Leveraging Phase Change Memory to Achieve Efficient Virtual Machine Execution

Ruijin Zhou  
Intelligent Design of Efficient Architectures Laboratory  
University of Florida  
zhourj@ufl.edu

Tao Li  
Intelligent Design of Efficient Architectures Laboratory  
University of Florida  
taoli@ece.ufl.edu

Abstract
Virtualization technology is being widely adopted by servers and data centers in the cloud computing era to improve resource utilization and energy efficiency. Nevertheless, the heterogeneous memory demands from multiple virtual machines (VM) make it more challenging to design efficient memory systems. Even worse, mission critical VM management activities (e.g. checkpointing) could incur significant runtime overhead due to intensive IO operations. In this paper, we propose to leverage the adaptable and non-volatile features of the emerging phase change memory (PCM) to achieve efficient virtual machine execution. Towards this end, we exploit VM-aware PCM management mechanisms, which 1) smartly tune SLC/MLC page allocation within a single VM and across different VMs and 2) keep critical checkpointing pages in PCM to reduce I/O traffic. Experimental results show that our single VM design (IntraVM) improves performance by 10% and 20% compared to pure SLC- and MLC- based systems. Further incorporating VM-aware resource management schemes (IntraVM+InterVM) increases system performance by 15%. In addition, our design saves 46% of checkpoint/restore duration and reduces 50% of overall IO penalty to the system.

Categories and Subject Descriptors  D.4.8 [Operating Systems]: Performance; C.4 [Computer System Organization]: Performance of Systems

General Terms  Management, Performance, Design

Keywords  Virtualization, Phase Change Memory, Checkpointing, Memory Management

1. Introduction
Virtualization [1] improves server utilization by allowing a physical machine to host multiple virtual machines (VMs). Among all the shared hardware resources, memory largely affects the performance of VMs. It is more challenging to design efficient memory subsystems for the virtualized platforms since the VMs have various memory demands and the applications running within each VM further manifest heterogeneous working sets. Therefore, it is both desirable and crucial to have a memory system that is adaptable to different memory requirements of VMs and applications. Moreover, the design of intelligent memory systems could benefit critical VM management activities. For instance, checkpointing is widely used for system backup, fault tolerance and disaster recovery in virtualized servers and data centers. Nevertheless, it is regarded as one of the most expensive VM operations since significant IO operations are involved to transfer the entire VM memory image back to the disk. If the main memory is non-volatile, the checkpoint/restore duration and IO penalty on the system can be reduced by substituting unnecessary IO transactions with memory-to-memory operations.

Recent advances in emerging memory technologies provide new opportunities to build efficient and intelligent memory systems for virtual machine execution. In this work, we explore the opportunity of leveraging phase change memory (PCM) [2] to achieve this goal. PCM is one of the most promising technologies for future memory system design due to its superior scalability, non-volatility and capability of storing multiple bits per cell (i.e. Multi Level Cell or MLC). Recent work [2, 3, 5, 7] incorporates PCM management schemes at hardware memory controller level and the software is only invoked to ensure the correct execution. It is our belief that the virtual machine monitor (VMM), which has full knowledge of system runtime characteristics, should play a more active role in improving the efficiency of PCM systems.

Adhering to this philosophy, we explored holistic, VM-aware PCM-based memory system design that can intelligently manage memory resources via the cooperation between VMM and hardware. Our proposed design includes three layers of management, namely, hardware (physical page frame), memory page and VM. At the bottom layer, hardware interacts with physical memory and exposes the PCM device features to the VMM. The memory page management layer considers not only the tradeoff between latency and capacity within single VM but also the heterogeneous memory demands across multiple VMs. Our scheme dynamically identifies the optimal SLC (one bit per cell)/MLC page ratio for both single and multiple VMs. The top
layer, VM-level management, targets the optimization of performance critical VM activities (e.g. checkpoint/restore). Our scheme exploits lazy restore mechanism [5] and PCM non-volatility to retain all SLC pages of that checkpoint within the main memory and only writes back MLC pages of the checkpoint to disk. Doing so not only reduces the checkpoint/restore time but also the IO penalty to other normal running VMs. Experimental results show that our single VM design (IntraVM) enhances the system performance by 10% and 20% respectively compared to the SLC- and MLC- based design. This improvement is comparable to hardware-based solution. Further incorporating VM-aware optimizations (IntraVM+InterVM) increases performance by 15%. In terms of checkpoint/restore, our proposed design not only achieves 46% less checkpoint/restore time but also reduces 50% overall IO penalty to the system.

The rest of this paper is organized as follows: Section 2 provides background and motivation. Section 3 describes our design in detail. Sections 4 and 5 present our simulation framework and experimental results. Section 6 discusses related work and Section 7 concludes the paper.

2. Background and Motivation

![Figure 1. The Conversion between SLC/MLC Pages](image1)

![Figure 2. The Trade-off between Capacity and Latency on SLC, MLC and other material (Figure not to scale)](image2)

2.1. Phase Change Memory and Hardware-based Hybrid PCM Design

Phase change memory (PCM) is a promising technology to build future memory systems due to its high scalability and non-volatile nature [2, 6]. A PCM device that stores one-bit per cell is denoted as a single-level cell (SLC) while the one that can represent multiple bits per cell is called a multi-level cell (MLC). Note that in this paper, a MLC is assumed to contain 2 bits per cell. A PCM cell can be programmed to be either a SLC (1 bit per cell) or a MLC (2 bits per cell). The high density of MLC comes at the price of high latency. From the software perspective, a single 4KB page mapped to SLC devices can be converted to two 4KB (total 8KB) pages when the PCM devices are configured as MLC and vice versa, as illustrated in Figure 1. Therefore, the PCM device doubles its effective capacity when converted from SLC to MLC.

