Throughput Enhancement for Phase Change Memories

Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang, *Member*, *IEEE*

Abstract—Phase Change Memory (PCM) has emerged as a promising candidate for future memories. PCM has high cell density, zero cell leakage, and high stability in deep sub-micron technologies. Although PCM has limited endurance, recent endeavors have shown that its lifetime can be improved by orders of magnitude. However, a major hurdle for PCM is the long write latency and high write power. For this reason, PCM cannot deliver satisfactory memory bandwidth for high-end computing environment such as multi-processing and server systems. In this paper, we develop a non-blocking PCM bank design such that subsequent reads or writes can be carried in parallel with an on-going write. This is effective in removing long blocking time due to serial operations. Moreover, we propose novel memory request scheduling algorithms to exploit intra-bank parallelism brought by our non-blocking hardware. Our non-blocking hardware with scheduling enhancement improves PCM memory throughput by 51% on average. Finally, we propose a fine-grained power budgeting scheme to achieve more throughput improvement under power budgets. Experiments show that our scheduler enhanced with power budgeting scheme can achieve a throughput improvement of 118% on average.

Index Terms—Phase change memory, memory scheduling, low power

1 Introduction

The demand in memory capacity is constantly growing with today's data-intensive applications. The trend in integrating more cores on-chip also poses significant pressure on memory capacity. Unfortunately, the DRAM technology that has been used in main memory for decades is now facing scalability and power challenges in deep sub-micron technologies [1], [2]. Recently, a class of non-volatile memories, such as Flash, Phase-Change-Memory (PCM), Spin-Torque Transfer RAM (STT-RAM), have emerged with zero cell leakage, good resilience to single event upset, and good scalability. Among various technologies, PCM has been studied most as a candidate for future main memory [3]–[14] due to its excellent scalability compared to other alternatives. Also, the zero cell leakage is attractive in main memory design as its leakage power can be dramatically reduced [14]. The drawbacks of PCM technology, however, are in its limited write endurance, long write latency and low memory throughput. Many recent efforts have proposed solutions to mitigate the endurance problem and make the lifetime of PCM competitive to DRAM main memory [3]–[6], [8]–[14]. The long write latency of PCM creates major performance hit because subsequent reads are blocked if the bank is occupied by a write, significantly increasing the read time. Techniques proposed to tackle this problem include write-cancellation, write-pause [15], and using a small DRAM as a cache for a large PCM memory [4], [9], [16].

However, there has not been a technique proposed to improve the memory throughput of a PCM. A state-of-the-art PCM chip can achieve 9 MB/s program throughput [17], while the throughput of DDR2-800 DRAM is 100 MB/s per chip [18]. Such a large discrepancy will be a major barrier to adopting PCM in large-scale computing where high memory throughput is a key requirement. The throughput of PCM is mainly limited by its long write latency (e.g., 1000 ns [19]), large programming logic and high write current per cell (e.g., $0.6 \sim 1$ mA [20]). High write current and large programming logic limit the number of bits that can be written at the same time, forcing a write to split into several iterations which increases the write latency [20], [21]. When a memory bank is serving a write for a long time, no other operations can be performed in this bank, limiting the throughput of the bank. Write-cancellation or write-pause technique [15] can allow subsequent reads to proceed by canceling or pausing the current write, but reads and writes are exclusive to each other. Therefore, they help only in improving the read latency, not memory throughput. The Read-While-Write technique developed recently [17] allows a read and a write to be active simultaneously in two different partitions of a PCM chip. While promoting read-write parallelism, as we will show later, throughput improvement from Read-While-Write is rather limited because PCM memory throughput is mostly bounded by its long write latency.

In this paper, we first present a PCM bank design that allows its left and right half to operate relatively independently so that they can, instead of accommodating one write at a time, serve 2 writes and 2 reads simultaneously. Our goal is to create more request parallelism without dividing a bank into two for the purpose of preserving cell density which is critical in PCM designs. Also, our design is not a dual-port memory design [22] which increases array area quadratically.

- *P. Zhou, B. Zhao, and J. Yang are with the Electrical and Computer Engineering Department, University of Pittsburgh, Pittsburgh, PA 15261. E-mail: {ping.a.zhou, bozhao.us}@gmail.com, juy9@pitt.edu.*
- *Y. Zhang is with the Computer Science Department, University of Pittsburgh, Pittsburgh, PA 15260. E-mail: zhangyt@cs.pitt.edu.*

Manuscript received 01 Jan. 2012; revised 29 Sep. 2012; accepted 13 Mar. 2013. Date of publication 27 Mar. 2013; date of current version 15 July 2014. Recommended for acceptance by L. Wang.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TC.2013.76

Additionally, our design is cell-type independent and is applicable to single-level, multi-level PCM cells, STT-RAM cells or other new memory technologies that suffer from long write latency. Next, we develop novel memory scheduling enhancements to fully exploit the hardware capability and further improve throughput. Our scheduling enhancements reorder requests in each bank queue to exploit intra-bank parallelism. Finally, although each bank can now serve more than one write and PCM writes are of high-power, previous work showed that not all bits need to be written in a write [3], [14]. Hence, introducing parallelism does not put pressure on PCM's power consumption. On the contrary, with our fine-grained power management scheme, we can further improve throughput under the same power budget. Experiments show that our scheduler alone can improve throughput by 51% on average over a baseline blocking PCM design. With our proposed power budgeting scheme, an average of 68% of throughput improvement can be achieved.

2 Background

Phase-change memory (PCM) is one type of non-volatile memory that exploits the unique behavior of chalcogenide alloy such as $Ge_2Sb_2Te_5$ (GST) to store information. A PCM cell usually consists of a thin layer of GST and two electrodes. The chalcogenide has two stable states, crystalline and amorphous. It can switch between them with application of heat via injecting current into the PCM cell.

There are two write operations associated with GST cells. The SET operation heats GST above the crystallization temperature ($\sim 300^{\circ}\mathrm{C}$) but below the melting temperature ($\sim 600^{\circ}\mathrm{C}$) over a period of time. This turns the GST into the low-resistance crystalline state that corresponds to logic '1'; the RESET operation heats GST above the melting temperature and quench quickly. This places the GST in the high resistance amorphous state that corresponds to logic '0'. Due to such heating process, both SET and RESET take relatively long time to complete, and SET is slower than RESET. Recent PCM prototype show that RESET operation can be as short as 60 ns while SET is on the order of several hundreds of nanoseconds [17].

3 Related Work

The concept of parallelizing a read with a long write was first implemented as a read-while-write (RWW) operation in NOR flash [23]. A RWW NOR flash is essentially two or more flash memories on a chip. One is used to store data and the other is for code. When the data memory is written, code can be read out from the other memory. This concept is also adopted in recent PCM prototypes [17], where a PCM chip is divided into smaller partitions, and two partitions can be simultaneously active, one being read and the other being written. However, the above implementations are too coarse-grained, and cannot satisfy the throughput requirement if PCM is used as main memory. In addition, providing RWW alone in the hardware cannot achieve high throughput without modifying the memory scheduling algorithm. As we will show later, our parallelism is provided within each bank of a memory which is much more fine-grained and delivers much higher parallelism, with negligible overhead. Moreover, we develop novel memory scheduling enhancements to fully exploit the hardware capability and achieve much higher throughput.

There have been several approaches to mitigating the long write latency. The first one was to use a small DRAM as a cache for a large PCM [4], [9], [16], which is helpful to reduce average memory latency of a DRAM+PCM. However, this technique does not address the PCM throughput problem. The write-cancellation and write-pause technique [15] help those reads that are blocked due to a busy bank occupied by a write. These techniques either cancel current write or pause the write (for multi-level cells) to serve the read first. The write is re-started later or resumed where it was left off. As we can see, these techniques can only allow one operation in one bank at a time. Writes and reads are still in serial but not in parallel. Hence, they do not help to improve the throughput of the bank.

On the memory bank optimization side, there have been works on changing the DRAM rank architecture for lower power. Those approaches, e.g., mini-rank [24] and rank sebsetting [25], aim to limit the number of banks activated, since modern DRAM fetches more bits than necessary in an access. Hence, only a subset of banks in a rank needs to be activated, saving much power. The goal of this paper is not to lower the power of PCM, as it is a fundamental device limitation that bounds the memory throughput. We aim to increase its throughput under the same power budget given by the device specification. Hence, while previous optimizations on DRAM tend to put more banks into low-power mode, we prefer to activate as many half banks as possible as long as the power stays within device specification.

