAMinate incurred performance degradation. When the page swap interval was 5 seconds, the worst performance degradation was by approximately 17% in MCF with 40MB DRAM, and the least degradation was by approximately 7% in Graph 500 with 400MB DRAM. Since the replacement mechanism needs to temporarily stop the VM while remapping guest pages, a certain amount of performance degradation is inevitable. This suggests that there is a trade-off between power saving and performance degradation. RAMinate has the advantage of reducing power consumption of the memory subsystem. It should be noted that during the

[2] As explained in Section 3.3, a page swap operation is performed via the temporal buffer. The temporal buffer is allocated from the memory of the host operating system, not from the DRAM pool used for VM memory.

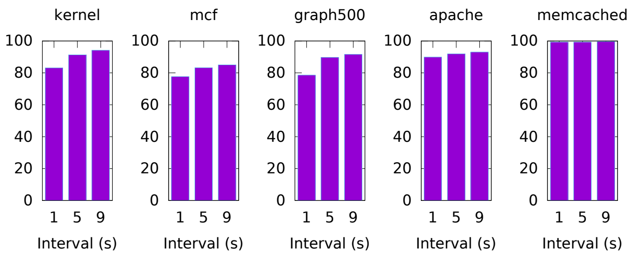


Figure 11: Normalized benchmark performance (in %). The DRAM size is 1% of VM memory (40MB).

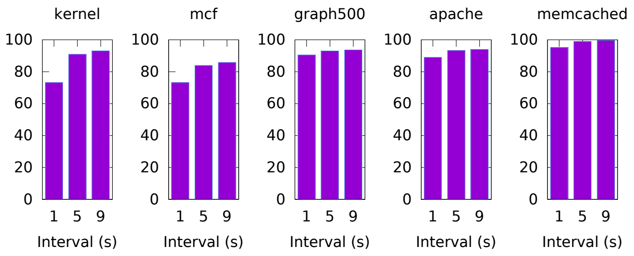


Figure 12: Normalized benchmark performance (in %). The DRAM size is 10% of VM memory (400MB).

Memcached the network bandwidth between the client and server was the performance bottleneck. In those graphs, performance degradation due to shorter page swap intervals is not as evident.

We noticed that the actual performance degradation was larger than expected from the preliminary experiments in Section 3.3. A possible reason is the overhead of flushing EPT entry cache. According to the technical document of Intel CPUs, when changing existing EPT entries, the cache of EPT entries in physical processors must be flushed by invoking a special CPU instruction (i.e., INVEPT). Upon completion of guest pages remapping, our hypervisor invokes INVEPT before restarting the guest operating system. In order to execute in the context of the guest operating system, the processors need to determine the mapping of guest physical and host physical pages. If the workload running on the guest operating system accesses a wide range of memory pages, re-caching that information may incur noticeable performance degradation. This problem can be addressed by slightly extending the existing CPU instruction. In the same manner as the instruction to invalidate TLB entries (i.e., INVLPG), INVEPT can be extended to allow software programs to specify which EPT entries to be invalidated in the CPU cache.

As noted in Section 2.3, we assumed that the read/write latencies of STT-MRAM are the same order of magnitude as those of DRAM according to the technology roadmap. Although these latencies will not be exactly the same in reality, one study[14] shows that a small increase of latency (e.g., 5%) will not impact application performance in large-scale scientific computing. Moreover, CPU cache also reduces the performance impact of the increased latency, by benefiting from the locality of memory access. Thus, in many cases, small differences in memory access latency among guest physical pages will not adversely impact the performance of guest operating systems running on our hypervisor-based hybrid memory system.

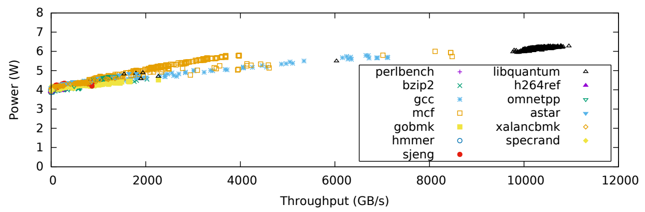


Figure 13: The relationship between memory I/O throughput and power consumption during the SPEC CPU benchmarks. Values were obtained via the performance monitoring unit of the memory controller of the used PM.

