DUANG: Fast and Lightweight Page Migration in Asymmetric Memory Systems

Hao Wang[2], Jie Zhang[3], Sharmila Shridhar[2], Gieseo Park[4], Myoungsoo Jung[3], Nam Sung Kim[1]

[1]University of Illinois, Urbana-Champaign
nskim@illinois.edu

[2]University of Wisconsin, Madison
{hwang223, sshridhar}@wisc.edu

[3]Yonsei University
School of Integrated Technology
Yonsei University Convergence Technology
{jie, m.jung}@yonsei.ac.kr

[4]University of Texas, Dallas
gieseo.park@utdallas.edu

Abstract

Main memory systems have gone through dramatic increases in bandwidth and capacity. At the same time, their random access latency has remained relatively constant. For given memory technology, optimizing the latency typically requires sacrificing the density (i.e., cost per bit), which is one of the most critical concerns for memory industry. Recent studies have proposed memory architectures comprised of asymmetric (fast/low-density and slow/high-density) regions to optimize between overall latency and negative impact on density. Such memory architectures attempt to cost-effectively offer both high capacity and high performance. Yet they present a unique challenge, requiring direct placements of hot memory pages[1] in the fast region and/or expensive runtime page migrations. In this paper, we propose a novel resistive memory architecture sharing a set of row buffers between a pair of neighboring banks. It enables two attractive techniques: (1) migrating memory pages between slow and fast banks with little performance overhead and (2) adaptively allocating more row buffers to busier banks based on memory access patterns. For an asymmetric memory architecture with both slow/high-density and fast/low-density banks, our shared row-buffer architecture can capture 87-93% of the performance of a memory architecture with only fast banks.

1. Introduction

DRAM has been used for the main memory system for decades due to its high performance and low cost. Although DRAM's capacity and bandwidth have dramatically increased over years, DRAM's random access latency has remained almost constant (i.e., more CPU cycles per access). This makes memory latency a performance bottleneck in many computer systems [1]. Furthermore, as DRAM scaling becomes limited by less reliable charge storage and data sensing, resistive memory technologies, such as phase change memory (PCM) [2, 3, 4] and magnetoresistive random-access memory (MRAM) [5], have been proposed as a promising alternative to DRAM. However, main memory systems using such technology can still remain as a performance bottleneck.

Trade-off between latency and density: In principle, as more memory cells are integrated in an array for higher density, it takes more effort to retrieve states from the cells in the array. This is the prime reason for long latency of the DRAM-based main memory system pursuing high density [6]. Moreover, resistive memory has another latency and density trade-off aspect associated with its unique property that allows us to store one or more bits per cell (i.e., a single- or multi-level cell (SLC and MLC)). MLCs can offer 2 or 3× higher capacity at the expense of considerably longer latency than SLCs in particular for write operations [7, 8].

Asymmetric memory architecture: Since the memory industry considers achieving high density as its top priority, it is usually not willing to sacrifice density for latency. Thus, recent studies have proposed memory architectures comprised of asymmetric sub-arrays [1] or banks [6], attempting to optimize between overall latency and negative impact on density. This leads to fast and slow regions within a single memory device. The same concept can be applied to resistive memory where some are used as SLC banks and others as MLC banks to constitute fast/small and slow/large regions. Observing that a relatively small fraction of memory pages are frequently accessed and thus performance-critical [9, 10], an asymmetric memory architecture has the potential to achieve both high density and low latency by placing performance-critical (hot) pages in the fast region.

Page placement and migration: However, there are two undesirable aspects in the current asymmetric memory architectures. First, the asymmetry is often determined at design time [1, 6] and thus it cannot be adapted depending on given program-specific memory access patterns at runtime. This in turn requires a smart page placement mechanism to place all hot pages in the fast region with limited capacity. However, determining the hot pages remains as a key challenge since page placement should be decided when a program accesses an unmapped virtual address for the first time [11]. In other words, a processor has no runtime opportunity to observe the memory access patterns associated with

[1] In this paper, a page denotes a block of bytes that can be stored in a row buffer, while a frame refers to a basic unit of virtual-to-physical memory mapping in operating system's terminology.

978-1-4673-9211-2/16/$31.00 ©2016 IEEE

the page to make an informed decision on where to place the page. This necessitates a page migration when a page is placed in the undesired (slow) region at first. However, page migration at runtime can be very costly in terms of both performance and energy consumption due to lack of a high-bandwidth path between fast and slow regions. A simple approach is to read the entire page to the processor through a memory channel and write it back to another location. This will occupy the memory channel for 640ns (i.e., more than 2000 4GHz-CPU cycles) to transfer a 4KB frame[1] through DDR3-1600. Although an in-DRAM bulk data movement technique such as RowClone may alleviate the overhead of migrating a page within one single sub-array [12], it is much less efficient when the target system has to migrate a page to a different bank.

Shared row-buffer architecture: Unlike commodity DRAM architecture using cross-coupled inverters for both sensing and buffering data, resistive memory architecture decouples sensing and buffering and provides multiple row buffers per bank to improve row-buffer locality and bank concurrency [2, 4]. In this paper, building upon a baseline architecture with two row buffers per bank depicted in Figure 1(a), we propose a novel row-buffer architecture for resistive memory. Figure 1(b) illustrates our row-buffer architecture that physically shares four row buffers between a pair of two neighboring banks at design time, but logically partitions the row buffers and adaptively allocates them between the two banks at runtime. This shared row-buffer architecture enables the following two attractive techniques:

(1) Lightweight page migration: Figure 2(a) depicts an asymmetric memory architecture where the upper and lower banks use MLCs and SLCs for high capacity and low latency, respectively. As discussed earlier, such a memory architecture requires either an oracle page placement mechanism or a magical page migration technique to efficiently service memory requests. Leveraging our shared row-buffer architecture, we propose a practical technique to migrate pages between two neighboring MLC and SLC banks with little performance overhead. As shown in Figure 2(b), a request to a memory page in an MLC bank will bring the page into one of shared row buffers. The memory controller then labels the row buffer as one that logically belongs to the SLC bank. When the memory controller decides to evict the page from the row buffer later, it will write back the page to the SLC

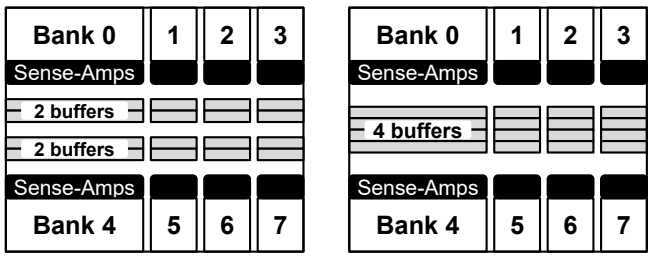


Figure 1. (a) Baseline architecture that has 2 row buffers per bank; (b) our shared row-buffer architecture.

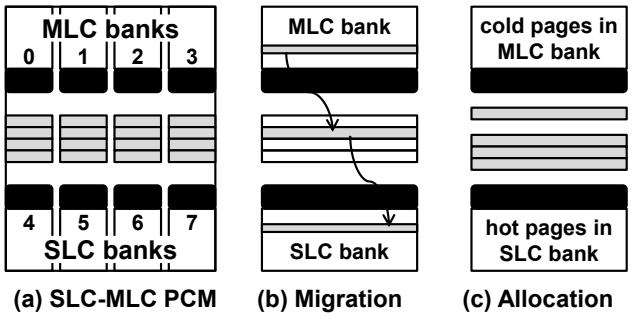


Figure 2. Support page migration and row-buffer allocation in asymmetric memory architecture.

bank. In contrast to existing asymmetric memory architectures, our approach allows almost free page migrations as it does not need to either explicitly transfer pages or occupy a memory channel (and memory internal global I/O bus) during the transfers. With our lightweight page migration technique, the asymmetric architecture (Figure 2(a)) can capture 77%-89% performance of an SLC-only architecture.

(2) Adaptive row-buffer allocation: Our lightweight page migration technique migrates more hot pages to fast SLC banks which in turn service much more memory requests than slow MLC banks. This can reduce the effective bank-level parallelism and increase the pressure on SLC banks. To compensate for these negative effects, the memory controller can take one of two row buffers from an MLC bank and assign it to the paired SLC bank, as shown in Figure 2(c). This technique can even outperform the baseline row-buffer architecture with oracle placements of hot pages in SLC banks from the beginning. In a sense, this adaptive row-buffer allocation provides a way to configure the asymmetry adaptively based on memory access patterns at runtime. A joint application of our adaptive row-buffer allocation and lightweight page migration techniques can capture 87%-93% performance of an SLC-only architecture (4%-10% higher performance than the lightweight page migration technique only), *even without accounting for the performance benefit of larger capacity*.