We also introduce some representative related work on memory scheduling. Memory scheduling has been widely studied for DRAM-based memory controllers. The simplest memory scheduling scheme is first-come-first-serve (FCFS). A simple enhancement to it, first-ready FCFS (FR-FCFS), was proposed to improve row buffer hit rate and the memory system performance [26], [27]. In [28], Mutlu et al. proposed parallelism-aware batching scheme (PAR-BS). PAR-BS organizes incoming requests into batches, and schedules the requests within a batch to exploit bank-level parallelism. The batching mechanism solves starvation problem, while the scheduling within a batch improves bank-level parallelism and the overall throughput. A later work by Kim et al. proposed Thread Cluster Memory Scheduling [29]. TCM categorizes threads into two "clusters" (memory intensive and non-intensive) according to their MPKI. Bandwidth are divided between two clusters, and different policies are employed in them. In the non-intensive cluster, TCM prioritizes low MPKI threads to improve system throughput; while in the memory-intensive cluster, TCM uses rank shuffling to ensure fairness.

Existing schemes all target at bank-level scheduling, i.e. how the requests are dispatched to logic banks. Within each bank queue, requests are issued in the order they are dispatched. Our enhancement are designed as an "add-on" to existing memory schedulers. It further exploits intra-bank parallelism brought by our improved hardware, and hence is orthogonal and additive to those schemes.

In a recent work, Hay et al. proposed a scheme termed "Power Token [30]", which aims to improve PCM throughput via issuing more writes, each writing only the changed bits

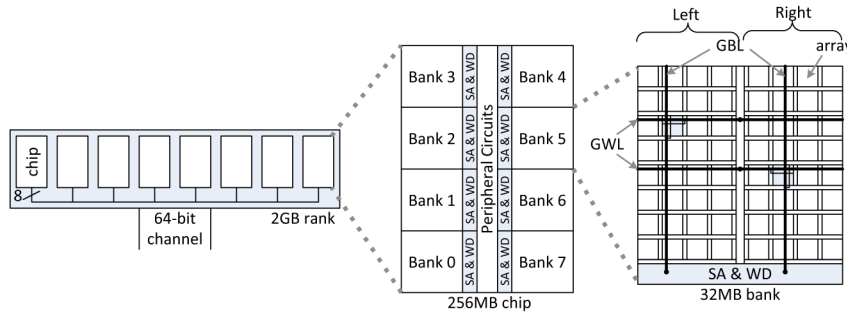


Fig. 1. Organization of one PCM rank. SA—sense amplifier, WD—write drivers, GWL—global wordline, GBL—global bitline. Chip and bank floorplan are from [21].

instead of every bit. The total number of bits written concurrently does not exceed a limit defined by the device. Power Token performs a conservative estimate on the bit changes in the *last level cache*. However, since PCM has the capability of counting bit changes in the hardware [31], the accurate information can be obtained directly in PCM, which is adopted by our power management scheme. In addition, we exploited the fact that a write needs several iterations to finish, and iterations of zero bit changes can be entirely omitted, reducing the write latency which further benefits memory throughput. As we will show in our evaluations, our power management scheme outperforms Power Token by 20% on average in memory throughput.

4 Non-Blocking PCM Bank Design

In this section, we give an overview of our proposed non-blocking PCM bank design. The detailed circuit-level implementation and the overhead quantification are deferred to Section 7.

Our PCM chip and physical bank organization follow a prototype from Samsung [21]. The chips are organized into a memory system similar to a typical DDR2-800 DIMM [32]. Fig. 1 shows our design of a 2 GB PCM memory rank for a standard 64-bit channel. The rank consists of 8 256 MB PCM chips, each having 8 32 MB banks. In each channel transition, all PCM chips work together to deliver 64 bits of data, with each chip producing 8 bits which are generated from one bank within that chip. Hence, a read request of 64B data will require 4 channel transitions (DDR2 interface).

Inside each PCM bank, the 32 MB capacity is divided into 64 4Mb cell arrays (Fig. 1). These arrays are partitioned into left and right half of the bank that can work concurrently. The row decoder, shared by both halves, is in the middle of the bank to avoid long wordline driving. The read and write circuits, SA&WD, are at the bottom of the bank. Unlike DRAM in which each small array comes with its own read/write circuits, a PCM bank typically has many arrays share one set of SA&WD to increase the overall cell density, as seen in many industry prototypes [17], [20], [21], [33]. This is because the SA&WD of PCM are much larger than of DRAM as PCM's write current is much higher. For instance, only 64 sets of SA&WD can fit into the width of 4096 column of cells [21], whereas in DRAM every two columns of cells can have one set of SA&WD. In the Samsung prototype [21], there are 64 sets of SA&WD below each half bank.

When a write is performed, the row decoder selects, drives and holds one wordline high to open the cells in that row. Proper bitlines corresponding to the address are then selected to start writing. One write always activates only one array in the bank. To make a write non-blocking, we enable another operation whose activity is in a different array of the bank. This is fundamentally feasible because when a write is in-progress, only the write circuit is occupied but the read circuit is idle. With proper circuit changes, the read circuit can be used to service a different read in concurrent with the write.

However, the first challenge here is that the wordline of the write is held high by the row decoder for the duration of the entire write. The decoder needs to be freed in order to decode a second address and drive a second wordline. The second challenge is that the read wordline will interfere with write because it opens cells at its cross points with the write's bitlines, and these cells would be mistakenly writen. Likewise, the write's wordline will also interfere with the read's bitlines. This is depicted as the two crosses in Fig. 2, which will destroy both the write and the read.

The first challenge can be addressed by latching the wordline of the write after it has been driven high and then releasing the decoder. Hence, the decoder can continue to service subsequent operations. The second challenge can be addressed by leveraging existing hierarchical wordline and bitline organization in many industry prototypes [20], [21], [33], i.e. one global wordline/bitline (GWL/GBL) with local wordlines/bitlines (LWL/LBL) in each array. The idea is to offload an access from GWL to LWL and release GWL for the new access. However, the GBL cannot be released as it connects with the read/write circuit. Such connection must be maintained throughout an access. Hence, two operations that fall within the same column of arrays cannot be performed concurrently because they need to share the same GBL. These arrays will be referred to as "conflicting" arrays in

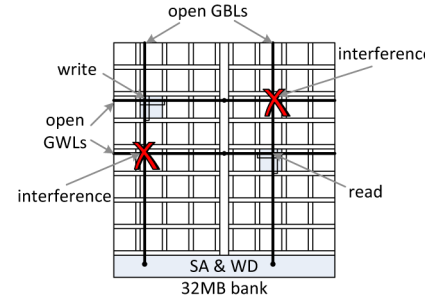


Fig. 2. Interference among wordlines and bitlines for write and read in different arrays.

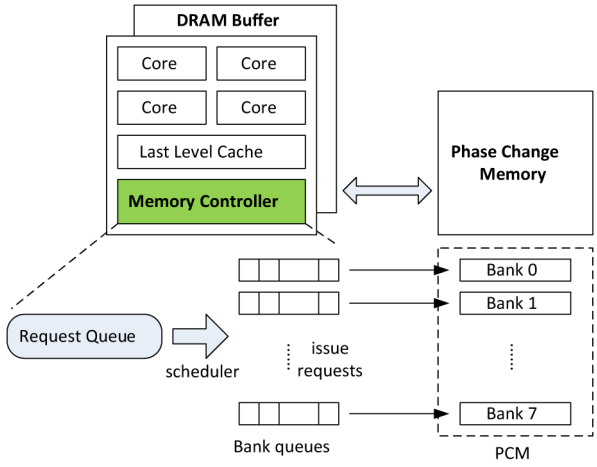


Fig. 3. Architecture overview of experimental platform.

this paper. We revised existing hierarchical wordline/bitline designs to support our non-blocking bank accesses. The details can be found in Section 7.

We remark that our design is *not simply breaking a bank into two (or throwing in more banks) for more parallelism*. Those designs would dramatically reduce the PCM capacity density because each bank must be equipped with a new set of peripheral circuit including decoders, drivers, I/O buffers, and address/data/power routings. Our design is based on the Samsung prototype [21] except that we utilize its idle SA& WD for parallelize-able requests, and time-multiplex the shared row decoder to enable parallelism. For example, when the left half is serving a write, the WDs below the right half and all SAs are idle, but the center row decoder is busy. We let free the decoder so that it can serve another read or write and utilize the idle SA& WDs. As we will show in Section 7.2 that our design adds only 5% of hardware overhead, which is much more lightweight than dividing a bank into two, or using more banks to achieve the same parallelism. This is the key advantage of our bank design.