4.5 Energy Estimation

Based on the experimental results above, we have estimated the energy consumption of our hypervisor-based hybrid memory system. The intent is to explore how much energy can be saved (or wasted) under various workloads if our system is applied to future STT-MRAM devices. We introduce a simple model used for energy estimation and then explain our simulation results.

4.5.1 Energy Model

First, we measured actual memory I/O throughput and DRAM power consumption during the SPEC CPU benchmark. Using the same PM for the above experiments, we ran the benchmark program and periodically recorded memory I/O throughput and DRAM power consumption by reading performance monitoring units of the memory controller every second. As shown in Figure 13, there is an approximately linear correlation between memory I/O throughput and DRAM power consumption; 1W for every 4GB/s increase in throughput. When I/O throughput is nearly zero, power consumption is non-zero ($\approx$ 4W) due to the idle power of DRAM modules. For energy estimation, we assume this simple, linear energy model, which is sufficient to explore the impact of write energy consumption of future STT-MRAM (being orders of magnitude larger than DRAM) on the entire energy consumption of a hybrid memory system.

Another reason to use a simple linear model is that the energy saving mechanisms of DRAM will have less chance to work in busy server systems. Prior studies[16, 24] discussed the details of DRAM power consumption by accounting for the internal structure of DRAM modules (e.g., rank and bank) and energy saving mechanisms (e.g., CKE On/Off and self-refresh mode). As for the DRAM energy model itself, we did consider that it would be possible to improve its accuracy by accounting for these details. Actually, we also measured the CKE On/Off ratios during the benchmark, and noticed that the deviation from the linear correlation is partially explained by the difference of the CKE On/Off ratios observed during each workload. Although, currently, we intend to discuss the feasibility of our hybrid memory system for future IaaS data centers. A PM will consolidate multi-

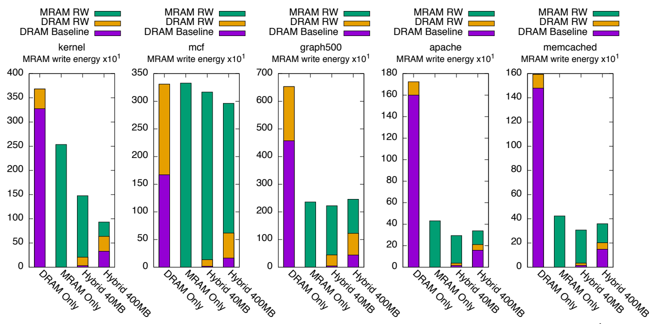


Figure 14: Estimated energy consumption (MRAM write energy x10^1)

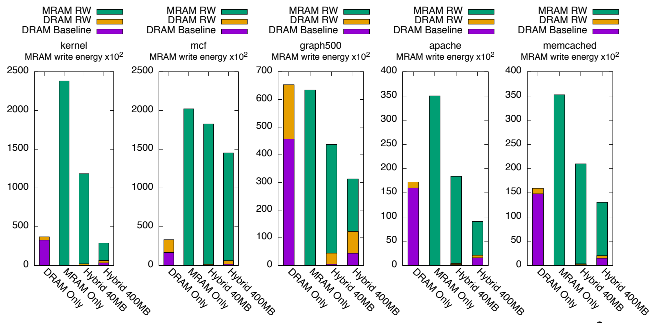


Figure 15: Estimated energy consumption (MRAM write energy x10^2)

ple busy VMs, where its memory controller will have less chance to enable these energy saving mechanisms.

For energy estimation, we assume that the read/write energy of DRAM and the read energy of STT-MRAM are the same, E_0 (J/byte). In addition to the read/write energy, DRAM always consumes baseline power, P_i (W), but STT-MRAM does not.[3] The value of P_i is adjusted to be proportional to the physical DRAM size assigned to a VM. Given the total amount of read and written data to DRAM as d_r (bytes) and d_w (bytes), the DRAM energy consumed by the VM during an experiment is estimated as $E_0 * d_r + E_0 * d_w + P_i * t$ (J), where t is the experiment duration. Given the write energy of STT-MRAM is E_1 (J/byte) and the total amount of read and written data to STT-MRAM as m_r (bytes) and m_w (bytes), the STT-MRAM energy consumed during an experiment is estimated as $E_0 * m_r + E_1 * m_w$ (J). We consider 3 future possible write energies of STT-MRAM, which are 10^1, 10^2 or 10^3 times larger than that of DRAM. A write energy 10^2 greater is in line with the perspective noted in Section 1; i.e., $E_1 = 10^2 * E_0$. For the parameters of the energy model, we used the values obtained on the PM.