Prior studies [7, 8] proposed hardware-based hybrid PCM systems that can achieve the benefit of both SLC and MLC. Existing design largely focuses on the hardware level and the software is only used to ensure the correctness. As a result, the management of SLC and MLC devices is limited to the memory controller.

2.2. Xen and Virtual Machine Monitor

Xen is one of the most popular virtual machine monitors (VMM) that allow multiple virtual machines to run on top of a single server [1, 9]. Since there are several virtual machines competing for memory resources, Xen needs to balance the memory allocation across different VMs. After a VM boots up and runs, most memory requests are directly handled by the hardware without involving the VMM. However, machine to physical page tables, which contain page allocation information for all guest VMs, are considered as privileged data and can only be accessed by the VMM. In addition, the VMM validates critical operations such as installing a new page table or updating a page table entry [1, 9, 10]. This allows the VMM to have comprehensive knowledge of memory requests from all guest VMs.

VMs may run a wide range of workloads, leading to quite different demands on memory resource. Therefore, balancing memory utilization across VMs can improve the overall system performance. To achieve this, [11] proposed the balloon technology to resize VM memory capacity. As a kernel space driver, the balloon driver provides the OS an illusion that it requests more memory to run. In fact, the VMM reclaims the allocated memory to the balloon driver and the available memory for that VM is reduced. To balance the memory requirement across different VMs, the VMM pages out the memory of one VM’s balloon driver to provide free memory for another VM by shrinking its balloon driver.

Checkpoint/restore [14] is a crucial event in the virtualized environment to support system backup, fault tolerance and disaster recovery for servers and data centers [26]. Checkpointing a running VM involves creating a snapshot of its memory, IO device states, network connections, and the contents of its virtual CPU registers. After a checkpoint is created, the VMM writes it back to the permanent storage (i.e. hard disk). When a VM is restored from that checkpoint, the VMM performs disk IOs to reload it back to memory and CPU registers. Performance degradation is unavoidable due to bursty, high-volume disk IO operations. Therefore, how to reduce IO traffic becomes an important issue for checkpoint/restore.

2.3. Motivation

Conventional DRAM systems with fixed capacity cannot adapt with varying memory demands during the different execution phases of workloads, which could result in low performance due to increased page faults. A hardware-based hybrid PCM system with adaptable memory capacity [7] is proposed to address this issue. Based on the physical addresses issued to the memory controller, the ratio of SLC and MLC pages are determined. In order to guarantee the correctness, balloon technology is used to adjust the

180
available memory that is visible to the OS. Nevertheless, the hardware memory controller is not aware of system-wide runtime and resource utilization characteristics (e.g., memory demands across different VMs, domain switch, VM specific activities), which hinders the hardware approach from fully exploiting the benefits of hybrid PCM systems in virtualized execution environment. To achieve better coordination between hardware and software, we propose to incorporate virtual machine monitor, which has the ability to control both hardware and software behavior, into PCM management schemes.

Within a single VM, there is a tradeoff between memory access speed and capacity. As can be seen in Figure 2, although SLC pages have fast access speed, the effective capacity of SLC-based memory is smaller, which can potentially lead to more page faults, triggering expensive IO requests. On the other hand, the capacity of MLC-based memory is larger whereas accessing pages in MLC mode is slower. If the ratio between SLC and MLC pages for single VM can be tuned according to the memory demand, the performance can be improved by not only having the least number of page faults but also achieving the maximal benefit from the lower latency provided by the SLC devices.

Across different VMs, there is conflict between maximum memory supplies and variable memory demands. The maximum available memory capacity is prescribed when a VM is created. However, during execution, memory demands will vary, which makes the maximum memory supplies unbalanced. In an extreme case, the idle VMs may occupy a large fraction of faster memory pages (SLC) while other busy VMs are suffering from slower pages (MLC or HDD), which will degrade overall performance.

Furthermore, checkpoint/restore, as a frequent VM behavior, triggers significant amount of IO requests to transport all the memory pages to disk. These bursty IO requests not only determine the performance of checkpoint/restore but also affect the IO performance of normal running VMs. Due to the non-volatile feature of PCM, there is no need to swap the entire VM memory to disk. If the VM checkpoint pages can be appropriately distributed between MLC and HDD, the amount of disk data transfer will be significantly reduced, which will further result in faster checkpoint/restore time and lower IO performance degradation for normal running VMs.

Different VMs require different memory capacities to keep the same page miss ratio. As shown in Figure 3, VM2 which exhibits poor memory locality and needs more memory capacity, is categorized as capacity sensitive VM while VM1, which manifests good memory locality and requires less memory capacity, is attributed to latency sensitive VM. To achieve better performance, it is critical to identify the optimal page distribution ratio among SLC, MLC, and HDD. As shown in Figure 4, for latency sensitive VMs, we need to allocate more SLC pages so they can benefit from fast access. For capacity sensitive VMs, we prefer to assign more MLC pages so that the number of page faults can decrease. For checkpointed VMs, part of its memory pages can remain in memory as MLC mode.