To summarize, each half bank in our design can serve 1 read and 1 write at the same time. Hence a bank can serve up to 2 reads and 2 writes concurrently. There is also hardware limitation on this concurrency: two requests that access arrays on the same column cannot be active at the same time.

5 Exploiting Intra-Bank Parallelism via Memory Scheduling

In this section, we propose our memory scheduling enhancements to exploit the opportunities of intra-bank parallelism provided by our non-blocking bank design.

5.1 Baseline Architecture

We use a typical 4-core CMP architecture with private L1 and shared L2. A 128 MB DRAM buffer resides between the processor and PCM main memory to mitigate PCM's long latency and limited write endurance, as shown in Fig. 3.

The row buffer organization of PCM bank is also different than DRAM bank. We adopt a similar architecture as proposed in [6], in which sensing and buffering are separate. This allows greater flexibility in row buffer organization by enabling multiple buffer entries [6]. And as shown in [6], it is more advantageous for PCM to use multiple narrow buffer entries in terms of energy and delay. In this study, we adopt a row buffer with 8 256B buffer entries. Since a read request is 64B wide, filling a 256B buffer entry involves multiple reads, which increases bank occupation time. However, read latency is not affected as critical words are sent back first. This is modeled in our experiments. Such a multi-entry buffer design is also natural to concurrent intra-bank reads and writes.

Our multi-entry buffer uses a write-through policy, i.e., writing into a logic bank and the corresponding buffer entry occurs simultaneously. This is because if we use a write-back policy, a read operation may be held for long time due to slow write back, harming the average read latency. Although write-back policy may reduce the total number of PCM writes, we found this advantage to be diminished: Each write request is 64B wide, and each buffer entry is 256B. Write-back can save total number of PCM writes if a same 64B line of a 256B entry is written more than once before the entry is evicted, otherwise the total number of PCM writes stay the same. However in our experiments, no write hits on a same 64B line were observed.

Memory requests arriving at the memory controller are first buffered in the input queue, waiting to be dispatched into memory banks. The controller then schedules these requests to banks according to certain policy, e.g., to maximize bank-level parallelism. We adopt one such scheduler PAR-BS [28] in our baseline. Once a request is dispatched to a bank, it is moved from the input queue to the corresponding bank queue. Requests in each bank queue are then issued in order. Our baseline design uses conventional blocking circuit, meaning that each bank can serve one request at a time.

5.2 Motivation

Although our proposed hardware design can provide more concurrency inside each PCM bank, the final throughput also depends on whether the memory requests issued to the bank can fully take advantage of such *intra-bank parallelism*. Existing memory scheduling schemes aim to exploit *inter-bank parallelism* and are not aware of this new opportunity. Once dispatched, requests in one bank queue are issued in order. This could lose significant opportunities to further improve throughput: if the head request of the bank queue conflicts with an on-going request, it cannot be issued until the on-going request finishes. All subsequent requests are blocked even though some of them do not have such conflict. This is even worse with PCM's extreme latency asymmetry between reads and writes (50 ns vs. 1000 ns in Numonyx 45 nm 4Gbit prototype [19]). We now use an example shown in Fig. 4 to illustrate this problem.

Consider a bank queue containing request sequence $\{W1, R2, R3, R4, R5, W6, R7, R8\}$. Among these requests, $\{W1, R2, R4, R5, R8\}$ access the left half, and $\{W6, R3, R7\}$ access the right half of the bank, as shown in Fig. 4(a). W1 conflicts with R5 and R3 conflicts with W6. We assume 1000 ns and 50 ns for write and read respectively [19].

- **Fig. 4(a)**. In baseline architecture, each bank can only serve one request at a time. Requests are issued in order. Total time to finish the sequence is $\sim$2300 ns.
- **Fig. 4(b)**. We use non-blocking bank without any scheduling enhancement. Requests in the bank queue are still issued in order. Since R5 conflicts with W1, it cannot be issued until W1 finishes. This immediately delays all

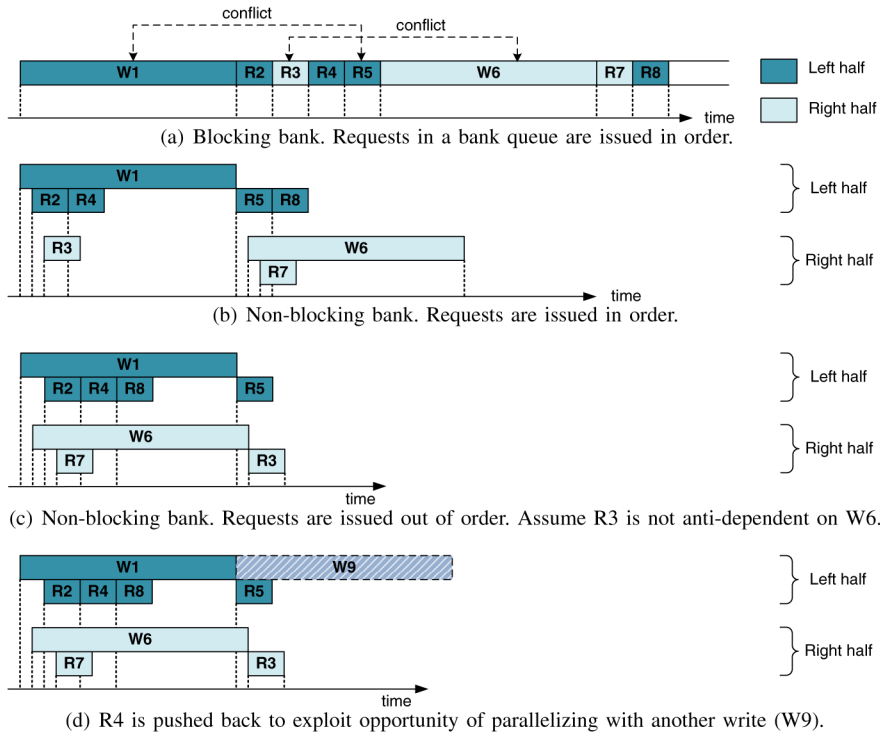


Fig. 4. The impact of scheduling on requests finish time. Assume W1 conflicts with R5 and R3 conflicts with W6.

subsequent requests. However, W6 does not conflict with W1. So W6 could have been issued in parallel with W1. Also, it does no harm to issue R8 sooner (than R5) as it does not conflict with W1. Nevertheless, the total completion time is approximately 1000 ns (W1) + 1000 ns (W6) = 2000 ns, a 13% improvement over baseline.

- **Fig. 4(c)**. We reorder the requests in the bank queue to exploit intra-bank parallelism, assuming dependencies among them have been resolved earlier. In this sequence, W1 and W6 are parallelized. All read requests except R3 and R5 are parallelized with writes. The total time spent by this sequence is approximately 1000 ns (W1 and W6) + 50 ns (R3 and R5) = 1050 ns. Comparing to baseline, the completion time is reduced by more than 54%.

A key point shown in this example is that in our non-blocking bank design, *reordering requests* is critical to the overall throughput and the average read latency. Also, due to the significant gap between read and write latencies, it is utmost important to *overlap* writes as much as possible to shorten the total latency of the entire sequence. This often requires to move writes ahead of many reads. But such move will not hurt the reads too much because they can be parallelized with writes most of the time.

If a read conflicts with a write, we still prefer to issue the write first because the read may have a chance to overlap with a subsequent write. For example, in Fig. 4(d), R5 follows W1 since R5 may be able to overlap with another write W9. Scheduling R5 before W1 would lose such parallelism unless W9 is also moved ahead of W1 to run in concurrent with R5. Either case shows a write-precedence scheduling generates more parallelism. This is also confirmed by our experimental results in Fig. 11: comparing the scheme of putting read requests first and our proposed write-precedence scheme, the latter achieves 15% more throughput improvement on average.

Based on these observations, we develop several enhancements of *intra-bank reordering* that favor writes over reads to exploit intra-bank parallelism. Those are non-conventional as most existing DRAM-based policies prioritizes reads over writes. Our algorithm is thus designed for the unique properties of PCM operations.