The rest of this paper is organized as follows. Section 2 discusses current asymmetric memory architectures and their challenges. Section 3 details our shared row-buffer architecture. Section 4 describes the lightweight page migration and adaptive row-buffer allocation techniques enabled by the shared row-buffer architecture. Section 5 briefs the experimental methodology. Section 6 presents evaluations. Section 7 provides various discussions on our architecture. Section 8 describes related work. Section 9 concludes the paper.

2. Asymmetric Memory Architecture

2.1 Asymmetric DRAM Architecture

Previous studies proposed two asymmetric DRAM architectures. Lee et al. proposed a tiered-latency DRAM architecture [1]. It divides a sub-array into two regions using isolation transistors. To read a page from the region near to the sense amplifiers, the isolations transistors are turned off to

disconnect the cells in the far (slow) region from the bitlines. This reduces the bitline capacitance and thus access latency of the near region. In another study, Son et al. proposed an asymmetric DRAM bank organization [6]. Some of DRAM banks are comprised of high-aspect-ratio mats (fewer cells per local bitline) and have their column decoders and sense amplifiers close to the I/O pads to enable the fast region.

The tiered-latency DRAM architecture employs an efficient mechanism to migrate memory pages between the fast and slow regions within a single sub-array. However, the fast and slow regions are not really operating independently, which can degrade the effective performance of fast regions. For example, a preceding request to a slow region can delay a subsequent request to a fast region in the same bank. Therefore, although every PCM cell can be possibly programmed as either SLC or MLC [13] and hence it is possible to have SLC and MLC rows within a single bank, we use an asymmetric architecture with separate SLC and MLC banks, as shown in Figure 2(a). This asymmetric bank organization can exploit the full performance benefit of the fast region as they are independent banks; however, there is no efficient way to transfer data between two banks.

2.2 Hybrid SLC-MLC PCM

PCM employs a chalcogenide alloy such as Ge2Sb2Te5 (GST), which can represent two unique structural phases based on the degree of heating temperature: (i) amorphous and (ii) crystalline phases [14]. Unlike typical charge-based memories (e.g., DRAM), PCM can store multiple bits into a single memory core cell by allowing more resistance levels between the amorphous and crystalline phases, as shown in Figure 3. Consequently, it requires more effort to retrieve states from MLCs than SLCs, leading to longer read latency. Further, it takes even longer latency to store states to MLCs because MLCs have a smaller region of programmed resistance than SLCs, as depicted in Figure 3 [15]. A single programming pulse signal used for SLCs is not a solution to program multiple resistance levels for MLCs. Instead, an iterative multi-level programming technique is typically employed, but this increases write latency even further [7].

Table 1 tabulates the timing parameters of SLC and MLC PCMs using the DDR3-1600 interface. First, tRCD is the PCM read latency while tRP in fact represents write latency on the PCM array since previously buffered data have to be written back to the array if they are dirty. The row decoding latency is from [2] whereas the cell sensing latency is

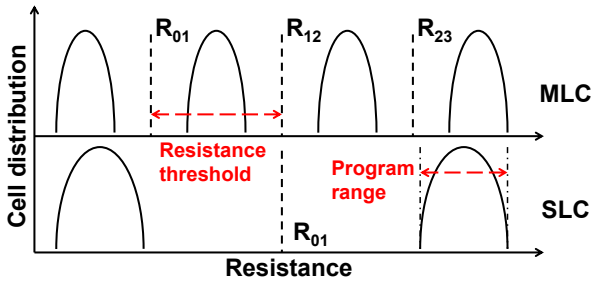


Figure 3. SLC- and MLC-based PCMs.

Table 1: PCM timing parameters based on DDR3-1600.

Timing params (cycle)	SLC	2-bit MLC	3-bit MLC
tRCD (read latency)	50	94	138
tRP (write latency)	120	864	1040
tCAS	11	11	11
tWL	8	8	8
tCCD	4	4	4
tWR	12	12	12
tRTP	6	6	6
$tRRD_{ACT}$	2	2	2
$tRRD_{PRE}$	10	6	7
tREFI	infinite	7.4e10	3.0e7
tRFC	0	6656	8064

derived based on the cell biasing circuit described in [16] and partially verified against real devices [17] (i.e., real 3-bit MLC device not available) with 8.5% variance. Note that tRCD and tRP in Table 1 do not include the latency of driving data between sense amplifiers of a bank and one of multiple row buffers, which is discussed later in Section 3.2 in more detail. Second, $tRRD_{ACT}$ and $tRRD_{PRE}$ specify the delays between two consecutive *activate* and *write-back* commands, respectively. Finally, in contrast to SLC PCMs, MLC PCMs have much smaller resistive margins that may not be enough to tolerate the resistance drift over time, as shown in Figure 3. To avoid the potential of data corruption, MLC PCMs require periodical refresh operations based on data retention time. The refresh process is defined by tREFI and tRFC, which are derived based on [18] and [19].

As illustrated in Figure 2(a), we can use four SLC banks and four MLC banks in a PCM device. Compared with SLC-only PCM, we can increase the total capacity while potentially approaching the performance of a memory system based on SLC-only PCM.

2.3 Page Placement and Migration Challenges

It is very challenging to precisely determine which pages are hot and then place the hot pages in fast banks at initial page placement time. First, as discussed in Section 1, it is not viable to place a page after observing its memory access patterns [11]. Thus, typical hardware-based profile-and-predict techniques are not applicable for such a purpose. Second, a page that is frequently requested by memory instructions is not necessarily a hot page in the memory, due to machine-specific filtering effect of on-chip caches [20]. Hence, static compiler-based profiling techniques (e.g., [21]) may not be effective. Finally, either static or dynamic profiling techniques cannot capture the interference effect of arbitrary co-running applications on the shared last-level cache.

Any history-based page selection policy requires a dynamic page migration. There are various page selection policies regarding how to track page accesses and rank the pages [9, 20, 22, 23, 24, 25]. In the end, a dynamic page migration is needed to fulfill the page selection results. However, transferring a page through a relatively low-bandwidth memory channel can incur a significant performance penalty. Fur-

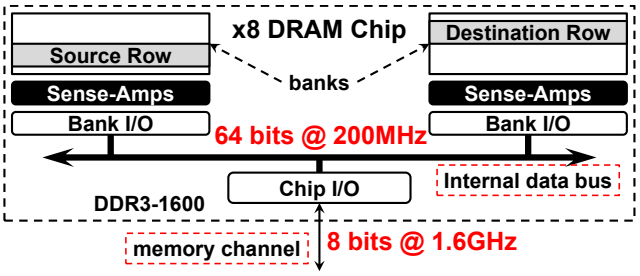


Figure 5. Inter-bank row copy proposed by RowClone.

thermore, the high cost of page migrations limits the minimum length of each epoch for fine-grained adjustments and exacerbates the negative impact of imperfect page selection policies on memory system performance. Although we can attempt a finer-grained migration (e.g., a 64-byte block instead of a page) to reduce the latency of each migration, the cost of storing and tracking all re-mapping information will substantially increase.

A recent proposal (i.e., RowClone) provides two mechanisms to copy pages between two rows within a DRAM device [12]. The first mechanism (denoted by fast-parallel mode (FPM)) can migrate a page within a single sub-array in a very efficient way without any changes to the architecture and interface of commodity DRAM devices. A row copy operation can be done by simply sending two consecutive *activate* commands to the source and destination rows, respectively. The second mechanism (denoted by pipelined serial mode (PSM)) can migrate a source row in one bank to a destination row in another bank. It exploits the DRAM internal global I/O bus shared by all the banks, as depicted in Figure 5. Although the internal global I/O bus is wider (e.g., 64 bits per ×8 DRAM device or 64 bytes per rank comprised of 8×8 DRAM devices), it operates at a lower frequency (e.g., 200MHz for DDR3-1600 [26]). In fact, the bandwidth of in-memory migration (64 bits per 200MHz cycle) is the same as the off-chip I/O bandwidth (8 bits per 1.6GHz cycle). Since the round-trip migration process now changes into a one-way process, the migration latency is cut by half, but it is still over 1000 4GHz-CPU cycles. Furthermore, the memory channel must be disconnected from the internal global I/O bus during a row migration, and hence, it cannot respond to any memory requests originating from co-running applications to other banks [12]. Note that this does not suggest that RowClone is inefficient because of two reasons: (1) it does not require a decoupled sensing and buffering structure, which may increase the cost of DRAM devices, and (2) it is more efficient in copying a row within a single sub-array (FPM) than across different banks (PSM). Yet only PSM can be employed in our memory architecture with asymmetric banks.