In this paper, we explore PCM system design and optimization for platforms that use virtualization technologies. Unlike the existing approaches, our design exposes the emerging PCM device features to the VMM. Meanwhile, it employs the cooperative intra-/inter-VMM memory resource allocation mechanisms to dynamically allocate/balance SLC/MLC PCM pages that manifest different characteristics within and across VMs, resulting in more desirable trade-off between memory capacity and access latency. Moreover, we propose VMM management techniques that leverage the PCM technologies to facilitate key operations such as VM creation, checkpoint/restore and live migration. Figure 5 provides an overview of the proposed cooperative and cross-layer PCM system for effective virtual machine execution.
3.1. VMM Support for Hybrid PCM Page Frame Tuning

At the hardware level, we use PCM that consists of MLC devices. Those devices can operate in either MLC or SLC modes, which differ in both storage capacity and access latency. We implemented three new functions in VMM to interact with the underlying PCM hardware. Among those, Switch_page_type is invoked to switch the operating mode of a physical page between MLC and SLC; MLC_alloc() is responsible for allocating MLC pages; and SLC_alloc() is used for SLC page allocation. Based on the information provided, the memory controller reconfigures the MLC/SLC devices.

Note that the above dynamic page tuning requires that the VMM track physical memory pages in each type. To this end, we extend one bit in machine to physical translation table (in VMM) to indicate whether a page is in SLC or MLC mode. Since only free memory pages can be adapted on-the-fly, two page lists, namely free_MLC_list and free_SLC_list, which track the available MLC and SLC pages, are added to our design. Initially, all the free PCM pages are added to the free_MLC_list. At runtime, the following events will trigger MLC/SLC page tuning: (1) switching a PCM page to a different mode, and (2) requesting PCM pages in a specific mode. For (1), the MLC/SLC PCM manager, which is the software module that converts the SLC/MLC pages through memory controller, will first check the target free list to determine whether there are available pages. If so, it performs a page copying from the source to the destination and reclaims the source page by putting it back into the corresponding free list. If not, the MLC/SLC PCM manager will identify a page in the source free list and convert it to the target page type. It then performs page copying and frees up the unused page to the corresponding free list. For (2), the MLC/SLC PCM manager iteratively invokes operation (1) until it collects sufficient free pages. In both cases, if there are no sufficient free pages, the MLC/SLC PCM manager will discard the request and inform the VMM.

Due to dynamic tuning, the physical address will inevitably be shifted. In order to correctly locate the physical page frame, the SLC/MLC address translation table is implemented to maintain the addresses for both SLC and MLC pages. In addition, as shown in Figure 1, one SLC page can only be converted to two contiguous MLC pages and vice versa. Therefore, the SLC/MLC address translation table can also be used to easily locate the contiguous pages for conversion.

3.2. PCM Page Management in Virtualized Platform

3.2.1. Virtual Memory Management

Our page management consists of two parts: information collector and load balancer. To make the information collector aware of page accesses, we customize the method proposed in [28, 29], which traps memory accesses as minor page faults by giving the pages of interest a higher access privilege. Here, we define two types of minor page faults: SLC minor page fault and MLC minor page fault. The information collector uses both types of minor page faults to gather memory locality statistics by customizing the LRU and Stack Algorithm in [16]. However, only MLC minor page faults will trigger the load balancer to tune the number of SLC pages, MLC pages and the size of balloon driver for each VM. Sampling mechanism [29] is also used to lower the cost of memory tracking. As a result, our tracking and management scheme are disabled periodically and the memory pages remain in their current type during those periods. Figure 6 shows the flow of PCM page management.

In order to form an LRU histogram for virtual memory page usage, we implemented a counter array to track the access frequency for all the pages. If a page is accessed, the corresponding counter will be incremented by 1. Based on the LRU histogram, the miss ratio can be computed as:

$$\text{MR}(m) = 1 - \frac{\sum_{i=1}^{n} \text{Hit}[i]}{\sum_{i=1}^{n} \text{Hit}[i] + \text{Hit}[\infty]}$$

where m is the number of pages provided; n is the total number of available pages; Hit[i] is the number of times that page i is hit; Hit[\infty] is the number of accesses which results in a miss even if all n pages are allocated [3]. Using equation (1), the load balancer can compute the miss ratio curve for each VM, as shown in Figure 3.

Due to temporal locality in memory access, it is desirable to give the most recently used page faster access time (SLC mode). As shown in Figure 7, we implemented two LRU linked lists: i.e. SLC list and MLC list to manage the virtual memory pages. Note that the lengths of those lists are limited by the maximum physical memory size of SLC and MLC pages allocated by the VMM. When a page in the MLC mode is accessed (③), it will first locate another free MLC page (②). Then, they will be put together and converted to one SLC page (①) and inserted at the end of SLC list (⑤). In case that the SLC list reaches the maximum physical SLC memory size and there are more pages coming into the SLC list, the page at the head of SLC list will be de-queued (⑥) and the page will be converted to MLC mode (⑦). At the same time, a free MLC page will be released and inserted at the head of MLC list (⑨). When the MLC list reaches the maximum physical MLC memory size and there are more pages to be evicted from the SLC list, pages at the head of MLC list will be evicted from memory and swapped to disk (⑧).

In order to further lower the cost, upon the SLC minor page faults, we only increment the counter associated with that page.
For each MLC minor page fault, our scheme increments the corresponding counter, updates the list and invokes the load balancer to tune the SLC/MLC pages.