4.5.2 Simulation Results

Figures 14-16 show estimated energy (J) consumed for the 4GB VM RAM under different orders of STT-MRAM write energy. Each graph compares 4 configurations of the VM RAM: 1) composed of solely DRAM, 2) composed of solely STT-MRAM, 3) hybrid memory with 1% DRAM (i.e., 40MB DRAM and 4056MB MRAM, denoted as ”Hybrid 40MB”), or 4) hybrid memory with 10% DRAM (i.e., 400MB DRAM and 3696MB MRAM, denoted as ”Hybrid 400MB”). Although the Y axis scale is different among these Figures, the DRAM-only system consumes the same amounts of energy. As shown in Figure 15, if we assume 10^2 time larger MRAM write energy than DRAM read/write energy, the hybrid memory system with 10% DRAM will consume smaller amounts of energy than the DRAM-only system for most workloads. For Graph 500, its energy consumption is just 48% of the DRAM-only case. For Apache, it is just 53%. However, for MCF, the hybrid memory system has 4.4 times larger energy consumption than the DRAM-only system. Since MCF writes approximately 1.7GB memory space intensively, the hybrid memory system even with 10% DRAM cannot effectively concentrate write accesses to the DRAM area. If we can increase the DRAM ratio, for example, to 50%, the hybrid memory system will outperform the DRAM-only system also for MCF.

As shown in Figure 14, if we assume more energy-efficient STT-MRAM, the hybrid memory system even with 1% DRAM outperforms the DRAM-only system for any tested workloads. For example, the hybrid memory system with 1% DRAM reduces energy to 40% for kernel compile, and the hybrid memory system with 10% DRAM reduces energy to 25%. The MRAM-only system also achieves less amounts of energy for the workloads except MCF. Figure 16 shows simulation results with 10^3 times larger MRAM write energy than that of DRAM read/write. Clearly, it is not feasible to use MRAM for the main memory of the VM for any tested workloads. Even in the best case (i.e., Graph 500 with 10% DRAM), the hybrid memory system consumes 1.5 times larger energy than the DRAM-only system.

Through simulations, we are confident that the hypervisor-based hybrid memory system using DRAM and STT-MRAM will provide an energy-efficient memory subsystem for PMs in future IaaS data centers. It will be possible to overcome the disadvantage of STT-MRAM (i.e., large write energy) by concentrating write operations to the DRAM part of the hybrid memory. Although the amount of consumed energy depends on the behavior of a workload running on the guest operating system, if a future STT-MRAM technology can suppress write energy below 10^2 times of DRAM read/write energy, there will be many

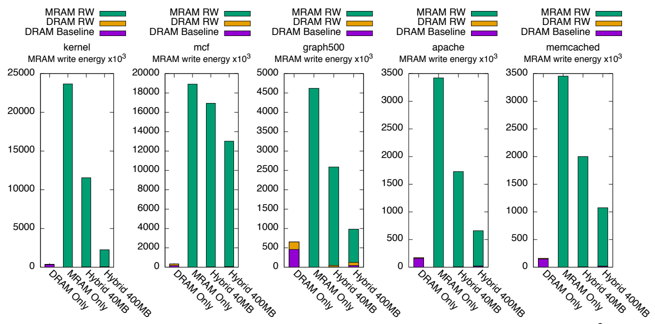


Figure 16: Estimated energy consumption (MRAM write energy x10^3)

[3] Although the peripheral circuit driving STT-MRAM cells may still consume some amount of energy, researchers are studying mechanisms to reduce and eliminate it, for example, by power gating. In this paper, we assume that the leak energy of the peripheral circuit is negligible.

situations where the hybrid memory system is beneficial for energy saving. In some cases, the optimal DRAM ratio of the hybrid memory system will dynamically change while a VM is running. Although there is a concern about fairness among customers (discussed in Section 3.1), it would be possible to extend the page replacement mechanism so as to adjust the DRAM ratio on the fly without shutting down the guest operating system.