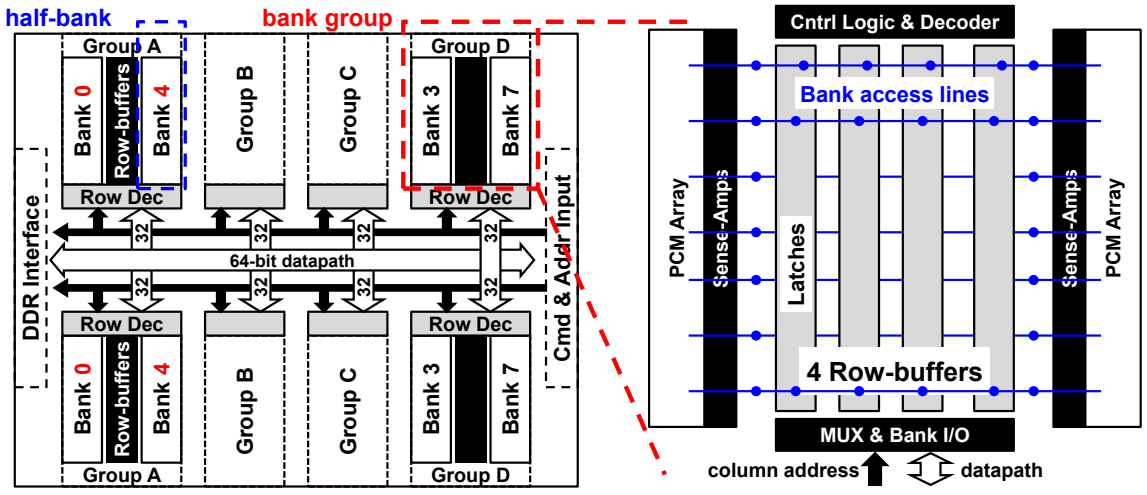
An early DRAM study proposed a cache DRAM architecture containing a 1Mb DRAM and an 8Kb SRAM cache [27]. Theoretically, the SRAM cache can be used as a transfer station similar to our shared row buffers for page migrations. However, from this centralized SRAM cache to one of the DRAM banks, it will be a low-bandwidth path, as can be inferred based on Figure 5. In fact, because of this intermediate transfer station, the migration latency will be twice the latency of RowClone PSM.

In summary, the previous asymmetric memory architecture requires costly page migrations at runtime even with a technique such as RowClone. This in turn diminishes the benefit of asymmetric memory architectures. Consequently, we need a memory architecture that can provide a lightweight page migration mechanism and/or adapt the asymmetry based on the memory access patterns at runtime.

3. Shared Row-buffer Architecture

3.1 Architecture

In this section, we describe our proposed shared row-buffer architecture. Consider that PCM architectures with multiple row buffers have been already proposed in previous studies [2, 4] and implemented in silicon [28]. Thus, we take a PCM architecture with two private row buffers per bank as our



(a) High-level view of a PCM device. **(b) Row-buffer sharing in a bank group.**

Figure 4. An 8-bank PCM device with 4 row buffers shared by two paired banks.

baseline (cf. Figure 1(a)) and focus on the required changes for sharing row buffers between two paired neighboring banks. Note that commodity DRAM does not use global row buffers at the edge of each bank as shown in Figure 1(a), because it uses local, per-mat cross-coupled inverters for both sensing and buffering data. However, PCM decouples sensing and buffering and its sense amplifiers drive explicit latches, giving more flexibilities in row-buffer organization, e.g., multiple row buffers per bank [2]. In fact, this is observed in other types of commercially available non-volatile memories (e.g. flash) employing multiple data/cache registers to hide latency.

Figure 4(a) presents a high-level view of a PCM device with 8 banks (i.e., essentially a physical view of Figure 1(b)). It uses a split-bank architecture that can be found in modern DRAM architectures [6]. A single logical bank is split into two half-banks that are located at the upper and lower sides of a device, respectively, as depicted in Figure 4(a) to reduce the number of inter-bank datalines [6]. On one side of the device, two half-banks, which belong to two neighboring logical banks, are closely placed, and their sense amplifiers and row buffers are positioned between these two half-banks. These two half-banks denote one bank group, and their corresponding logical banks become a pair of neighboring banks sharing these row buffers. The only required change is the need for two row-buffer selection bits (RB [1:0]) to select one of four shared row buffers instead of one bit to select one of two private row buffers. Figure 4(b) depicts the organization of four shared row buffers within a single bank group. The major change is that the sense amplifiers of a bank should be connected to all four row buffers through bank access lines (BALs). This is to some extent equivalent to increasing the number of private row buffers per bank from two to four, except for the further mechanisms to prevent a race condition on BALs.

As shown in Figure 6, the sense amplifier (S/A) gate ensures that the two sets of sense amplifiers of the two paired banks do not drive the BAL at the same time. For example, suppose we activate a row in a bank and store the retrieved page into a row buffer. In such an operation, the S/A gate is turned on for one cycle after the data is stabilized in the

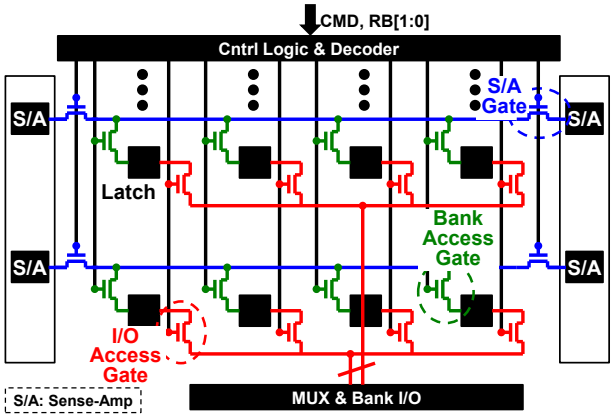


Figure 6. Circuit-level details for row-buffer sharing.

Table 2. Latency analysis for moving data between sense amplifiers of a bank and one of the row buffers.

Latency	Baseline (2 buffers per bank)	Shared row-buffer (4 buffers per bank pair)
decoder gate	486 ps	500 ps
decoder output wire	380 ps	380 ps
equalization	50 ps	50 ps
bank access gate	80 ps	90 ps
bank access line	18 ps	35 ps
latch	80 ps	80 ps
Total	1094 ps	1135 ps

sense amplifier and ready to be driven into the target row buffer. Then we just need to append a few more timing constraints to the memory controller to ensure that the S/A gates of two paired banks are not turned on in the same cycle.

3.2 Latency Overhead

As can be observed from Figure 6, sharing the row buffers between two paired banks can increase the latency for driving data from the sense amplifiers of a bank to a row buffer and vice versa, since more row buffers are connected through BALs. Table 2 compares the latency components of shared row-buffer architecture against those of the baseline architecture. The increased latency components are as follows. First, a 2-to-4 decoder replaces the 1-to-2 decoder in the baseline architecture and results in longer decoder gate latency. Second, the bank access gate latency is increased since a BAL needs to drive four bank access gates instead of two. Finally, the longer BALs to connect more row buffers increase the wire latency of BALs. In summary, the shared row-buffer architecture increases latency of transferring data between sense amplifiers and row buffers only by 41ps compared with the baseline architecture according to our SPICE simulation. The latency increase is very small because the major latency components are in the vertical direction while the shared row-buffer architecture only slightly increases the latency components in the horizontal direction in Figure 6. For conservative evaluation, nonetheless, we add one cycle to tRCD and tRP.

3.3 Timing Protocol Changes

Since two paired banks share a BAL connected to all four row buffers, the sense amplifiers from two paired banks are not supposed to drive the BAL in the same cycle. Note that the cycle here refers to the internal array cycle (5ns or 200MHz for DDR3-1600), while the external command rate is 800MHz. As a result, if the memory controller issues two consecutive *activate* commands to the two paired banks, there can be a race condition on the BAL. Therefore, we introduce a new timing parameter $tRRD_{ACT\_PAIR}$ (= 4 command cycles (4×1.25ns)), as shown in Figure 7. Similarly, there is also a minimum 4-cycle delay between two *write-back* commands to the paired banks ($tRRD_{PRE\_PAIR}$). However, $tRRD_{PRE\_PAIR}$ is completely hidden by the existing timing parameter, tRRDpre (6~10 cycles) as shown in Table 1. Besides, a read operation turns on the S/A gate to drive the re-

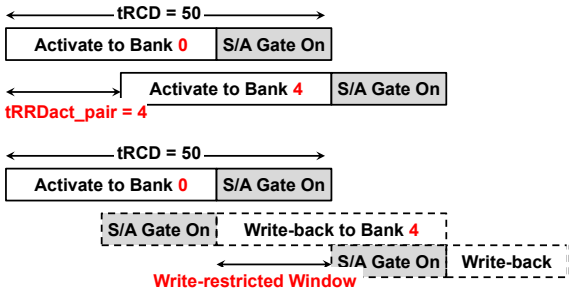


Figure 7. Additional timing constraints.

trieved data into a row buffer at the end of tRCD, while a *write-back* operation switches on the S/A gate at the beginning of tRP to drive the data into the sense amplifiers. Therefore, supposing that an *activate* command is issued to Bank 0 at time T_0, we see a 4-cycle time window (from T_0 + tRCD − 4 to T_0 + tRCD) in which a *write-back* command to Bank 4 should not be issued, as a write-restricted window shown in Figure 7.