![Virtual Memory Page Allocation](image)

**Figure 7. Virtual Memory Page Allocation**

### 3.2.2. Intra-VM Utility Driven PCM Page Allocation

We apply the utility function based methods [3, 4] to determine two values in miss ratio curve: $m_1$ for the SLC size and $m_2$ for the total memory size, which indicate how many SLC and MLC pages should be allocated respectively to achieve the optimal efficiency.

Note that whenever a VM increases one unit of page in SLC, it needs to trade in two units of page residing in MLC, as illustrated in Figure 1. Therefore, when $m_1$ (SLC) is increased by 1, $m_2$ (SLC+MLC) will decrease by 1. The additional low-latency SLC pages brought by increasing $m_1$ will cause the reduction of total PCM capacity (decreasing $m_2$), which will further lead to increase in page faults. To determine $m_1$ and $m_2$, we define the utility functions as

$$\Delta MR(m) = \frac{MR(m + \Delta m) - MR(m)}{\Delta m}$$

(2)

$$U_{m_1} = -\Delta MR(m_1) \times \text{latency of MLC}$$

(3)

$$U_{m_2} = -\Delta MR(m_2) \times \text{latency of SLC}$$

(4)

$$U_{slc} = U_{m_1} + U_{m_2} \quad \text{(increase $m_1$; reduce $m_2$)}$$

(5)

$$U_{slc} = U_{m_1} + U_{m_2} \quad \text{(reduce $m_1$; increase $m_2$)}$$

(6)

In the above equations, $\Delta MR(m)$ indicates the difference in miss ratio if the page supply is changed by $\Delta m$. $U_{m_1}$ defines the utility of changing pages from MLC to SLC. The latency ratio between SLC and MLC is used to weight the change, i.e. $\Delta MR(m_j)$. Following the same philosophy, $U_{m_2}$ defines the utility of changing pages from HDD to MLC. Recall that when we vary the number of SLC or MLC pages, both $m_1$ and $m_2$ will change. Therefore, $U_{slc}$ defines the utility of morphing two MLC pages into one SLC page, while $U_{slc}$ illustrates the utility of changing one SLC page into two MLC pages. Upon MLC minor page fault, the load balancer will compute $U_{slc}$ and $U_{slc}$ to determine the ratio of SLC/MLC pages within one VM. If $U_{slc}$ is greater than 0, it is worth increasing one unit of SLC by reducing two units of MLC. If $U_{slc}$ is greater than 0, it is worth to morph one SLC page into two MLC pages. If both $U_{slc}$ and $U_{slc}$ are negative, the load balancer will maintain current allocation between SLC and MLC pages, indicating the desirable tradeoff between memory access latency and the number of page faults.

Note that in our design, the abovementioned techniques are integrated into the MMU in the hypervisor. Unlike the hardware-based method [7] that manages PCM systems in the form of circuit level columns and rows, our schemes work at page granularity, which makes it easier to interact with the software layer. Besides, we abstract the utilities of both SLC and MLC for each individual VM. Doing so opens up the possibility to further exploit PCM features in virtualized environments.

### 3.2.3. Inter-VM PCM Resource Adaptation

In highly utilized virtual servers, asymmetric memory resource allocation is common: some VMs run memory intensive jobs with less available memory space while other VMs are idle with plenty of unused memory pages. Therefore, only tuning SLC/MLC pages within each VM cannot achieve the optimal memory efficiency. This motivates us to further extend our intra-VM page balancer to support better memory utilization across VMs.

Recall that VM balloon technology makes it possible to resize memory for individual VM by changing the size of balloon within guest VMs. Our technique leverages this balloon feature to reallocate memory resources across different VMs. We refer the VM that demands more memory as a caller VM and the VM from which the memory is being deprived as a victim VM. In case a VM runs out of PCM SLC/MLC pages but still benefits from providing more pages (as indicated by the utility function), it will become a caller VM. Upon a request, the VMM will check other VMs for available pages based on their utility functions ($U_{slc}$ or $U_{slc}$). Since a VM can benefit from increasing the number of a given type of PCM page if the corresponding utility value is positive, only the one with the lowest negative utility value will be selected as victim VM. Doing so will solve the asymmetrical memory distribution problem. Note that VM priority can also be incorporated into our proposed utility function to solve the issue when there are multiple callers. In the case of over-commitment (i.e., the total number of requested memory pages exceeds the memory capacity when all the pages are in MLC mode), there will be no victim VM in the system. Our scheme will select the VM with lower priority and swap their MLC pages to disk. In this study, all the VMs are assumed to have the same priority for simplicity and arbitration is done in a round robin fashion.

The runtime overhead of our proposed scheme includes maintenance overhead for the minor page faults and the page table. Compared with real page faults, which invoke disk IO access, these maintenance activities only invoke memory or memory-to-memory operations, which are less expensive. In addition, during most of the execution period, applications only access their working set pages (in SLC mode). On a SLC minor page fault, our scheme only sets the hit counters, which takes O(1) time. Therefore, most of the time, the load balancer is not triggered and the overhead is low. Coupling with sampling mechanism [29] further lowers the overhead in real systems.
3.3. Benefits for VM Dedicated Behaviors

VM dedicated behaviors include VM creation, shutdown, normal execution, checkpoint/restore, and live migration. VM creation and shutdown can be treated as a normal running VM, which can directly benefit from our technique. VM checkpoint/restore, which is the basis for VM live migration, can also take advantage of the PCM non-volatile feature. Instead of dumping the entire memory to disk, part of the checkpointed memory can stay in memory to reduce IO traffic.