Since our enhancements target requests inside bank queues to exploit intra-bank parallelism, they can be easily integrated with existing schedulers that target requests inside the input queue (exploiting inter-bank parallelism). We use PAR-BS [28] as the memory scheduler, followed by our proposed scheduling enhancements as described below.

5.3 Aggressive Write-Precedence Reordering

Algorithm 1 The AWP algorithm.

foreach *free slot (start with free write slots if available)* **do**

Look for a request that does not conflict with any on-going or selected requests;

if *request found* **then**

Mark the request as selected;

end

end

if *one or more requests selected* **then**

Move the selected requests to the front, in the order as they are selected;

end

The initial algorithm is directly derived from the intuition developed above. We term this algorithm aggressive write-precedence (AWP) reordering since we aggressively reorder requests in bank queues. The basic idea is to move requests that can be parallelized with on-going requests to the front of a bank queue, so that they can be issued right away.

With non-blocking bank design, one bank can serve up to 4 requests at a time (2 writes and 2 reads), which are termed *slots* of the bank in the following discussion. AWP simply tries to fill free slots with suitable (non-conflicting) requests. The algorithm can be described as Algorithm 1. When there are both free write slot(s) and free read slot(s), AWP tries to fill write slot(s) first. This means that writes are prioritized over reads, hence the term "write-precedence".

5.3.1 Integration with PAR-BS

Since our scheduling algorithm gives priority to writes over reads, there is a chance for a read to be pushed back infinitely. The PAR-BS [28] we use already avoids starvation through request batching: all requests in the previous batch must be issued to the bank before a new batch starts. AWP follows this by reordering only among those requests that are in one batch. Therefore, the starvation avoidance mechanism of PAR-BS is naturally inherited by AWP (as well as our other enhancements).

The original PAR-BS marks up to N requests per thread per bank when creating a new batch. This implies that there could be imbalanced number of requests for each half bank in a batch, which could harm the parallelism between the two halves. We revise the design to mark up to N/2 requests per thread per *half* bank to balance the requests dispatched to the two halves. We term this simple revision $\texttt{PAR}-\texttt{BS/Half}$. As we will see in Section 7.4.1, this optimization improves throughput over the original PAR-BS by about 20%.

5.3.2 Problems with AWP

With aggressive request reordering in the bank queue, AWP introduces a severe side effect: reduced row buffer hit rate. When requests are dispatched by the memory scheduler (PAR-BS in our case), they are typically ordered to achieve high row buffer hit rate, in addition to improving memory throughput. AWP destroys such locality as it reorders non-conflicting request without considering row buffer hit. A request reordered by AWP may generate a row miss and also evict a buffer entry that could have been a hit by other requests. Our experiments show that the read row buffer hit rate is degraded by more than half using AWP (as shown in Section 7.4.2). Although throughput is important for main memory, the row buffer hit rate is critical to read access latency, and hence the performance of a workload. Next, we will develop an improved AWP that does not harm the row buffer hit rate that much, but still achieves high memory throughput.

5.4 Row-Hit Aware Write-Precedence (RAWP) Reordering

To overcome the problem of AWP, we develop an improved reordering scheme that maintains row buffer hits while achieving high throughput—Row-Hit Aware Write-Precedence Reordering (RAWP). To achieve both goals, RAWP follows two principles when reordering requests in a bank queue:

1. Reorder write requests in a similar way as in AWP to achieve high throughput.
2. Issue read requests in similar way as in PAR-BS to maintain row buffer hits.

5.4.1 Read Insertion

The first challenge of RAWP is that when issuing read requests in a similar order as in PAR-BS, some of them may conflict with on-going writes and be blocked for a long time. To address this challenge, we develop a technique called *Read Insertion* to resolve the conflict and preserve the row buffer hit.

Due to PCM cell's high write current, a write access is typically completed in several rounds of partial writes [20], [21] (e.g., a 512-bit write can be divided into 8 rounds, each round writing 64 bits). Read Insertion simply allows a read to be "inserted" between two rounds of an on-going write. The on-going write is paused and necessary information is saved so that it can resume properly. The hardware requirement is analogous to the "write pausing" technique [15] which pauses a write for a multi-level cell (has a different write mechanism from a single-level cell) and resume it later.

With Read Insertion, a read request can be inserted in the middle of an on-going write even if they conflict. This provides a key advantage in that we can issue read requests in a similar order as in PAR-BS to achieve similar row buffer hit rate. Note that Read Insertion may impact the memory throughput as some parallelizable reads give way to conflicting reads for row buffer hits. However, as we will show in Section 7.4, such an impact is quite limited.

5.4.2 The RAWP Algorithm

With read insertion, RAWP uses a two-step approach to reorder the requests in a bank queue: 1) select issue candidates; 2) form issue group from the candidates in 1). **Step 1. Selecting issue candidates.** In this step, RAWP picks candidates for each free "slot" of the bank. It first ranks all requests. The request with the highest rank will be selected first as an issue candidate. Like in AWP which tries to fill write slot(s) first, RAWP starts ranking with free write slot(s).

For a write slot, the ranking criteria with decreasing weight are: 1) batched; 2) row hit; 3) thread load (first 3 criteria are determined by PAR-BS); 4) number of reads this write conflicts with, the fewer the better.

For a read slot, the criteria are: 1) batched; 2) row hit; 3) not conflicting with on-going writes or any write candidates we have already marked; 4) thread load (criteria 1),2),4) are determined by PAR-BS).

For example, a read satisfying first 3 criteria will have a higher rank than one satisfying first 2 criteria. With Read Insertion technique, we consider reads that conflict with a previous write but could be inserted into a write as non-conflicting.

Once candidates are selected for all free slots, they form an "issue group" that will be moved to the head of bank queue. Our next step is to determine the relative order among these candidates before moving them to the head of bank queue.

Step 2. Forming issue group from issue candidates. In this step, RAWP forms an issue group from the candidates picked

in the previous step and move them to the head of bank queue. Depending on their types and row buffer hit status, the candidates can be classified into four categories: 1) write row hit; 2) write row miss; 3) read row hit and 4) read row miss. RAWP determines the relative order of these candidates according to their categories. In implementation, this can be done by treating the issue group as an "issue queue" and appending candidates to the queue incrementally (starting from an empty queue), as described in Algorithm 2.

Algorithm 2 RAWP forming issue group

Start from an empty issue group (queue);

foreach *Write row hit candidate(s)* **do**

Append the candidate to the end of issue group (queue);

end

foreach *Read row hit candidate(s)* **do**

Append the candidate to the end of issue group (queue);

end

foreach *Write row miss candidate(s)* **do**

Append the candidate to the end of issue group (queue);

end

Move all candidates to head of bank queue, in the order as they are in the issue group (queue);

RAWP places write hits in the first place since they are beneficial to both throughput and row buffer hit rate. Between read hits and write misses, we prefer read hits because reads are very fast, and they may be parallelized with previous writes. So a subsequent write (miss) does not need to wait long. In addition, if the write miss goes first, then it may evict a row buffer which may cause the original read hit to become a miss. Finally, if a read miss is selected as a candidate, it is not included in the issue group because it does not help throughput, nor row hit rate. Therefore, read misses will be left in the batch till the end, and they will be executed in the order determined by PAR-BS. Once an issue group is formed, all requests must be issued before forming a new issue group.

An example. We now use an example to finish the discussion of RAWP. Suppose there is a sequence of requests $\hat{W}_1^*$, R_2, $\hat{R}_3^*$, $\hat{R}_4$, W_5, R_6^*, $\hat{R}_7$, R_8 ($\wedge$ means a request to left half, * indicates a row buffer hit currently). Assuming that all slots are available, RAWP picks one candidate for each slot: $\{\hat{W}_1^*, \hat{R}_3^*\}$ and $\{W_5, R_6^*\}$ for left and right half respectively. Next, RAWP forms an issue group from these requests following Algorithm 2: $\hat{W}_1^*, \hat{R}_3^*, R_6^*, W_5$. This is because $\hat{W}_1^*$, $\hat{R}_3^*$ and R_6^* are all row hits, and W_5 is a miss. As we can see, all row hit requests are preserved in this issue group, illustrating the effectiveness of RAWP. Moreover, after this issue group, the row hit/miss status will change since the write miss will update the row buffer. Hence, there might be row hits for the remaining reads because of this change. This is the reason why sometimes RAWP can result in a little higher row buffer hit rate than the original PAR-BS.