3.4 Simultaneous Sharing vs. Dynamic Partitioning

As four row buffers are physically shared between two paired banks, there are two options regarding how to utilize them. First, the memory controller can consider that all four row buffers are logically shared by two paired banks at any given time, denoted by simultaneous sharing. Second, the memory controller assumes the row buffers are logically partitioned between two paired banks, but can be re-partitioned dynamically. Although simultaneous sharing may offer higher performance with more flexibility, it can also significantly complicate the memory controller design. A major complication introduced by logically sharing the row buffers is that the two paired banks are no longer fully independent. For example, in order to issue an *activate* command for a request to Bank 0, we need to evict a page currently stored in a selected row buffer. When the page is dirty and belongs to Bank 4, it requires the memory controller to write the page back to Bank 4 first. That is, a request to Bank 0 incurs a write operation to Bank 4. Consequently, significant changes are required to the per-bank schedulers in memory controllers described in various studies [29, 30, 31].

Hence, for each pair of banks the memory controller needs to maintain a 4-bit status register in which each bit corresponds to one of the four row buffers that are physically shared by two paired banks. Each bit denotes which bank the corresponding row buffer logically belongs to. When the memory controller decides to re-partition the physically shared row buffers at runtime, it can simply clears a row buffer and flips the corresponding bit in the status register.

4. Lightweight Page Migration

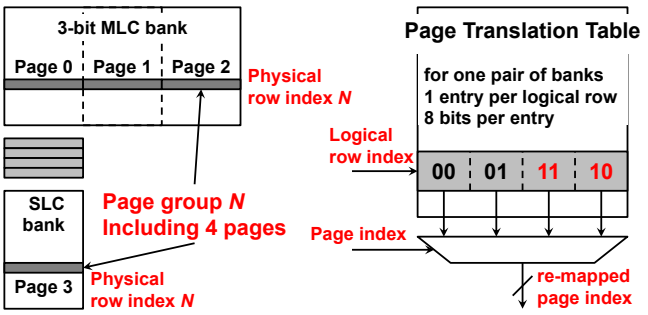
4.1 Page Migration Mechanism

As explained at the end of Section 1 and Figure 2(b), the migration process itself is straightforward with the help of our proposed shared row buffers. The memory controller only needs to (1) issue an *activate* command to bring a migrating page into a row buffer from the source bank; (2) flip the bit in the status register associated with the row buffer indicating that it is assigned to the other bank; (3) update the destination row address tracked by the memory controller for the row buffer; and (4) label the page as a dirty one. At this point, the migration process is logically done, and the memory controller does not need to perform any special action further. When the migrating page gets evicted from the row buffer sometime later, it will be naturally written back to the destination bank. In this section, we will first describe how we manage the asymmetric banks and how the page migration is utilized.

4.2 Data Management in Asymmetric Banks

Regarding how to exploit the asymmetric memory architecture, there are various design approaches: (1) fast regions are used as caches; and (2) fast and slow regions form a unified address space but mapping between fast and slow regions are managed by hardware or software with various hot page selection policies. Since various design options have been already discussed in previous studies (e.g., [1, 20]), we will focus on describing our design with necessary reasoning. Note that this design space exploration is orthogonal to our proposed page migration technique, but we expose the storage capacity of fast SLC banks to the OS and hence SLC and MLC banks together form a unified address space to avoid a major loss of total capacity in this paper.

Hardware-managed mapping scheme: We use a hardware-managed page translation table to track swapped pages and thus make the page migration transparent to the OS. Otherwise, if the page migration is exposed to the OS by modifying the virtual-to-physical frame mapping, TLB updates and cache invalidations are required [9, 22]. To reduce the size of the page re-mapping table, we use a direct mapped scheme. In Figure 8(a), suppose that a pair of 3-bit MLC and SLC banks is sharing four row buffers. A (physical) row of an SLC bank has 1 page slot while that of a 3-bit MLC bank has 3 page slots. Two (physical) rows with the same physical row index from a pair of SLC and MLC banks form one logical row. We consider 4 pages in one logical row as one page group, while restricting the page migration only within one page group. Therefore, we can use 2 bits for each page (i.e., 8 bits for each logical row) to track its re-mapped physical slots within one page group. For example,



(a) Direct mapped scheme. **(b) Translation table.**

Figure 8. SLC bank as a direct mapped fast partition.

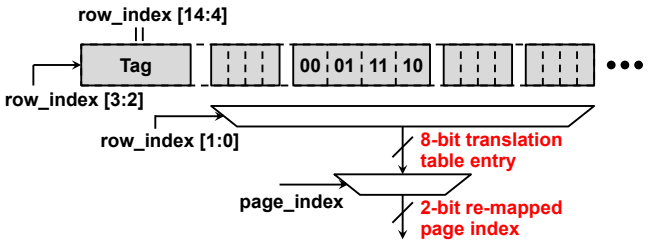


Figure 9. One entry in the 32-way 128-entry (4 page groups per entry) translation buffer.

in Figure 8(b) pages 2 and 3 are swapped in the shaded logical row such that page 2 is now mapped to the page slot in the SLC bank. Overall, we need one page translation table for each pair of SLC-MLC banks; there is one entry for each physical row and 8 bits for each entry. For the memory system we simulate (Table 4), each bank has 32K physical rows, resulting in 32KB additional storage for each pair of banks in the processor. While one can apply our approach for asymmetric memory with 2-bit MLC banks which exhibit better reliability, we use 3-bit MLC banks here to simplify the physical-to-device address mapping for illustration purpose, since each page group will have 3 pages (instead of 4) with 2-bit MLC banks.

Translation buffer: Although we have significantly reduced the size of a page translation table by limiting the freedom of migration with a direct mapped scheme, the 32KB on-chip storage for each pair of banks may be still relatively large. Therefore, we store the full page translation table in the first eight rows of the SLC bank. After the booting process, the OS should be informed that the first eight rows are not available for virtual-to-physical frame mapping. Similar to the TLB used in a processor, we also maintain a translation buffer to cache the recently-used page translation table entries. In addition, we put the page translation information of four consecutive logical rows (page groups) into one entry in the translation buffer since the data field is relatively small (8 bits for one logical row or page group) by default, compared with the tag field (13 most significant bits (MSBs) of 15 row index bits). As shown in Figure 9, the two least significant bits (LSBs) of a row index are used to select the requested page translation entry, and the eleven MSBs of a row index are used for tag comparison. In this way, the number of cached page translation entries is increased by 4 times. Finally a simple FIFO policy is used as our replacement policy of the translation buffer. In summary, a 32-way 128-entry (4 page groups per entry) translation buffer needs 0.7KB storage for each pair of banks. Using 32nm high-performance technology, our estimation based on CACTI [32] exhibits that the translation buffer has only 0.23ns access latency, 5.3pJ energy consumption per access and 6.4mW leakage power consumption.

4.3 Page Swapping Procedure

A memory request to an MLC bank incurs a page swap between the requested page and the page with the same physical row index currently in the SLC bank (victim page). The memory controller labels this request as a triggering request, and creates another pseudo triggering request to the victim page. As a result, the two triggering requests will bring the two to-be-swapped pages into the physically shared row buffers. After an *activate* command is issued for the triggering request, the target row buffer is labeled as one assigned to the other paired bank. The page is labeled as a dirty one and the associated row and page-slot indices tracked by the memory controller are also updated. Because the corresponding entry in the translation buffer is updated immediately after the triggering requests are pushed into the request queue, a strict ordering is enforced for the two triggering requests with respect to their preceding and following requests to the same pair of banks. Note that a pseudo triggering request does not need to actually read/write data through the memory channel; after its *activate* command is issued, it is done and removed from the request queue. Thus it does not consume the memory channel bandwidth.

The performance overheads associated with the page swapping process include an *activate* operation and a *write-back* operation for the victim page, and a *write-back* operation for the requested page. Note that the *activate* operation for the requested page is on-demand since it is invoked by an actual memory request at runtime. The extra read operation for the victim page is in the SLC bank and hence is not likely to be on the critical path considering the slower *activate* operation to the requested page in the MLC bank. Besides, the victim page is not demanded by an actual memory request (a pseudo triggering request instead). Thus, we may use an eager-write-back policy for the victim page considering that it is less likely to be re-used in the near future.

4.4 Adaptive Row-buffer Allocation

Since the SLC banks are used as exclusive caches to the MLC banks, hot pages tend to reside in the SLC banks. Consequently, most incoming memory requests will be serviced by the SLC banks, reducing the effective bank-level parallelism. More importantly, the SLC banks in an SLC-MLC PCM system will exhibit a ~2× higher access rate than the banks in an SLC-only PCM system, incurring more page conflicts and longer memory request latencies. On the other hand, as a bank has more row buffers, it can offer significantly higher performance for most applications [2]. Thus, we can take one row buffer from the MLC bank and assign it to the SLC bank.