Based on the non-volatility of PCM and the IntraVM+InterVM schemes, we redesign the checkpoint/restore routine. In the checkpoint routine as shown in Figure 8, once the checkpoint command is issued, a preparation signal is sent to checkpoint VM to inform the guest OS (①). Meanwhile, VMM collects page allocation information from the information collector (in section 3.2.1) (③) and classifies the pages into 4 categories: Working Set pages (SLC page), Page Table Pages, Non Working Set Pages (MLC Pages) and Pages with VM State (vCPU, Device) (③). Working Set Pages, Page Table Pages and VM State Pages will be transferred to MLC pages and marked as checkpoint pages; Non Working Set pages will be dumped to disk (③). A checkpoint_page_list is implemented to track those checkpoint pages in memory (③). The freed pages will be put back into the free_SLC_list and free_MLC_list. In the restore routine as shown in Figure 9, the VMM first reads in the checkpoint_page_list and locates the checkpoint pages (③). And then, VM state information is used to create an empty VM while Page Table Pages and Working Set Pages will be transferred to SLC page (②③). At this point, we leverage the lazy restore concept [5] to resume the VM in advance. Since the VM is running with its minimal requirement (VM state, Page Table Pages and Working Set Pages), it is possible that Non Working Set Pages will be accessed and cause page fault during the execution. To solve this issue, whenever there is a page fault, the fetch on demand process [26] will be invoked to request an SLC page from the free_SLC_list and bring the pages from disk to memory (⑧). At the same time, the Information Collector and Load Balancer will be invoked to perform PCM page management as discussed in Section 3.2.

Note that there are two caveats when keeping checkpoint pages in MLC pages: 1) When active VMs are greedy for memory capacities, memory pages for the checkpointed VM will be dumped to disk and later restored using traditional schemes. 2) The availability of checkpoints can be maintained by adopting architectures similar to RAMCloud [30], which are designed to share the memory system among several servers (i.e., as SAN for disk). In that case, VM live migration will work just as the checkpoint/restore scheme and the disk IO is replaced by the network IO operations. Since the network IO operations are more expensive than disk IO operations, the benefit for VM live migration is expected to be more significant than that for checkpoint/restore.

Compared with traditional checkpoint/restore schemes, our approach significantly reduces the amount of IO requests by saving some memory pages in PCM. The benefits include 1) improving the checkpoint/restore time by keeping the critical data in memory, and 2) reducing the IO penalty for other running VMs by reducing the amount of data written to disk. Furthermore, when using our approach, the checkpoint/restore latency is no longer proportional to the memory size but depends on the ratio of SLC/MLC pages (determined by the size of working set).

![Figure 8. The Save Stage of Checkpointing](image)

![Figure 9. The Restore Stage of Checkpointing](image)

The implementation overheads for our PCM system design are summarized as: 0.2% of memory space for SLC, MLC Page lists and 0.1% for the counter array. 95% of total minor page faults are SLC minor page fault, which only takes 1us to set the counter. Only 5% of total minor page faults are MLC minor page fault, which takes 1ms, on average, to update the linked list and trigger the load balancer.

4. Experimental Setup

4.1. Simulated Hardware Platform

We evaluate the efficiency of our design using Simics full-system simulator [17] with g-cache timing model and further integrate it with DRAMSim2 memory subsystem simulator [18]. Since Simics does not simulate paging functionality, we implemented a new simulator module [19] to model the behavior and timing cost for paging. We further extend it with our proposed PCM page
management scheme and the associated overheads are counted in our modeling. The sampling interval is determined via empirical approach. DRAM is simulated as cache on top of PCM. We express PCM device and circuit characteristics using conventional DDR timing parameters. This allows characterizing PCM in the context of more familiar DRAM parameters [20]. Table 1 lists the configuration of simulated machine and the average access latency on SLC, MLC and HDD, which are consistent with recent published work [7][20][21][22].

Table 1. The Configuration of Simulated Machine

<table>
<thead>
<tr>
<th>Processor</th>
<th>4-OOO-core, 2GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Private L1 Cache</td>
<td>16K I-cache and 16K D-cache, 64-byte lines, 4-way set associative</td>
</tr>
<tr>
<td>Private L2 Cache</td>
<td>1MB, 64-byte lines, 4-way set associative</td>
</tr>
<tr>
<td>DRAM Cache</td>
<td>4MB, write-back, 50 cycle access time (25ns)</td>
</tr>
<tr>
<td>Main Memory Size</td>
<td>600MB for page management experiments; 2~16GB for checkpoint experiments</td>
</tr>
<tr>
<td>PCM SLC Page</td>
<td>Reads: 250 cycles (125ns); Writes: 700 cycles (350ns)</td>
</tr>
<tr>
<td>PCM MLC Page</td>
<td>Reads: 500 cycles (250ns); Writes: 1700 cycles (850ns)</td>
</tr>
<tr>
<td>Page Conversion</td>
<td>SLC to MLC: 1950 cycles; MLC to SLC: 1200 cycles</td>
</tr>
<tr>
<td>Hard Disk Drive</td>
<td>1500K cycles access time</td>
</tr>
<tr>
<td>Memory to Memory Copy</td>
<td>1200~2000 cycles access time</td>
</tr>
</tbody>
</table>