6 Throughput Improvement under Power Budgets

The increased parallelism achieved by non-blocking PCM bank design and our memory scheduling enhancements leads to a larger number of concurrent PCM write requests. While this benefits memory throughput, it does, however, introduce concerns of higher power consumption in PCM. The high write current of PCM cells can cause significant voltage bouncing to the charge pumps of write drivers, and can be a burden for the pumping capacity during the operation [34]. Hence, it is challenging to design the charge pumps that can supply high current while sustaining high voltage at the same time [21]. For those reasons, the number of bits that can be written concurrently is limited, which is termed the *power budget* of the PCM.

Unfortunately, to increase the write concurrency, simply raising the power budget of PCM memory is not a cost-effective design. On the other hand, keeping the original power budget as in the baseline bank design may require limiting the number of simultaneous PCM write requests, which contradicts to our effort in throughput improvement. Hence, a technique is needed to preserve the throughput improvement from our scheduling enhancements without having to raise the power budget.

In order to improve the throughput of PCM memory without raising its power budget, an intuitive approach is to improve the *utilization* of the power budget. This can be accomplished by reducing the number of bit writes in each write request (i.e., the "power demand" of each write request). There have been schemes proposed to serve this purpose [3], [14]. The Differential-Write (DW) [14] scheme writes only those bits that are changed in each write. This is achieved by comparing existing data with new data through XNOR gates and throttling the unnecessary bit writes using comparison results. This simple enhancement has low overhead as the XNOR gates are much smaller than the write drivers. Flip-N-Write (FnW) [3] shares similar idea with DW except that it writes either the original value or its inversion, whichever results in less bit flips. Moreover, DW provides fine-grained (bit-level) information about a write request's power demand, which can be used to facilitate our power budgeting decisions (as we will see in following sections).

In this sections, we propose a fine-grained power budgeting technique, Bit Level Power Budgeting (BPB), which leverages the information provided by DW and achieves better throughput under the same power budget.

6.1 Overview of Bit-Level Power Budgeting

The basic idea of BPB is to leverage the information provided by DW and get better estimation of a write request's *power demand*. This would facilitate the decisions on how a write should be issued. BPB is performed after requests have been scheduled by our proposed algorithms. It is responsible for deciding whether a write request can be issued to a bank with currently available power budget. A write request thus may stall to wait for enough power budget before it can go to a bank. When making a decision, BPB selects a proper *write configuration* of a write to see if it can be issued. As we discussed earlier, a write data block is often divided into equal chunks that are written in serial, requiring several

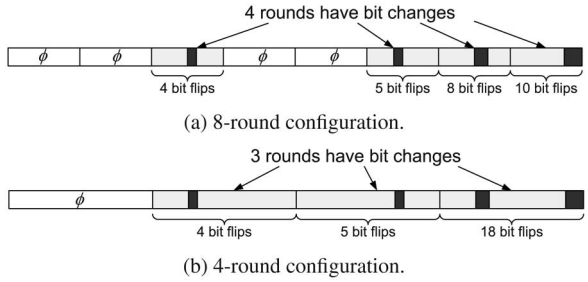


Fig. 5. Comparing different write configurations. Boxes containing ϕ are redundant rounds that can be skipped. Dark parts are bit changes. Those rounds must be performed.

rounds to complete. A write configuration determines how a 512-bit write request is divided and how many rounds it takes to finish. For example, an 8-round configuration will finish in 8 rounds, with each round writing 64 bits. Or, the write may use a 4-round configuration which finishes in 4 rounds, with each round writing 128 bits. In this work, we assume 4 possible write configurations: 8-, 4-, 2- and 1- round. We design our scheme so that the information can be processed locally and does not interfere with memory scheduling. This allows the power budgeting controller to reside on the PCM DIMM, avoiding the transmission between the PCM DIMM and the memory controller. It also allows our scheme to work with other memory schedulers without complex changes.

6.2 Trade-Offs in Flexible Write Configuration

With Differential Write (DW), number of bit changes in a write request is not fixed. Depending on the distribution of bit changes in the line, number of of bit changes is also different in each round of the write request. We assume that the power demand of a write configuration is determined by the round with most number of bit changes. As shown in the example in Fig. 5, 4-round and 8-round configuration has power demand of 10 bit writes and 18 bit writes respectively.

As we can see, low-round configurations (4-round in the example) tend to have higher power demand than high-round configurations (8-round in the example). In fact, this is a trade-off between write latency and power demand: small round number means low write latency, but requires higher power budget, and large round number means just the opposite. On the other hand, choosing a small round number may delay the issue of a write since more power budget is required, and a high round number may let the write start earlier since less power budget is required. Essentially, we look at the *finish time* of a write, which is the sum of the request's projected start time (T_{start}) and latency ($T_{latency}$).

One advantage of high round number is that some of the rounds may be entirely skipped if no bit changes are present, to reduce the write latency. Previous work has shown that on average, 85% of bit writes are redundant in memory [14]. Such a high percentage increases the likelihood of such redundant rounds when each round is relatively narrow. BPB allows skipping of these redundant rounds to reduce actual number of rounds and latency. This is also illustrated in Fig. 5: the actual number of rounds performed in 8-round and 4-round configurations are 4 and 3 respectively. With this technique, a write configuration determines the upper bound of its number of rounds instead of a fixed one (e.g., 8-round configuration means a write request will finish in *up to* 8 rounds). Also can be seen from the example is that the 4-round configuration is only 1 round less than 8-round configuration, but the former has much higher power demand. When evaluating the trade-off between less power demand and shorter latency, BPB must also take this new variable into consideration.

6.3 The BPB Algorithm

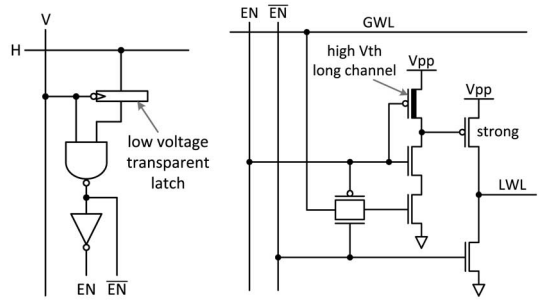
BPB selects a proper write configuration for each write by looking at its *finish time*, which can be expressed as $T_{finish} = T_{start} + T_{latency}$. T_{start} is the time in the future when there is enough power budget for this write configuration. In reality, this number is affected by many other factors in addition to power budget, such as channel busy status, command bus availability, etc. $T_{latency}$ is the time required to serve the write request. This is mainly determined by how many rounds are actually performed. With Read Insertion, this value is also somewhat nondeterministic. Hence, instead of trying to calculate precise T_{finish} accurately, we use its lower bound—the "earliest possible finish time"—as a simple approximation.

For a particular write configuration, $\hat{T}_{start}$ is defined as its "earliest possible start time", which is the earliest time in the future when there will be enough power budget, assuming no more write requests are activated from now till then. Under this assumption, available power budget will keep increasing as on-going write requests finish and leave the system, so that we can always determine the earliest start time. $\hat{T}_{latency}$ is the "shortest possible" latency required, assuming no read insertion. For a given write configuration, this can be determined by its number of non-redundant rounds. BPB then iterates on each possible write configurations to pick the one which generates minimum $\hat{T}_{finish}$.

The "greedy" nature of this procedure tends to grab as much power budget as it can to reduce latency. This could sometimes lead to a situation that a few write requests with high power demand use up most of the available power budgets and cause other issued requests to be held. This is undesirable as there might be low-demand requests that could be otherwise served in parallel. To avoid this side effect, we add a simple constraint $DemandCap$ to the iteration to limit power demand of the picked write configuration. Based on our experiments, we set $DemandCap$ to 64 in our evaluation.

6.4 Overhead of BPB

Finally, $\hat{T}_{start}$ and $\hat{T}_{latency}$ require the knowledge of number of changing bits and their distribution in the write request. In FnW and DW, such information is obtained after the pre-write read and a comparison in the bank. Due to BPB's decoupled design, it can process this information locally instead of having them transmitted between the memory controller and DIMM. When a write request is issued, BPB uses its bit change information to select a write configuration for it. Since the write request often needs to wait for enough power budget, the time to compute its write configuration may overlap with the wait time. In our experiments, however, we charge a 100 ns penalty for each write request. Our experiment results show that this overhead is worthwhile for improving throughput under power budgets.