5. Related Work

In the computer architecture area, hybrid memory systems using DRAM and PCRAM have been investigated by means of hardware simulation. In [18], DRAM is placed between a memory controller and PCRAM, which buffers written data and reduces write operations to PCRAM. In [19], a memory controller implements MQ and performs page migrations. When the memory controller performs page migrations, the information of new page mapping is reported to an operating system. Then, the operating system updates corresponding TLB entries. A study [27] asserts that 3D-stacked PCRAM is energy efficient because the higher temperature of 3D chips contributes to reducing the programming energy of PCRAM cells. A memory controller implements MQ. The paging mechanism of an operating system is extended to optimize page locations between PCRAM and DRAM. In [5], a memory controller counts the number of write operations to PCRAM, and if the number exceeds a threshold value, the controller sends a hardware interrupt to an operating system. The operating system implements a PCRAM-aware memory allocator and a page swap mechanism. While these studies implement page remapping at the hardware level with the aid of an operating system, our mechanism performs page remapping at the hypervisor level, which does not need design changes of existing memory controllers.

There are several studies researching the possibilities of building the main memory only with PCRAM[12, 28] or STT-MRAM[11] without using DRAM together. Because an electric charge in the capacitor of a DRAM cell is destroyed upon a read operation from the cell, DRAM chips are designed to write back all the bits of an entire row when closing the row. On the other hand, the state of an NVM cell is not altered by a read operation, and it is preferable to avoid as many cell write operations as possible because of energy (and PCRAM's endurance) concerns. These studies present techniques to avoid redundant bit-writes, such as data comparison and write tracking inside a cache line. Although they are targeting NVM-only main memory, these techniques will be also applicable for the NVM portion of a hybrid memory system. These hardware-level optimization techniques are complementary to our hypervisor-level technique. If they are implemented in a memory controller, RAMinate will enjoy further energy saving.

STT-MRAM will be likely used also for the replacement of SRAM on CPU. For example, one study [22] aims at reducing power consumption of CPU cache by partially replacing SRAM on CPU with STT-MRAM. The study claims that, by lowering retention time of STT-MRAM cells, comparable read/write latency is possible. At the early stage of production, it is expected that the size of an STT-MRAM module will be small and suitable for the use as CPU cache. As STT-MRAM technologies mature, the density of an STT-MRAM module will increase. Our study is based on such a rather long-term projection of future STT-MRAM technologies.

The advent of NVM technologies also motivated researchers to explore software-based support for persistent memory. A study [25] presents a programming interface for persistent memory, which provides userland libraries for transactional memory operations. It extends a C/C++ compiler and linker to strictly control CPU cache and instruction ordering. In [3], a C++ library safely allocates objects on persistent memory, which prevents users from writing inconsistent pointer operations. A pointer on persistent memory is not allowed to refer to an object on non-persistent memory. Such a reference will cause a dangling pointer upon a system crash. [9] abstracts the NVM portion of main memory as a NUMA node in the virtual memory system of an operating system. To seamlessly integrate NVM with existing memory management mechanisms, it exploits the page placement framework of NUMA also for NVM. [4] and [6] present the designs of file systems targeting fast, byte-addressable NVM technologies. The page cache mechanism is bypassed since NVM performance is comparable to DRAM. To guarantee consistency, they developed techniques to safely update objects on persistent memory, such as cacheline-based metadata logging and light-weight copy-on-write. Although RAMinate hides the existence of NVM from the guest operating system of a VM, if its dynamic page placement is disabled, RAMinate maps NVM to a fixed range of guest physical addresses. It is possible to use these persistent memory technologies on the guest operating system.

6. Conclusion

In this paper, we propose a hypervisor-based hybrid memory mechanism that reduces write traffic to STT-MRAM by optimizing page locations between DRAM and STT-MRAM. Our system successfully reduced write traffic to STT-MRAM by approximately 70% for tested workloads, which results in a 50% reduction in energy consumption in comparison to a DRAM-only system. If a future STT-MRAM technology can suppress write energy below 10^2 times of DRAM read/write energy (as predicted by a technology roadmap), there will be many situations where the hybrid memory system is beneficial for energy saving.

The authors would like to thank Dr. Jason Haga and other colleagues for their invaluable feedback. This work is partially supported by the IMPULSE project of AIST and JSPS Grant KAKENHI 16K00115.

References

[1] Siege. `http://www.joedog.org/siege-home`.

[2] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The Gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, Aug. 2011.

[3] J. Coburn, A. M. Caulfield, A. Akel, L. M. Grupp, R. K. Gupta, R. Jhala, and S. Swanson. NV-Heaps: Making persistent objects fast and safe with next-generation, non-volatile memories. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 105–118. ACM, 2011.