4.2. Virtualization Layer

We port Xen 4.1.0 [9] together with Debian Linux 2.6.26 kernel with PAE support into Simics. In order to track VM behaviors such as memory update, domain switches and checkpoint, we also instrument the Xen hypervisor and Memory Management Unit (MMU) in Linux kernel with the HAPs and Magic Instructions, which are APIs provided by Simics for communication between software and simulated machines. Shadow page table is enabled and magic instructions are instrumented into minor page fault interrupt service routine so that all the minor page faults will be passed to Simics. In order to track the behavior of checkpoint, we insert magic instructions into xc_domain_save.c and xc_domain_restore.c, which are the source codes for VM checkpointing. By doing this, the simulated machine is aware of all VM behaviors. On booting, Xen starts up a control domain called dom0, from which we create several guest VMs (domU) and launch our workloads.

4.3. Workloads

To emulate various memory access patterns in virtualized data center servers, we use a mix of SPEC 2006 and PARSEC benchmarks in our simulations. We boot up a maximum of 6 domains in Xen and mix multiple benchmarks to emulate the following virtualization scenarios: one VM runs one benchmark, one VM runs multiple benchmarks, multiple VMs run one benchmark each, and multiple VMs run multiple benchmarks each. Table 2 summarizes the simulated workload execution scenarios. To measure the performance of VM checkpoint operations, we run SPEC 2006 and PARSEC workloads (as summarized in Table 3) in each VM.

5. Evaluation Results

In this section, we begin by characterizing the performance of phase change memory system on virtualized platforms and existing hardware-based hybrid PCM system. Then, we evaluate the performance of pure MLC-based (pure-MLC), pure SLC-based (pure-SLC), and hybrid SLC/MLC with Intra-VM management (IntraVM) systems under symmetric memory allocation. We further incorporate VM-aware dynamic memory balancing with Intra-VM management (IntraVM+InterVM) and compare it with IntraVM under asymmetric memory allocation scenarios. In the end, we evaluate the efficiency of our checkpoint schemes in terms of data transfer time and the IO performance penalty on other running VMs.
5.1. Characterizing Hardware-based PCM Design

We first analyze the impact on memory latency when increasing the capacity of phase change memory system by morphing SLC pages into MLC pages. Figure 10 (a) shows that the number of page faults reduces by 30% on average as more SLC pages are replaced by MLC pages. On the other hand, MLC pages increase the memory latency by 3.5X as well, as shown in Figure 10 (b). The optimal operation point, on which the system gains the most benefit from SLC pages while suffering the least page fault penalty, is the lowest point for each line in Figure 10 (c). Figure 10 (c) shows that the execution time decreases as we increase memory capacity at the initial stage. As the system exhausts the benefit of reducing page faults, it suffers significantly from the high latency MLC pages. As can be seen in Figures 10, some of the benchmarks (e.g. povtray-1-dom) are affected dramatically by memory capacity (up to 90% page fault reduction) while some benchmarks (e.g. milc-1-dom) are not (less than 5% page fault reduction). Therefore, the optimal operating point varies across different workloads. It is critical for the system to identify the corresponding optimal operating point on the fly.

Note that existing hardware-based hybrid PCM design [7] uses memory controllers to tune the ratio of SLC/MLC cells in circuit level. We evaluate the performance of such design in virtualized execution scenarios and the results are shown in Figure 11. As can be seen, the hardware-only approach can reduce total execution time by 10% compared to pure-SLC system and 18% compared to pure-MLC system.

5.2. IntraVM PCM Page Allocation Scheme

In this section, we only turn on the IntraVM function and compare it with pure-MLC, pure-SLC and hardware-based hybrid PCM schemes. The total execution time for those designs is shown in Figure 12.

We observe that pure-MLC system is about 1.47X slower than pure-SLC system while it reduces 11% page faults. The penalty of page faults is much more expensive than the memory access time, as indicated in Table 1. The performance of pure-SLC and pure-MLC systems varies across different benchmarks. As can be observed in Figure 12, by enabling IntraVM scheme, page faults penalty is reduced by 8%, which results in 10% improvement in total execution time compared with pure-SLC based system. On the other hand, compared with pure-MLC based system, the IntraVM scheme improves 43% in memory stall time, which leads to 20% improvement in total execution time. Although designed at the software layer, the IntraVM scheme yields equally well performance compared to hardware-based hybrid PCM system. This is because 1) both designs seek the optimal ratio of SLC/MLC in the case of single VM; 2) hardware-based solution is triggered much more frequently than IntraVM, which makes accumulated overhead equivalent to our design.

Furthermore, our IntraVM design can be improved by incorporating advanced page replacement algorithms. Theoretically, Belady’s optimal algorithm [27] can achieve the best performance by leveraging future page access information. Nevertheless, LRU algorithm is a more practical method, which yields relatively good estimation. We use Belady’s optimal algorithm as the theoretical upper bound of our design, which is shown as IntraVM Upper Limit line in Figure 12. As can be seen, by deploying the Belady’s optimal algorithm, total execution time of IntraVM design can be further reduced by around 25%.