5. Experimental Methodology

We use 20 benchmarks from SPEC CPU2006 and 12 GPGPU benchmarks from Rodinia suite [33] (Table 3). The SPEC benchmarks are categorized into three groups (H, M, and L) based on their memory intensities, measured by memory accesses per kilo memory cycles (MPKC). For GPU benchmarks, besides MPKC, we also present the row-buffer locality, measured as a ratio of row-buffer hits to the

Table 3: Workloads.

CPU benchmarks		MPKI
SPEC-LOW (L)	povray, tonto, h264ref, sjeng	0.005 – 0.45
SPEC-MEDIUM (M)	gromacs, astar, gobmk, calculix, hmmer, sphinx3, namd, bzip2, omnetpp, soplex	0.6 – 5.9
SPEC –HIGH (H)	leslie3d, libquantum, bwaves, milc, lbm, mcf	15.4 – 109.8

GPU benchmarks	Abbrev.	MPKC	RBL
Back Propagation	BKP	476	0.99
Breadth-First Search	BFS	223	0.41
CFD Solver	CFD	250	0.86
HeartWall	HTW	149	0.84
HotSpot	HSP	144	0.84
K-Means	Kmeans	463	0.87
LavaMD	LavaMD	477	0.99
LU Decomposition	LUD	133	0.97
Particle Filter	PTF	36	0.84
Path Finder	PFD	488	0.99
SRAD	SRAD	39	0.29
Streamcluster	SC	96	0.53

Multi-program CPU workloads	
LM1	povray – sjeng – gromacs – astar
LM2	tonto – h264ref – gobmk – hmmer
LM3	provray – h264ref – astar – soplex
LH1	povray – tonto – leslie3d – libquantum
LH2	tonto – sjeng – bwaves – milc
LH3	sjeng – h264ref – lbm – mcf
MH1	gobmk – hmmer – leslie3d – bwaves
MH2	bzip2 – soplex – libquantum – milc
MH3	hmmer – namd – bwaves – lbm
HH1	leslie3d – libquantum – bwaves – milc
HH2	libquantum – bwaves – milc – lbm
HH3	bwaves – milc – lbm – mcf

total number of accesses. We create multi-program CPU workloads by mixing benchmarks from various categories. For example, Mix-LM consists of two benchmarks from SPEC-LOW and SPEC-MEDIUM, respectively.

We use a cycle-level integrated gem5+GPGPU-Sim simulator [34, 35, 36]. We implement a cycle-driven memory controller model with the DDR3-1600 protocol and PCM timing parameters (Table 1). We configure the simulator to model the AMD A10-6800K Richland APU [37] (Table 4). While previous studies on PCM proposed hybrid PCM+DRAM memory systems, our proposed techniques are orthogonally applicable to PCM used in PCM+DRAM systems. In this paper, we use a PCM-only system for the main results and also a generic PCM+DRAM system to demonstrate the effectiveness of our proposed techniques.

Note that previous studies on multiple row-buffer design for PCMs employ smaller row buffers, which are narrower than a physical row in a PCM bank [2, 4]. In Section 4.2, our illustration assumed that row buffers have the same width as a physical row in a bank. It was only meant to simplify the explanation, while our evaluation does employ row buffers that are narrower than a PCM row as shown in Table 4.

Table 4: System configurations.

CPU	4 cores, 4.0GHz, 4-wide issue 2-way associative, 32KB private I-L1/D-L1 16-way associative, 4MB shared L2
GPU	12 SMs, 32 SIMD lanes per SM, 800MHz 4-way associative, 16KB L1 per SM 48KB scratchpad (shared) memory per SM 8-way associative, 512KB shared L2
Memory	DDR3-1600 interface, 8GB PCM 32/32-entry read/write queue, FR-FCFS 2 channels, 2 rank/channel, 8 banks/rank 8KB row size, 4KB row buffer size

6. Performance Evaluation

6.1 Effectiveness of Lightweight Page Migration

In this section, we use a 32-way 128-entry (4 page groups per entry) translation buffer for our evaluation. We simulate the following 5 configurations to evaluate our page migration mechanism.

- *SLC-Only*: SLC-only baseline system; each SLC bank is assumed to employ 2× physical rows to provide the same capacity as hybrid SLC-MLC system but have the same timing parameters for performance comparison.
- *MLC-Only*: MLC-only baseline system.
- *LPM (SLC-MLC)*: SLC-MLC hybrid system using the data management policy described in Section 4.2 and our lightweight page migration (LPM) for swapping pages.
- *RowClone (SLC-MLC)*: *LPM (SLC-MLC)* but using the RowClone technique for swapping pages.
- *OracleSelection (SLC-MLC)*: SLC-MLC hybrid system using a perfect runtime hot page selection policy but still using RowClone technique to migrate selected hot pages; it assumes that we can track the access count for each page and sort pages in each page group at end of each epoch to select the hot pages [20].

Figure 10 presents the performance evaluation of CPU benchmarks in the SPEC-MEDIUM and SPEC-HIGH categories. First, the MLC-only memory system delivers 43% lower geo-mean performance than the SLC-only baseline memory system. This indicates a substantial performance loss for using only high-density MLC banks. Second, *LPM (SLC-MLC)* achieves 89% of the geo-mean performance given by *SLC-Only*, while providing 2× higher memory capacity for the same chip area. In contrast, *RowClone (SLC-MLC)* achieves only 75% of the geo-mean performance of *SLC-Only*, since RowClone introduces migration overhead on every page swap. Finally, *OracleSelection (SLC-MLC)* achieves 81% of *SLC-Only*'s geo-mean performance. It is 6% higher than *RowClone (SLC-MLC)* since the extensive profiling effort results in a better hot page selection and

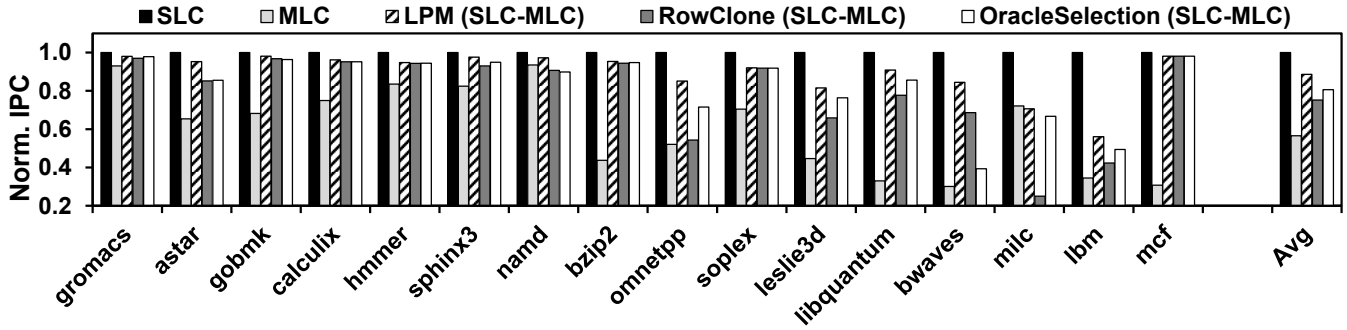


Figure 10. Performance evaluation of a hybrid SLC-MLC PCM system using Lightweight Page Migration (LPM) for swapping pages for CPU benchmarks. IPC values are normalized to those of a PCM system with only SLC banks.

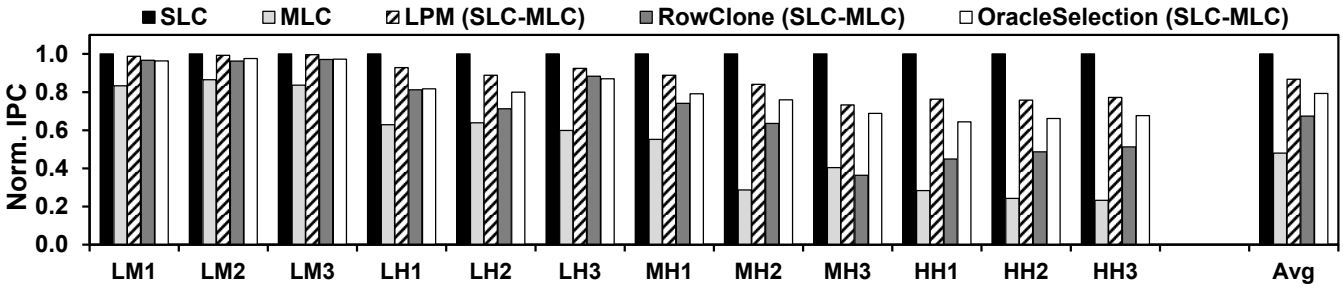


Figure 11. Performance evaluation of a hybrid SLC-MLC PCM system for multi-program CPU workloads.

hence reduces unnecessary page migrations which are expensive using RowClone. However, it is still 8% lower than *LPM (SLC-MLC)*, since any runtime page selection policy requires page migrations in the end to achieve a desired page placement. On the other hand, *OracleSelection (SLC-MLC)* swaps pages only when a memory request hits a profiled hot page in MLC banks, while *LPM (SLC-MLC)* incurs more page swaps, some of which can be unnecessary. The fact that *LPM (SLC-MLC)* outperforms *OracleSelection (SLC-MLC)* suggests that our lightweight page migration mechanism also lowers the requirement for perfect page selection policy, which can be expensive to implement.