(a) Generating array selection signal 'EN'. (b) LWL driver.

Fig. 6. Holding local wordline (LWL) high while releasing the global wordline (GWL) and the H line.

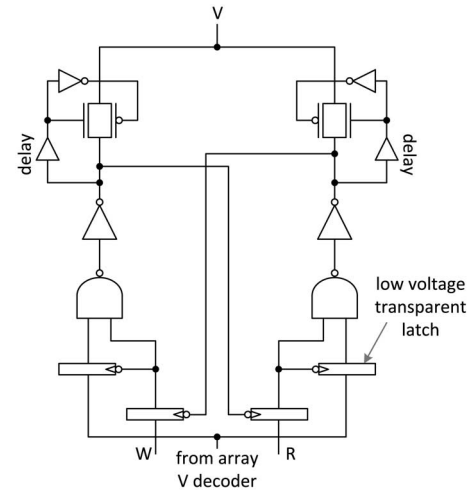


Fig. 7. The V_{ctrl} logic to hold the V line while releasing the V line decoder.

7 Implementations and Evaluations

In this section, we first elaborate the control circuit design of the existing hierarchical wordline and bitline architecture. This is essential to understanding how our proposed non-blocking bank can be implemented. We then evaluate its hardware overhead in 7.2. Finally, in 7.4 we present our experimental evaluations on throughput improvement, power budgeting schemes and performance comparison.

7.1 Circuit Implementation

Our PCM bank design follows a prototype from Samsung [21] with revised hierarchical GWL/GBL + LWL/LBL control. In those designs, hierarchical organization of wires provides more flexibility to optimizing GBL for low resistance, which is critical to delivering current to cells. We use this organization for implementing parallel accesses within a bank. The GWL/GBL will first be selected, followed by opening LWL/LBL in a local array. Then, further circuit support is needed to hold the LWL for the duration of a write while the GWL can be released for a new access. Next, we elaborate our hierarchical GWL/GBL + LWL/LBL control based on a realistic memory organization.

To make each array independently accessible, we place all 64 bits of a data in the same array, instead of distributing them into 4 arrays [21]. Because of such data placement, 64 GBLs are needed per column of arrays (Fig. 8). Such number of GBLs is physically feasible [20] with no area occupation as GBLs traverse over the array. Also, the global column mux (GCM) actually contains two separate muxes for SA and WD respectively [20].

Recall that the first challenge in our design is to release a row decoder after it drives the GWL high for a second request. We first introduce a set of horizontal (H) and vertical (V) signals as shown in Fig. 8. The intersection of an activated H and V opens an array and generates the corresponding enable signal (EN) for intra-array use. Since H and V signals are arranged in an 8×8 manner, two 3:8 decoders are sufficient for locating an array. When a row address arrives at the decoder, both GWL and H are driven high. One V line is then selected by the "array V decoder" when the column address arrives. This V signal then closes the latch in the intersection of V and H (Fig. 6(a)), which latches H internally. Hence, the H decoder can be freed. Meanwhile, the EN together with the selected GWL activates one LWL, and at the same time closes the passgate in all LWL drivers in the array (Fig. 6(b)), which latches GWL internally. Hence, the row decoder can be freed now.

The second challenge is to prevent the interference between the active write and read in their wordline and bitline cross-points. This challenge can be addressed by closing all arrays in the column so that they will not be affected by signals of a new request. Since an open V signal closes the latch in the array enable circuit as shown in Fig. 6(a), any further changes on H will not be seen as long as the V is active. Hence, V signal in fact locks the selection states of all arrays in the same column. In the active array, the active EN signal held by V closes the passgates, so the LWL driver will not see changes on GWLs. In inactive arrays, the inactive EN signal, also held by V, makes LWL drivers not responsive to GWLs. Furthermore, after being generated by the array V decoder, the V signal holding the entire column is in turn held by the write/read enable signals, shown as W and R in Fig. 8, with the V_{ctrl} circuits shown in Fig. 7. Thus, the V decoder can also be freed for another access.

The V_{ctrl} logic takes input from both the V line decoder, and a Read or Write command. If the input from the V line decoder is high (the V line should be driven), and either R or W command is high, it activates the V line and locks itself until the R or W drops low, i.e. the operation finishes. The circuit has a symmetric and cross-controlled structure. It ensures that when this V line is selected, and if one command line is high, its output will stay constant and not be affected by changes on the other command line. Through this way, the V_{ctrl} circuit can hold the V line by W or R command and dismiss the V line decoder for next request.

To summarize the procedure of issuing two accesses that overlap in time, we use a timing graph shown in Fig. 9 (not to scale) to illustrate the sequence of control signals for a write parallelized with a read. When a write request is issued to PCM, its row address will be first decoded by the H line decoder which then selects and pulls up one H line, H_w. At the same time, the row decoder activates one GWL, GWL_w. When the column address arrives, the V line decoder produces an output V_{out}_w which will make one V_{ctrl} logic ready for the upcoming W command. These signals will be discharged soon to take the next read operation. When the write command becomes high, the V_{ctrl} logic will open a V line, V_w,

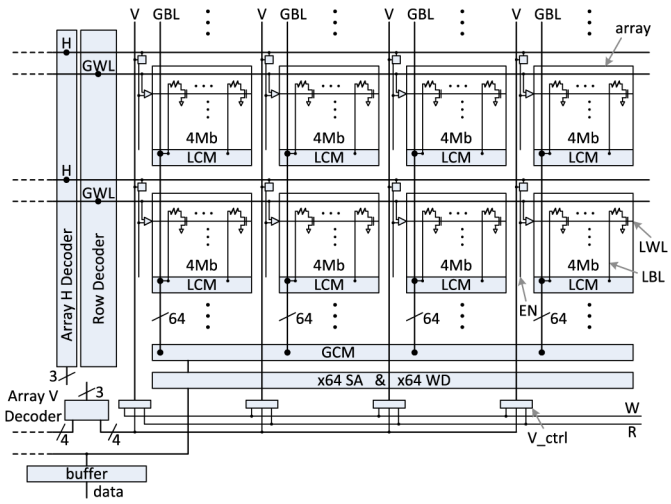


Fig. 8. Hierarchical wordline and bitline design in the right half bank. LCM/GCM: local/global column mux.

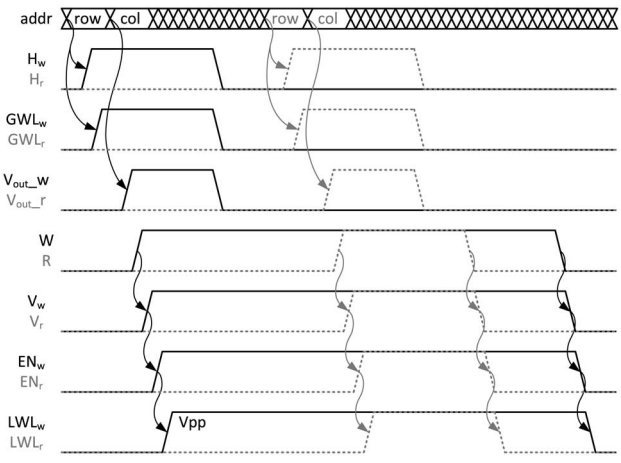


Fig. 9. Timing graph of a read in parallel with a write.

which will be held high as long as the W signal remains high. EN_w and LWL_w are then held high by V_w. When the write is in progress, the dismissed row decoder, H and V decoder are used by the following read access to generate H_r, GWL_r and V_{out}_r. The read command R will then trigger a similar series of signals, all of which occur in a different column of array.

7.2 Hardware Overhead Estimation

Most parts of our design such as chip organization, array partition, and hierarchical WL/BL are adopted from existing prototypes [20], [21], and thus do not incur overhead. The circuit components we added per PCM bank include: (1) H and V decoders, which are two 3:8 decoders; (2) H wire drivers ($\times 8$); (3) V_{ctrl} logic (Fig. 7, $\times 8$); (4) array enable circuit (Fig. 6(a), $\times 64$); (5) LWL drivers (Fig. 6(b), 64×2048), although these pre-exist in the prototype we referred to, we still study its overheads since the design is different now. We built and tested our added components in HSPICE with customized 45 nm PTM device model [35], and measured their delay, energy and area overheads. For area overhead, we also convert the number into percentage of the whole bank. To obtain the area of a bank, we scaled the dimensions reported in [21] from 100 nm to 45 nm. We also assume that the only additional H and V wires can traverse over cell arrays and thus do not occupy silicon area. All HSPICE simulations were performed with a power supply of 1.1 V at $90^{\circ}C$. Finally, the results are presented in Table 1.