By further analyzing the results in Figure 12, we found that: 1) the major component of execution time for latency sensitive VMs (e.g. povtray-1-dom, GemsFDFTD-1-dom) is memory latency so that the impact of improving memory stall time overwhelms page fault penalty; 2) page fault penalty dominates the capacity sensitive VMs (e.g. lbm-1-dom, MSXM-1-dom) so that performance is improved by reducing the number of page faults. Figure 13 presents the page distribution of both latency and capacity sensitive VMs. On latency sensitive VMs (e.g. povtray-1-dom), the IntraVM scheme dynamically assigns more SLC pages to improve the memory stall time. In contrast, IntraVM scheme assigns more high capacity MLC pages (around 94% of total memory pages) to reduce the page faults on capacity sensitive VMs (e.g. mcf-1-dom). It is also observed that capacity sensitive VMs consume 1.4X memory pages compared with latency sensitive VMs. Although the dominant part of execution time for different types of VMs varies, our IntraVM scheme successfully adapts itself across different types of VMs and shows better performance across all the cases.

![Figure 10](image1.png)

**Figure 10.** The Effect on (a) Page Faults, (b) Memory Latency, (c) Overall Execution Time when Memory Capacity Increases for Single VM (Normalized to Pure SLC System)

![Figure 11](image2.png)

**Figure 11.** The Total Execution Time for Hardware-based Hybrid PCM (Normalized to Pure SLC System)
5.3. IntraVM+InterVM PCM Page Allocation Scheme

Although not exhibiting huge advantages over hardware-based hybrid PCM system, the IntraVM scheme shifts the PCM memory management to software level, which opens the possibility to leverage the full system information (e.g. different memory demands from different VMs) and system behaviors (e.g. memory reallocation among different VMs).

To fully leverage the full system information and resources, we further incorporate VM-aware management and enable both InterVM and IntraVM schemes in our system (IntraVM+InterVM). We select workloads with at least two VMs running and create asymmetric memory allocation by 1) running the same benchmarks in different VMs with different memory supplies, 2) running different benchmarks in different VMs with the same memory supply for each VM.

Figure 14 shows page miss ratio of two VMs running different benchmarks (e.g. mcf-milc). As can be seen, when IntraVM scheme is enabled, the optimal SLC/MLC ratio within one VM is located and shown as dashed line. However, in this case, $U_{SLC}$ for milc-VM is negative while $U_{SLC}$ for mcf-VM is positive. With IntraVM+InterVM scheme, mcf-VM borrows SLC pages from milc-VM and converts them into MLC pages as shown in Figure 15. By doing this, the number of page faults for mcf-VM is reduced due to extended MLC pages. Meanwhile, the number of page faults for milc-VM will not increase because it still has enough SLC pages to keep low page miss ratio. Upon this situation, the overall page faults will be reduced, which results in significant performance improvement (20% better than IntraVM scheme).

Besides, we found that capacity intensive VM (e.g. mcf-VM) tends to grab MLC pages from other VMs while latency sensitive VM (e.g. milc-VM) has intention to borrow SLC pages from other VMs.

On the other hand, Figure 16 shows two VMs have different memory supplies but execute the same workloads (e.g. BGC-2-dom). With IntraVM scheme, the VM with smaller available memory size (VM$_1$) will have to suffer from higher page faults than the VM with larger memory size (VM$_2$). In this case, the $U_{milc}$ for VM$_1$ is positive while $U_{mcf}$ for VM$_2$ is negative. With IntraVM+InterVM scheme, VM$_1$ borrows MLC pages from VM$_2$ as shown in Figure 17. By doing so, the memory difference between two VMs becomes smaller and the overall page faults for entire system are reduced by around 25%, which leads to 40% performance improvement in total execution time. In addition, we found that MLC pages have higher possibility to be shifted between VMs than SLC pages in the situation that VMs have different memory supplies at the beginning.

In Figure 18, we classify our benchmarks into 3 groups. In Group 1, we run same workloads in different VMs with different memory supplies. As can be seen, the total execution time is improved by around 32% on average. In Group 2, we run the same workloads in VMs with same memory configuration but start them at a different time. Only 4% of improvement can be achieved in this situation because there are no idle pages to be tuned among VMs. In Group 3, we run different workloads inside different VMs with same memory configuration. At this situation, 10% of performance improvement can be achieved by enabling IntraVM+InterVM scheme. Averaging all the benchmarks, our scheme can yield a 15% performance improvement compared to the hardware-based method [7]. Furthermore, we found that the larger the difference between memory supply and memory demands is, the better our schemes will perform.
5.4. Checkpoint/Restore Efficiency

This section evaluates two major impacts of checkpointing on virtualized system: 1) the time it takes to checkpoint and restore a VM and 2) the IO penalty for other normal running VMs upon checkpointing.

To evaluate the effect of reducing checkpoint/restore time, we retrieve the memory allocation information just before a checkpoint (save stage) is made. We observe that, on average, only 46% of memory pages are in MLC mode (as shown in Figure 19), which means only 46% of memory pages need to be dumped to the disk. In Figure 20, by replacing expensive IO operations with much cheaper memory-to-memory copy operations (as shown in Table 1), our approach achieves around 46% improvement in data transfer time compared with contemporary DRAM-based method [5]. Further analysis shows that the improvement varies from 40% to 75% across different workloads and the results are highly related to the size of the working set (pages in SLC mode) before checkpointing. As can be seen in Figure 19, the memory distribution before checkpointing varies across different workloads. Some of the workloads exhibit large working sets (about 70% pages in SLC mode), while some of them have small working sets (about 30% pages in SLC mode). Workloads, which have larger working sets, gain larger improvement in data transfer time (see Figures 20 and 21). If the entire VM memory is in SLC mode before checkpointing happens, only memory-to-memory copy operations are triggered since we only consolidate the SLC pages to MLC pages. The average data transfer time is only 0.03% of traditional methods. On the other hand, if the entire VM memory is in MLC mode before checkpointing occurs, the data transfer time will be close to that of conventional DRAM-based methods (shown as dark black line in Figure 20) since our scheme opt to dump all the pages in MLC mode to disk. Although both of aforementioned scenarios rarely occur in our experiments, they provide the upper and lower boundaries of our approach.