For multi-program CPU workloads shown in Figure 11, *LPM (SLC-MLC)* achieves 87% of the geo-mean performance of *SLC-Only*, showing relatively more advantage than *RowClone (SLC-MLC)*, which only achieves 68% of the geo-mean performance of *SLC-Only*. This is because the pipelined serial mode of RowClone disconnects the internal global I/O bus from the memory channel during a bank-to-bank page migration, as shown in Figure 5. Consequently, a page swap invoked by one program will not only hurt the performance of its own but also co-running programs since the internal global I/O bus shared by all memory banks is occupied for 512 cycles (2×256 cycles for swapping). This exacerbates the negative effect of migration latency in multi-program environments. In contrast, a page swap using our proposed LPM technique is not as disruptive as RowClone, as explained earlier.

Furthermore, for GPU benchmarks in Figure 12, LPM exhibits even more advantage than RowClone in supporting page swaps in an SLC-MLC PCM system. *LPM (SLC-MLC)* achieves 77% of the geo-mean performance of *SLC-Only*, compared with only 42% for *RowClone (SLC-MLC)* and 53% for *OracleSelection (SLC-MLC)*. This is because the memory channel bandwidth wasted by swapping pages in RowClone exerts a more significant negative impact on the bandwidth-sensitive GPU benchmarks than CPU benchmarks. Figure 13 plots the page swapping rates, measured as the number of page swaps in 10K memory cycles per channel, when using RowClone as a migration mechanism for

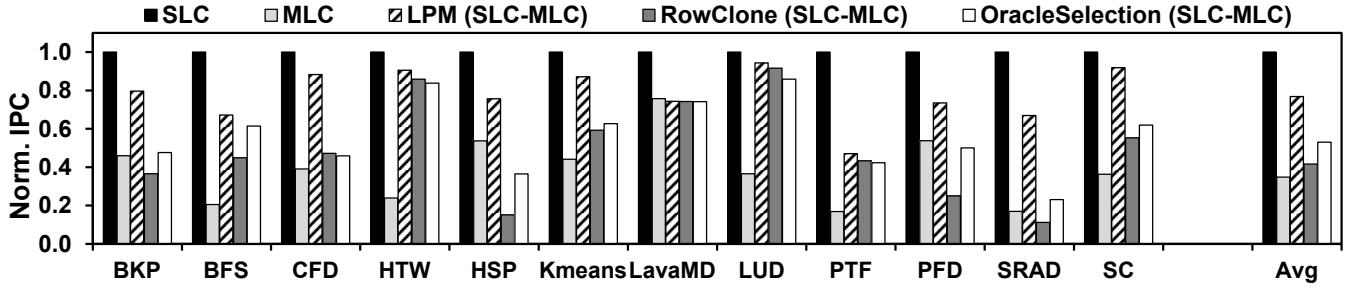


Figure 12. Performance evaluation of a hybrid SLC-MLC PCM system for GPU benchmarks.

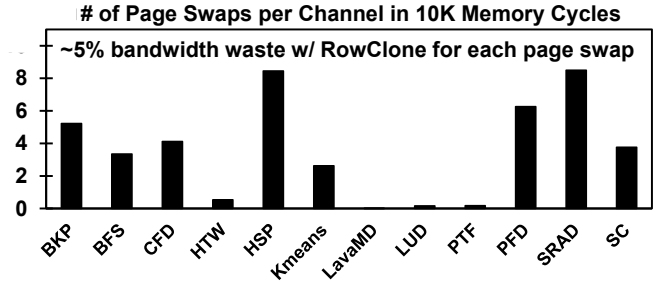


Figure 13. Page swapping rates using RowClone for GPU benchmarks.

running GPU benchmarks. Based on the previous calculation, one page swap in 10K memory cycles incurs ~5% bandwidth loss in this channel. Therefore, benchmarks with a higher page swapping rate will suffer more from the bandwidth waste incurred by RowClone, as we can observe by correlating Figure 12 and Figure 13.

6.2 Adaptive Row-buffer Allocation (RBA)

As explained in Section 4.4, we can take one row buffer from an MLC bank and assign it to the paired SLC bank to compensate the degraded bank-level parallelism and reduce the pressure on the busier SLC bank. We simulate the following 4 configurations

- *LPM (SLC-MLC)*: as described in Section 6.1.
- *LPM + RBA*: adding RBA on top of *LPM* to assign one more row buffer to the SLC bank.
- *LPM (FA) + RBA*: *LPM + RBA* but allowing a page to be migrated to any row instead of within a single logical row (fully associative vs. direct mapped described in Section 6.1).
- *OraclePlacement*: SLC-MLC hybrid system with hot pages (>95% accesses) directly placed in SLC banks from the beginning with perfect future knowledge.

Figure 14 presents the geo-mean performance of CPU, GPU, and multi-program workloads. We also plot the benchmarks that achieve relatively lower performance with only LPM. The IPC values are normalized to *SLC-Only*. First, our adaptive row-buffer allocation technique significantly improves the performance of some benchmarks. *LPM + RBA* provides additional 12% to 49% performance improvement on top of *LPM (SLC-MLC)* for *leslie3d*, *lbm*, *HH1*, *LavaMD*, and *PTF*. As an example, *lbm* is very sensitive to the number of banks, as also observed in [38]. Therefore, the degraded bank-level parallelism with only LPM significantly hurts the performance, while adaptive row-buffer allocation minimizes the performance degradation with increased bank concurrency and row-buffer locality owing to more row buffers for SLC banks. In summary, *LPM + RBA* achieves 87%-93% of the geo-mean performance of *SLC-Only*.

Second, *LPM (FA) + RBA* presents a perfect case that allows pages to migrate freely between a pair of neighboring banks instead of restricting in a single 4-page group, while ignoring the hardware cost for tracking migrations. As shown in Figure 14, *LPM (FA) + RBA* significantly improves the performance compared with *LPM + RBA* for *milc* and *SRAD*. This indicates that we do have cases where restricting migration freedom can hurt the performance. The reason is that for a non-trivial number of page groups, there are more than one hot pages competing for the only one fast page slot in the SLC bank. Nonetheless, we should note that this data management policy is orthogonal to our proposed lightweight page migration technique. Furthermore, LPM actually helps to reduce the negative effect of restricted migration freedom significantly. As shown in Figure 10 and Figure 12, *RowClone (SLC-MLC)* achieves only 25% and 11% of *SLC-Only*'s performance for *milc* and *SRAD*, while *LPM (SLC-MLC)* delivers a much higher performance (i.e., 71% and 67%), since the frequent migrations incurred by multiple hot pages in one page group introduce much lower overhead with LPM than RowClone.

Finally and most interestingly, *LPM (FA) + RBA* even outperforms *OraclePlacement* by 4%, 11% and 4% for CPU, GPU and multi-program workloads, respectively. Compared with *OraclePlacement*, where hot pages are directly placed in SLC banks and hence no migration is needed, *LPM (SLC-MLC)* exhibits only 3%-7% lower geo-mean performance due to our low-overhead migration mechanism. In addition, our adaptive row-buffer allocation technique further improves the performance, outperforming the oracle page placement with the baseline row-buffer architecture.

6.3 Effectiveness in PCM+DRAM Memory Systems

So far we evaluated a PCM-only memory system, but a hybrid PCM+DRAM setup can also be an appealing option to exploit the benefit of both memory technologies. In fact, LPM and RBA can be orthogonally applicable to PCM used in a PCM+DRAM system. However, the main concern for

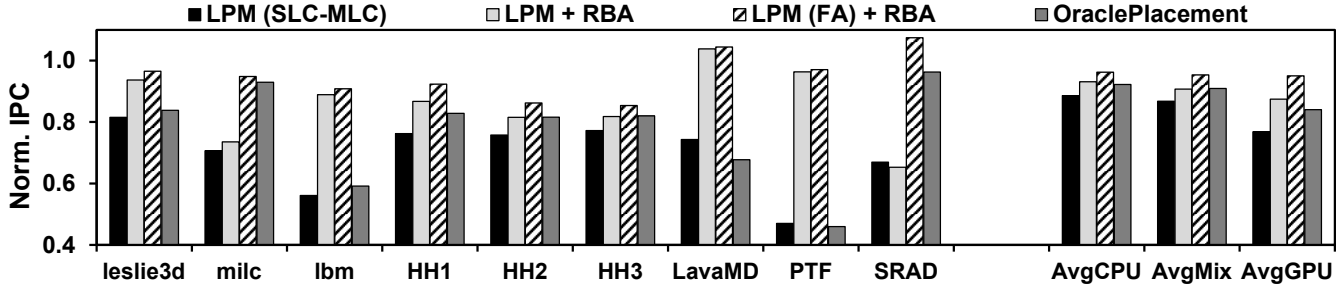


Figure 14. Combining Lightweight Page Migration (LPM) and Adaptive Row-buffer Allocation (RBA).