When calculating the total delay overhead, we note that the H decoder (1) and driver (2) are carried in parallel with the row decoder and GWL driver. Therefore, their delay can be overlapped with the latter. The V decoder delay is still there because the column address arrives later than the row address. Hence, the total delay should be the sum of (1) and (3-5). The total energy overhead per access is less than 1.5pJ, which is almost negligible when compared to the large current of a write operation. The total area overhead is around 5% mainly due to the LWL drivers. We conservatively consider the LWL driver overhead in all metrics though this part also exist in the original design but are slightly different.

TABLE 1
Delay, Energy, and Area Overhead

	Delay (ps)	Energy (fJ)	Area(μm^2)	
			In one bank	% of bank
3:8 decoders	80	23.2	50.04	0.0005
H wire driver	240	499.8	84.96	0.0008
V_{ctrl}	850	446.8	259.2	0.0025
array_EN	105	250.7	904.32	0.0087
LWL driver	110	242.5	524288	5.03
TOTAL	1145	1486.4	525586.52	5.0425

7.3 Endurance Implications

AWP/RAWP do not increase the total number of PCM writes or change their physical addresses. Hence they are not expected to harm PCM lifetime. They do, however, require that the addresses of requests are "final". This means any address remapping caused by wear-leveling should be done before AWP/RAWP takes place.

Wear-leveling may impact BPB or BPB+FnW, as the power demand of a write request is determined by its number of bit changes. If a request is remapped to another physical address due to wear-leveling decision, its power demand could be higher than otherwise. Nonetheless, this address remapping is typically infrequent to reduce wear-leveling overhead. Therefore, we expect the impact on our power budgeting techniques to be small.

7.4 Experimental Evaluation

We first experiment with various hardware designs (baseline blocking design, Read-While-Write design, proposed non-blocking bank design) and scheduling enhancements (AWP, RAWP) to study their throughput improvements (Section 7.4.1) and row hit preservation (Section 7.4.2). Next, we estimate system performance under different scheduling enhancements in Section 7.4.3. We use Weighted Speedup [36] as the metrics. Finally, in Section 7.4.4, we use RAWP-enhanced memory scheduling and experiment with various power budgeting schemes on top of it. In the baseline power budgeting scheme, there is no bit change information from Differential Write, meaning that every bit is written in each round. Baseline also does not have other enhancements such as flexible write configurations, skipping redundant rounds as

TABLE 2
Notations of Various Designs and Schemes Used in Experiments

Hardware Design	RWW	Read-While-Write bank, 1 read, 1 write per bank
	NB	Proposed non-blocking bank, 2 reads, 2 writes per bank.
Scheduling Enhancements	PAR-BS/Half	PAR-BS with half-bank marking, no intra-bank re-ordering
	AWP	PAR-BS/Half enhanced with Aggressive Write-Precedence reordering
	RAWP	PAR-BS/Half enhanced with Row-hit Aware Write-Precedence reordering
Power Budgeting Schemes	FnW-only	Baseline enhanced with Flip-N-Write
	BPB	Proposed Bit Level Power Budgeting scheme
	BPB+FnW	BPB enhanced with FnW

TABLE 3
Parameter Used in Our Experiments

Platform	CPU	4x 3.2GHz cores
	Cache	32K private L1 cache, 4MB shared L2 cache, 64-byte cache lines
PCM Settings	Interface	Single DIMM, 8 chips, 8 banks, single channel DDR2-800 interface, 6.4GB/s peak bandwidth
	Read	50ns (row buffer miss) / 10ns (row buffer hit)
	Write	8–round setting: 1000ns (200ns + 8 rounds $\times$ 100ns) 4–round setting: 600ns (200ns + 4 rounds $\times$ 100ns)
	Row Buffer	Multiple-entry row buffer, 8 $\times$ 256B/entry
	Power Budget	256, 512, 768, 1024 (100% power budget in [21]) concurrent bit writes

we proposed in BPB. We then compare three different power budgeting schemes with the baseline:

- `FnW–only`, which simply adds Flip-N-Write (FnW) function to the baseline. With FnW, BPB always estimates that half of the bits in each round will be changed when checking against available power budget. It has limited flexibility to choose between 8-round and 4-round configurations. `FnW–only` does not skip redundant rounds as it does not retrieve and process the bit change information.
- `BPB`, which is our proposed Bit Level Power Budgeting scheme using bit change information from DW. In this scheme, the information from DW is used to estimate power demand for each write request. BPB supports flexible write configurations and can skip redundant rounds.
- `BPB + FnW`, which is our BPB technique enhanced with FnW. DW can provide fine-grained bit change information, but does not have an upper bound of how many bits are changed. In some cases when write requests have a high number of bit changes, using DW alone could result in high power demand and hurt throughput improvement. By combining BPB with FnW, we can effectively mitigate this problem.

The notations and explanations of the schemes we experimented are listed in Table 2.

We collected memory access traces and fed them to a detailed memory model to measure the memory throughput. We ran memory intensive benchmarks from SPEC2006, SPLASH2, and STREAM [37], [38] on a 4-core CMP simulator in Simics [39] with similar settings to [28]. Each core runs at 3.2GHZ with 4 MB shared L2 cache, a 128 MB DRAM buffer and 4 GB main memory. The memory module we used was developed in GEMS [40] framework. We heavily modified the memory controller module to model more details such as bank/half busy counters, bus contentions, row decoder contentions, column conflicts, channel contentions, etc. We then implemented our non-blocking reordering schemes based on this model. Further parameters are listed in Table 3.

7.4.1 Throughput Improvement

The first set of results is memory throughput improvement for various memory scheduling schemes. Throughput is calculated as number of requests served over their total finish time. Fig. 10(a) and (b) show results for 4-round and 8-round settings respectively. The results are normalized to blocking design that also uses PAR-BS scheduler.

As we can see, `RWW` achieves $3 \sim 5\%$ throughput improvement on average because of its limited hardware capability. Using NB without any scheduling enhancement `NB + PAR − BS` improves throughput to 24% because our hardware provides much more parallelism. Further parallelism can be achieved through scheduling enhancement. `PAR − BS/Half` does a simple change to balance batch size

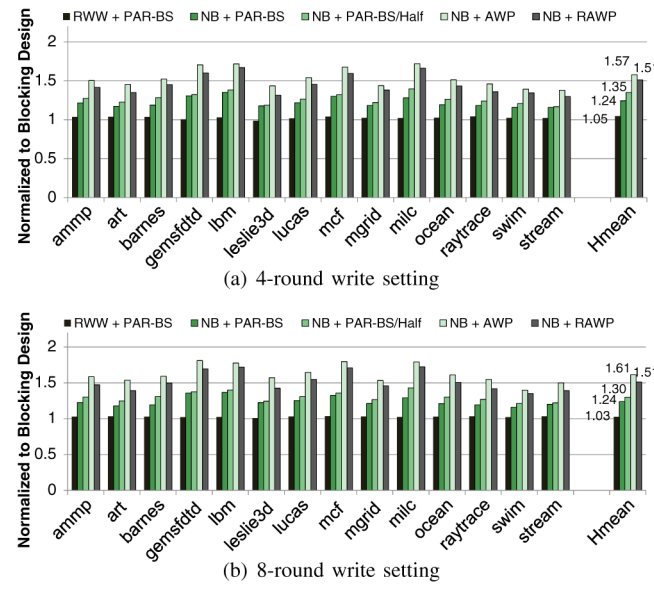


Fig. 10. Throughput improvement.

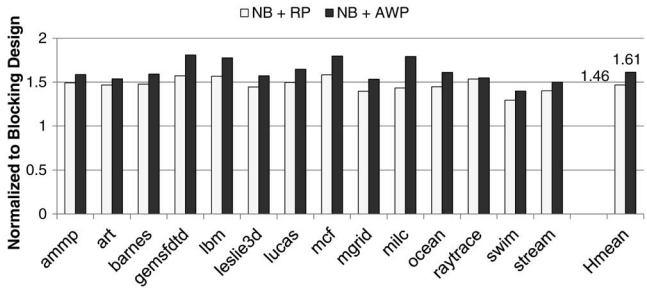


Fig. 11. Comparing throughput improvements with a scheme that puts read requests first (NB + RP).