[4] J. Condit, E. B. Nightingale, C. Frost, E. Ipek, B. Lee, D. Burger, and D. Coetzee. Better I/O through byte-addressable, persistent memory. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, pages 133–146. ACM, 2009.

[5] G. Dhiman, R. Ayoub, and T. Rosing. PDRAM: A hybrid PRAM and DRAM main memory system. In *Proceedings of the 46th ACM/IEEE Design Automation Conference*, pages 664–669, Jul 2009.

[6] S. R. Dulloor, S. Kumar, A. Keshavamurthy, P. Lantz, D. Reddy, R. Sankaran, and J. Jackson. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 15:1–15:15. ACM, 2014.

[7] Intel Corporation. Intel Xeon processor E5-2600 product familiy uncore performance monitoring guide, Mar. 2012.

[8] Intel Corporation. Intel 64 and IA-32 architectures software developer's manual, Sept. 2014.

[9] S. Kannan, A. Gavrilovska, and K. Schwan. pVM: Persistent virtual memory for efficient capacity scaling and object storage. In *Proceedings of the Eleventh European Conference on Computer Systems*, pages 13:1–13:16. ACM, 2016.

[10] A. Kivity, Y. Kamay, D. Laor, and A. Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, pages 225–230. The Linux Symposium, 2007.

[11] E. Kltrsay, M. Kandemir, A. Sivasubramaniam, and O. Mutlu. Evaluating STT-RAM as an energy-efficient main memory alternative. In *Proceedings of the 2013 IEEE International Symposium on Performance Analysis of Systems and Software*, pages 256–267, apr 2013.

[12] B. C. Lee, E. Ipek, O. Mutlu, and D. Burger. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 2–13, New York, NY, USA, 2009. ACM.

[13] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T. W. Keller. Energy management for commercial servers. *IEEE Computer*, 36(12):39–48, Dec 2003.

[14] D. Li, J. S. Vetter, G. Marin, C. McCurdy, C. Cira, Z. Liu, and W. Yu. Identifying opportunities for byte-addressable non-volatile memory in extreme-scale scientific applications. In *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pages 945–956, May 2012.

[15] J. S. Meena, S. M. Sze, U. Chand, and T.-Y. Tseng. Overview of emerging nonvolatile memory technologies. *Nanoscale Research Letters*, 9(1):1–33, 2014.

[16] Micron Technology, Inc. Calculating memory system power for DDR3. Technical Report TN-41-01, Micron Technology, Inc, May 2007.

[17] M. Poremba, T. Zhang, and Y. Xie. NVMain 2.0: A user-friendly memory simulator to model (non-)volatile memory systems. *IEEE Computer Architecture Letters*, 14(2):140–143, July 2015.

[18] M. K. Qureshi, V. Srinivasan, and J. A. Rivers. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 24–33, New York, NY, USA, 2009. ACM.

[19] L. E. Ramos, E. Gorbatov, and R. 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.

[20] Redis Labs. memtier_benchmark. https://github.com/RedisLabs/memtier_benchmark.

[21] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. DRAMSim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters*, 10(1):16–19, Jan 2011.

[22] C. W. Smullen, V. Mohan, A. Nigam, S. Gurumurthi, and M. R. Stan. Relaxing non-volatility for fast and energy-efficient STT-RAM caches. In *Proceedings of the IEEE 17th International Symposium on High Performance Computer Architecture*, pages 50–61, Feb 2011.

[23] The International Technology Roadmap for Semiconductors (ITRS). International technology roadmap for semiconductors 2013 edition, 2013.

[24] T. Vogelsang. Understanding the energy consumption of dynamic random access memories. In *2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 363–374, Dec 2010.

[25] H. Volos, A. J. Tack, and M. M. Swift. Mnemosyne: Lightweight persistent memory. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 91–104. ACM, 2011.

[26] L. Wang, C.-H. Yang, and J. Wen. Physical principles and current status of emerging non-volatile solid state memories. *Electronic Materials Letters*, 11(4):505–543, 2015.

[27] W. Zhang and T. Li. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In *Parallel Architectures and Compilation Techniques, 2009. PACT '09. 18th International Conference on*, pages 101–112, Sept 2009.

[28] P. Zhou, B. Zhao, J. Yang, and Y. Zhang. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, pages 14–23, New York, NY, USA, 2009. ACM.

[29] Y. Zhou, J. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the*

General Track: 2001 USENIX Annual Technical Conference, pages 91–104. USENIX Association, 2001.