To further evaluate the side effect of checkpoint/restore on other running VMs, we measure the IOPS of one VM while the other one is being checkpointed. Note that the longer the checkpoint/restore is, the longer the running VM will suffer from low IO bandwidth. We use \( \text{IO Penalty} \times \text{Time} \) to indicate the total IO penalty the running VM will suffer. As can be seen from Figure 21, as the memory size increases, the overall IO penalty for the running VM increases. However, the overall IO penalty for our proposed PCM system is (22%) lower than the DRAM-based system because our PCM with IntraVM+InterVM scheme takes less time to checkpoint/restore. Further comparing (a) and (b), we found that the improvement of our scheme is more significant in latency sensitive VMs (34%) than capacity sensitive VMs (13%) because there is less MLC pages to be dumped to disk in latency sensitive VMs. Figure 22 shows IO penalty for running VM when the number of checkpointing VMs increases. As can be seen, a PCM system equipped with IntraVM+InterVM scheme also has (50%) lower IO penalty than a DRAM-based system because the overall IO traffic generated by checkpointing or restoring is less in our schemes than the traditional DRAM-based schemes. As compared in Figure 22 (a) and (b), Latency Sensitive VM can benefit more from our scheme than Capacity Sensitive VM due to higher percentage of SLC pages. Besides, the improvement increases as the number of VM increases.
cross-layer software/hardware approach and our PCM page management unit makes decisions based on the full-system execution characteristics. Instead of just using OS balloon technology to guarantee the correctness upon memory change, our design further extend the functionality of VM balloon technology to gain performance improvement. We evaluate the efficiency of applying SLC/MLC PCM memory design on virtualized platforms and shows that our design yields 15% performance improvement compared to [7]. Furthermore, the non-volatile feature of PCM is not considered in [7]. [12] proposed to dynamically monitor the memory usage of each virtual machine, predicts its memory needs, and periodically reallocates the memory. Our work differs from [12] in that we propose VMM support for SLC/MLC PCM system and our page management considers both SLC and MLC PCM. [29] proposed a low cost working set tracking scheme, which can vary the sampling period based on the working set size. This work can be incorporated in our sampling scheme to further lower the cost of memory tracking. [5] proposed a method to perform fast VM restore based on working set estimation. Their method only loads the active working set back in memory and starts restore using a lazy scheme. Nevertheless, [5] is based on DRAM while we take both SLC and MLC PCM into consideration. There are other ways, such as page hashing [24] and copy on write [25], to improve checkpointing performance. These methods are orthogonal to our approach and can be combined with our proposal. [31] proposed to use 3D PCM to store checkpoint in MPP system. Our work differs from [31] in that we analyze the working set pages for each virtual machine and make checkpoints based on the detailed page distribution from our PCM page management scheme.

7. Conclusion
Virtualization improves resource utilization at the price of lowering the performance and efficiency of the memory subsystem. The non-volatile and resizable features make PCM a desirable candidate for designing adaptive and intelligent memory systems on virtualized platforms. The goal of this work is to better incorporate PCM technology into virtualized environments. To achieve this goal, we propose 1) hypervisor-level, collaborative management of non-volatile memory and storage (SLC/MLC PCM memory and hard disk), 2) a VM-aware SLC/MLC page balancer, 3) SLC/MLC page tuning in cooperation with VM ballooning to improve the performance of critical VM operations, and 4) leveraging the non-volatile property of PCM to reduce IO requests. Our experimental results show that the proposed techniques can improve the overall performance by 10% compared with pure-SLC systems and 20% compared with pure-MLC systems. Introducing advanced page replacement algorithm to IntraVM design can yield 25% performance improvement. By incorporating VM-aware optimization, the overall performance can be further improved by 15%. In terms of checkpointing, our design achieves 46% less checkpoint/restore time and reduces around 50% of overall IO penalty to the system.

6. Related Work
[7] proposed Morphable Memory Systems (MMS), which trades off latency with capacity in a SLC/MLC PCM memory system using hardware based approach. To achieve this, MMS uses a memory monitoring hardware to collect the physical addresses issued to the memory controller and further estimates the size of SLC. Our work differs from [7] in that we use cooperative and
Acknowledgement

This work is supported in part by NSF grants 1117261, 0937869, 0916384, 0845721(CAREER), 0834288, 0811611, 0720476, by SRC grants 2008-HJ-1798, 2007-RJ-1651G, by Microsoft Research Trustworthy Computing, Safe and Scalable Multi-core Computing Awards, by NASA/Florida Space Grant Consortium FSREGP Award 16296041-Y4, and by three IBM Faculty Awards.

References

[14] G. Vall’ee et al., Checkpoint/Restart of Virtual Machines Based on Xen, HAPCW 2006.
[17] Simics full system simulator www.virtutech.com/
[18] DRAMSim2: http://www.ece.umd.edu/dramsim/