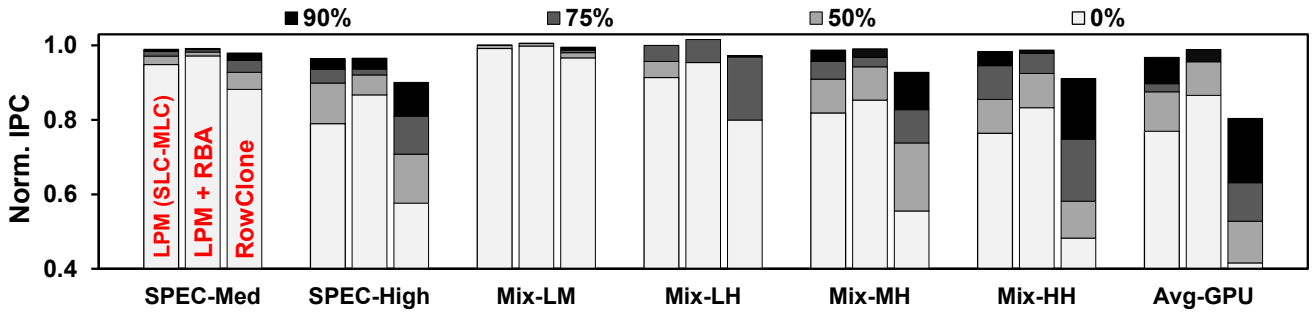


Figure 15. Performance evaluation of LPM and RBA in DRAM+PCM systems when 0%, 50%, 75%, and 90% of memory access are sent to DRAM.

fairly evaluating our proposed techniques in a PCM+DRAM context is that there are many open design options regarding the organization (e.g., DRAM as OS-transparent caches vs. part of unified address space) and the page placement policy (e.g., how to track, rank, and move pages between DRAM and PCM).

Thus, we demonstrate the effectiveness of our proposed techniques in PCM+DRAM memory systems in the following generic way. In each memory channel, we connect DDR3-1600 DRAM and PCM modules. The PCM still have a hybrid SLC-MLC setup with either LPM or RowClone to swap pages. We manipulate the address mapping such that *X*% of total memory accesses are sent to DRAM, where *X* is a configurable parameter in our simulation.

Figure 15 compares *LPM (SLC-MLC)*, *LPM + RBA*, and *RowClone (SLC-MLC)* when X equals 0, 50, 75, and 90. For each X value, IPC values are normalized to *SLC-Only*. When we consider running memory-intensive workloads (i.e., SPEC-high, GPU and Mix-HH) with X = 75, *LPM + RBA* achieves 94%, 93% and 98% of the geo-mean performance of *SLC-Only*, respectively. In contrast, *RowClone (SLC-MLC)* achieves 81%, 63%, and 75% of the geo-mean performance of *SLC-only*. In other words, *LPM + RBA* still gives notably higher performance than *RowClone (SLC-MLC)* for memory-intensive workloads. In general, with more memory accesses serviced by DRAM (i.e., higher X), the performance of PCM makes a less impact on the overall performance (IPC). However, Figure 15 demonstrates that *LPM + RBA* is still effective to improve the performance for memory-intensive workloads even when DRAM is used with PCM. That is, LPM and RBA are useful as long as the performance and capacity of PCM still matters for the overall performance of a computing system.

In summary, LPM and RBA increase the capacity of main memory systems with high-density MLC banks while offering the performance of SLC-MLC memory systems close to SLC-only memory systems, whether DRAM is used together or not. Note, such a benefit is demonstrated even without accounting for the performance benefit of larger capacity.

7. Discussions

The proposed architecture and techniques are based on the PCM's decoupled sensing and buffering structure, leveraging non-destructive read characteristics of resistive memory. However, they are not specific to the PCM technology itself. Any memory technology that employs decoupled sensing and buffering can exploit our key idea of sharing row buffers between banks.

Upon swapping two pages, both rows have to be written back to the memory bank even they are not really dirty, which in turn may shorten the PCM lifetime. However, firstly this concern actually is resulted from the data management policy instead of our LPM technique. Besides, it is reported that MLCs exhibit much shorter lifetime than SLCs [39], while our simulation suggests that the number of writes to MLC banks is reduced by ~85% on average (with few exceptions increasing the number of writes by up to 4.6×) since hot pages are often cached in SLC banks.

8. Other Related Work

In addition to the closely related work including tiered-latency DRAM [1], asymmetric-bank DRAM [6] and RowClone [12] that are already discussed in Section 2, we discuss other related work below.

PCM-based main memory: Lee et al. evaluated the performance and energy consumption of PCM-based main memory systems. A comprehensive analysis on the buffer design space suggests that it can improve the locality and reduce write energy consumption with multiple narrower row buffers [2].

PCM+DRAM hybrid memory system: Qureshi et al. proposed PCM-based main memory coupled with a small DRAM buffer, which acts as a last-level cache [3]. Such a hybrid architecture can exploit the low latency of DRAM and large capacity of PCM. Dhiman et al. proposed another hybrid memory system that DRAM and PCM constitute separate partitions of the address space [23]. This study also presented a data migration policy that once the number of writes to a PCM frame reaches a threshold, the data is migrated from PCM to DRAM partition. Ramos et al. proposed a page ranking policy to manage the data migration between PCM and DRAM. The memory controller monitors the access pattern and migrates the frequently-accessed, write-intensive page frames to DRAM [9]. Even though these pre-

vious work also studied page migration, they have focused on a different aspect compared to this paper. Since the PCM and DRAM are physically separated chips, an expensive page migration through memory channels is inevitable. Therefore, previous studies have focused on the policies to identify the pages that should be migrated to achieve an optimal page placement. In contrast, we focus on the migration process itself and provide an efficient mechanism to migrate pages between asymmetric memory banks.

Page placement in heterogeneous memory systems: Meswani et al. studied memory management policies of a heterogeneous memory system mixing die-stacked and conventional off-package memories. Various techniques regarding dynamic page access tracking and hot page selection are discussed and evaluated. However, as mentioned in Section 2.3, page migrations are required anyway given a runtime hot page selection result. Also the *OracleSelection (SLC-MLC)* case assuming an idealized runtime page selection policy has been evaluated in Section 6.1. From another perspective, Agarwal et al. proposed a smart page placement policy for GPU with high-bandwidth GDDR5 and low-cost DDR4 memories. The page placement policy combines compiler-based profiling information and program-annotated hints and performs close to oracular page placement, mitigating the requirement for expensive page migrations. In contrast, we provide a LPM mechanism and in turn reduce the burden on the software side. Lastly, the *OraclePlacement* case assuming an idealized page placement policy even for multi-program CPU workloads (cf. additional challenges described in Section 2.3) has been evaluated in Section 6.2.

9. Conclusion

In this paper, we propose a novel shared row-buffer architecture for resistive memory. This is motivated by the benefit of asymmetric memory architecture and the challenges associated with page placement and migration, which are needed to exploit the full potential of asymmetric memory architecture. With the physically shared row buffers between two neighboring banks, we propose two effective techniques that can significantly enhance the performance of asymmetric memory architecture. First, our shared row-buffer architecture enables a lightweight page migration (LPM) between neighboring banks. Second, our shared row-buffer architecture enables an adaptive row-buffer allocation (RBA) based on the imbalanced bank access rates at runtime. Hence the performance of fast banks, in which hot pages tends to reside, can be further improved by allocating more row buffers to them. We demonstrate the effectiveness of LPM and RBA using an SLC-MLC hybrid memory system. Compared to an SLC-only memory system, such an SLC-MLC hybrid memory system can increase the capacity with the slow, high-density MLC banks, while approaching the performance of an SLC-only memory system with the help of LPM and RBA.

Acknowledgement

This research was supported in part by NSF (CNS-1217102 and CNS- 1557244) and MSIP (Ministry of Science, ICT and Future Planning), Korea, under the IT Consilience Creative Program (IITP-2015-R0346-15-1008) supervised by NIPA (National IT Industry Promotion Agency). Nam Sung Kim has a financial interest in AMD and Samsung Semiconductor. Nam Sung Kim and Myoungsoo Jung are the co-corresponding authors.

Reference

[1] D. Lee, Y. Kim, V. Seshadri, J. Liu, L. Subramanian and O. Mutlu, "Tiered-latency DRAM: A low latency and low cost DRAM architecture," in *HPCA*, 2013.

[2] B. C. Lee, E. Ipek, O. Mutlu and D. Burger, "Architecting phase change memory as a scalable DRAM alternative," in *ISCA*, 2009.

[3] M. K. Qureshi, V. Srinivasan and J. A. Rivers, "Scalable high performance main memory system using phase-change memory technology," in *ISCA*, 2009.

[4] M. K. Qureshi, S. Gurumurthi and B. Rajendran, "Phase change memory: From devices to systems," *Synthesis Lectures on Computer Architecture,* vol. 6, no. 4, pp. 1-134, 2011.

[5] S. Chung, K.-M. Rho, S.-D. Kim, H.-J. Suh, D.-J. Kim, H.-J. Kim, S.-H. Lee, J.-H. Park, H.-M. Hwang, S.-M. Hwang, J.-Y. Lee, Y.-B. An, J.-U. Yi, Y.-H. Seo, D.-H. Jung, M.-S. Lee, S.-H. Cho, J.-N. Kim, G.-J. Park, G. Jin, A. Driskill-Smith, V. Nikitin, A. Ong, X. Tang, Y. Kim, J.-S. Rho, S.-K. Park, S.-W. Chung, J.-G. Jeong and S.-J. Hong, "Fully integrated 54nm STT-RAM with the smallest bit cell dimension for high density memory application," IEDM, 2010.

[6] Y. H. Son, S. O, Y. Ro, J. W. Lee and J. H. Ahn, "Reducing memory access latency with asymmetric DRAM bank organizations," in *ISCA*, 2013.

[7] M. K. Qureshi, M. M. Franceschini and L. A. Lastras-Montano, "Improving read performance of phase change memories via write cancellation and write pausing," in *HPCA*, 2010.

[8] N. H. Seong, S. Yeo and H.-H. S. Lee, "Tri-level-cell phase change memory: toward an efficient and reliable memory system," in *ISCA*, 2013.

[9] L. Ramos, E. Gorbatov and R. Bianchini, "Page placement in hybrid memory systems," in *ICS*, 2011.

[10] A. Bhattacharjee and M. Martonosi, "Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors," in *ISCA*, 2009.

[11] L. Liu, Z. Cui, M. Xing, Y. Bao, M. Chen and C. Wu, "A software memory partition approach for eliminating bank-level interference in multicore systems," in *PACT*, 2012.

[12] V. Seshadri, Y. Kim, C. Fallin, D. Lee, R. Ausavarungnirun, G. Pekhimenko, Y. Luo, O. Mutlu, P. B. Gibbons, M. A. Kozuch and T. C. Mowry, "RowClone: Fast and energy-efficient in-DRAM bulk data copy and initialization," in *MICRO*, 2013.

[13] M. K. Qureshi, M. M. Franceschini, L. A. Lastras-Montaño and J. K. Karidis, "Morphable memory system: A robust architecture for exploiting multi-level phase change memories," in *ISCA*, 2010.

[14] S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung and C. H. Lam, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development,* vol. 52, no. 4.5, pp. 465-479, 2010.

[15] L. Jiang, B. Zhao, Y. Zhang, J. Yang and B. R. Childers, "Improving write operations in MLC phase change memory," in *HPCA*, 2012.

[16] F. Bedeschi, C. Boffino, E. Bonizzoni, O. Khouri, C. Resta and G. Torelli, "Bit-line biasing technique for phase-change memories," in *ICSES*, 2004.

[17] F. Bedeschi, R. Fackenthal, C. Resta, E. M. Donzè, M. Jagasivamani, E. C. Buda, F. Pellizzer, D. W. Chow, A. Cabrini, G. M. A. Calvi, R. Faravelli, A. Fantini, G. Torelli, D. Mills, R. Gastaldi and G. Casagrande, "A Bipolar-Selected Phase Change Memory Featuring Multi-Level Cell Storage," *JSSC,* vol. 44, 2009.

[18] T. Nirschl, J. B. Philipp, T. D. Happ, G. W. Burrt, B. Rajendrant, M.-H. Lee, A. Schrottt, M. Yang, M. Breitwischt, C.-F. Chen, E. Joseph, M. Lamorey, R. Chee, S.-H. Chen, S. Zaidi, S. Raoux, Y. Chen, Y. Zhu, R. Bergmann, H.-L. Lung and C. Lam, "Write strategies for 2 and 4-bit multi-level phase-change memory," in *IEDM*, 2007.

[19] N. Papandreou, A. Pantazi, A. Sebastian, M. Breitwisch, C. Lam, H. Pozidis and E. Eleftheriou, "Multilevel phase-change memory," in *ICECS*, 2010.

[20] M. R. Meswani, S. Blagodurov, D. Roberts, J. Slice, M. Ignatowski and G. H. Loh, "Heterogeneous memory architectures: A HW/SW approach for mixing die-stacked and off-package memories," in *HPCA*, 2015.

[21] N. Agarwal, D. Nellans, M. Stephenson, M. O'Connor and S. W. Keckler, "Page placement strategies for GPUs within heterogeneous memory systems," in *ASPLOS*, 2015.

[22] M. Awasthi, D. Nellans, K. Sudan, R. Balasubramonian and A. Davis, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *PACT*, 2010.

[23] G. Dhiman, R. Ayoub and T. Rosing, "PDRAM: A hybrid PRAM and DRAM main memory system," in *DAC*, 2009.

[24] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding and O. Mutlu, "Row buffer locality aware caching policies for hybrid memories," in *ICCD*, 2012.

[25] X. Jiang, N. Madan, L. Zhao, M. Upton, R. Iyer, S. Makineni, D. Newell, Y. Solihin and R. Balasubramonian, "CHOP: Adaptive filter-based DRAM caching for CMP server platforms," in *HPCA*, 2010.

[26] D. H. Yoon, J. Chang, N. Muralimanohar and P. Ranganathan, "BOOM: Enabling mobile memory based low-power server DIMMs," in *ISCA*, 2012.

[27] H. Hidaka, Y. Matsuda, M. Asakura and K. Fujishima, "The cache DRAM architecture: A DRAM with an on-chip cache memory," in *IEEE Micro*, 1990.

[28] "Product brief: LPDDR2-PCM and mobile LPDDR2 121-Ball MCP," Micron Technology, Rev. B 12/12 EN, 2012.

[29] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson and J. D. Owens, "Memory access scheduling," in *ISCA*, 2000.

[30] B. Jacob, S. Ng and D. Wang, Memory systems: cache, DRAM, disk, Morgan Kaufmann, 2010.

[31] R. Ausavarungnirun, K. K.-W. Chang, L. Subramanian, G. H. Loh and O. Mutlu, "Staged memory scheduling: Achieving high performance and scalability in heterogeneous systems," in *ISCA*, 2012.

[32] N. Muralimanohar, R. Balasubramonian and N. Jouppi, "Optimizing NUCA Organizations and Wiring Alternatives for Large Caches with CACTI 6.0," in *MICRO*, 2007.

[33] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee and K. Skadron, "Rodinia: A benchmark suite for heterogeneous computing," in *IISWC*, 2009.

[34] 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," *ACM SIGARCH Computer Architecture News,* vol. 39, no. 2, pp. 1-7, 2011.

[35] A. Bakhoda, G. L. Yuan, W. W. L. Fung, H. Wong and T. M. Aamodt, "Analyzing CUDA workloads using a detailed GPU simulator," in *ISPASS*, 2009.

[36] H. Wang, V. Sathish, R. Singh, M. J. Schulte and N. S. Kim, "Workload and power budget partitioning for single-chip heterogeneous processors," in *PACT*, 2012.

[37] "AMD A10-Series A10-6800K - AD680KWOA44HL / AD680KWOHLBOX," [Online]. Available: http://www.cpu-world.com/CPUs/Bulldozer/AMD-A10-Series%20A10-6800K.html. [Accessed Sep. 2014].

[38] M. K. Jeong, D. H. Yoon, D. Sunwoo, M. Sullivan, I. Lee and M. Erez, "Balancing DRAM locality and parallelism in shared memory CMP systems," in *HPCA*, 2012.

[39] X. Dong and Y. Xie, "AdaMS: Adaptive MLC/SLC phase-change memory design for file storage," in *ASP-DAC*, 2011.

[40] W. Mi, X. Feng, J. Xue and Y. Jia, "Software-hardware cooperative DRAM bank partitioning for chip multiprocessors," in *IFIP Int'l Conf. Network and Parallel Computing (NPC)*, 2010.