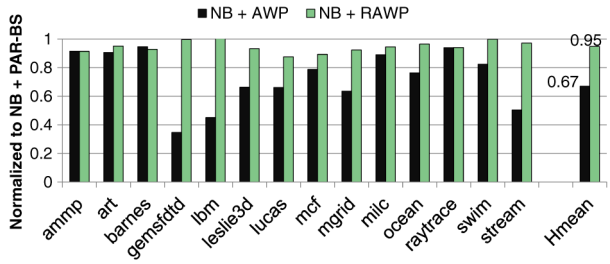


Fig. 12. Read row hit rate under different scheduling enhancements.

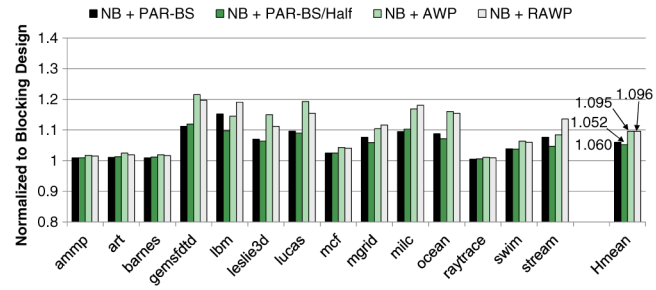


Fig. 13. Weighted Speedup [36] for different scheduling enhancements.

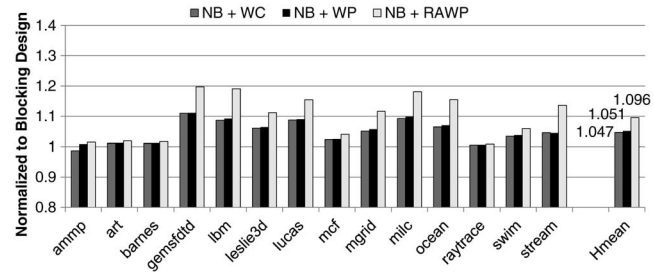


Fig. 14. Weighted Speedup [36] for Write Cancellation, Write Pausing [15] and RAWP.

between two halves inside each bank, resulting $30 \sim 35\%$ throughput increase. AWP achieves the highest improvement ($57 \sim 61\%$ on average) among all because of its aggressive reordering algorithm. RAWP not surprisingly has smaller improvement (51%) than AWP because of a carefully crafted algorithm that preserves row buffer hit rate.

Our write-precedence reordering scheme often prioritizes write requests over read requests as discussed in Section 5.2. The motivation is backed by our experimental results in Fig. 11, which compares the throughput improvements between NB + AWP and a scheme that puts read requests first (NB + RP). On average, our proposed write-precedence scheme achieves 15% more throughput improvement over the baseline.

7.4.2 Preserving Row Buffer Hits

As discussed earlier, RAWP overcomes the shortcomings of AWP and preserves row buffer hit rates. Fig. 12 shows the read row buffer hit rates for AWP and RAWP respectively. The results are normalized to non-blocking bank design with PAR-BS NB + PAR − BS. AWP degrades row buffer hit rate by 33% on average due to its aggressive reordering design. On contrary, RAWP preserves row buffer hit rate while achieving a comparable throughput improvement.

7.4.3 Performance

We estimated system performance improvement (Weighted Speedup [36]) under different scheduling enhancements, as shown in Fig. 13. We assume no power constraints in this experiment (i.e. power budgeting is disabled). The performance metric we used is Weighted Speedup [36], which is commonly used to measure the performance improvement for multiprogrammed systems.

Using non-blocking bank design without any intra-bank reordering (NB + PAR − BS and NB + PAR − BS/Half) results in 5.9/5.2% improvement. With scheduling enhancements, AWP and RAWP achieve 9.5% and 9.6% improvement respectively. Although AWP can achieve higher throughput improvement, its low read row buffer hit rate hurts read latency. As a result, AWP achieves slightly lower performance improvement than RAWP on average.

In Fig. 14, we compare the performance of RAWP with Write-Cancelation (WC) and Write Pausing (WP) [15]. When a read is blocked by an on-going write, WC allows the write to be canceled and issues the read first. The canceled write request is restarted later, meaning that the time that is already spent is wasted. WP uses similar idea to optimize read latency, except that write operation is resumed from where it left off, saving the time it already spent. As we can see, both techniques only optimizes read latency but not memory throughput. We built these techniques on top of non-blocking bank design and PAR-BS/Half to perform a fair comparison (NB + WC and NB + WP). The average performance improvement for WC, WP and RAWP are 4.7%, 5.1% and 9.6% respectively. RAWP aims to produce request parallelism, including read-read parallelism which is particularly helpful in reducing the average read latency. Write-cancelation and write-pausing, however, only preempt a write when a read comes in, so it cannot exploit the benefit produced by parallelism.

7.4.4 Throughput under Power Budget

We now present throughput results for different power budgeting schemes (applied on top of RAWP) in Fig. 15. The results are averaged over all benchmarks tested. All results are normalized to the baseline power budgeting scheme, and various power budgets are experimented.

As expected, both BPB and BPB + FnW achieve more throughput improvement because power budgeting is done at bit-level.Naturally, combing both DW and FnW (BPB + FnW) is most effective since it further lowers the bit changes in a write. An interesting observation from this experiment is that both schemes become more and more effective when power budget is decreasing. Note that the

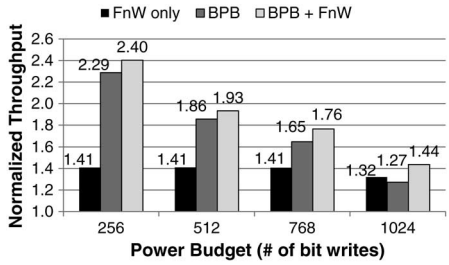


Fig. 15. Average throughput between power budgeting schemes, under different power constraints.

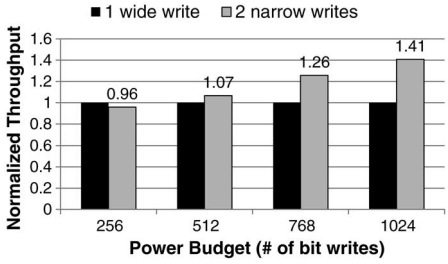


Fig. 16. Average throughput normalized to 1 `widewrite`; both use `BPB + FnW`.

throughput of baseline is also decreasing, but the improvements increase. This is because, when available power budget is low, the baseline's parallelism is more severely limited as it assumes each round changes every bit. On contrary, `BPB` and `BPB + FnW` leverage the fine-grained bit change information, so they can skip redundant rounds and can utilize the limited power budget more efficiently.

The next experiment tests that given a power budget, if our scheduler should issue one write with a wide width, say `W`, or two parallel writes with `W/2` width each. The former results in fewer rounds per write while the latter may result in fewer total rounds for the two writes, as illustrated in Section 6.2. Fig. 16 shows the throughput comparison between the two schemes. In most cases, having two parallel but narrower writes generates better throughput, in accordance with the example we showed. When power budget is low, e.g., 256, even issuing two parallel narrow writes is not possible and the two writes must be sequentialized. In those cases, issuing one write with wider width may win (by 4% on average). Such scenario can be picked up by our scheduler as if two writes cannot be parallelized, a proper write width with earliest write finish time, i.e., least rounds, will be selected.

Comparing to Power Token. Finally, we compare our proposed scheme `RAWP` and `BPB + FnW` to the Power Token (`PT`) scheme [30] with varying power budgets. The results are in Fig. 17. The original PT scheme supports only single-round writes, which is disadvantageous when power budget is low. We hence extended it to support multi-round writes (a fixed 4 round configuration is used). Also, we are optimistic about `PT` in bit change estimation and assume that it is accurate, as the original scheme only used approximate information. Hence, the true results of the original `PT` would be lower than what we show here. Since all three schemes we compare aim to improve PCM throughput, we use the *blocking* bank design as our baseline in this experiment.

When there is only 1/4 of the full power budget (256 bits), improving throughput with `RAWP`-only is limited because

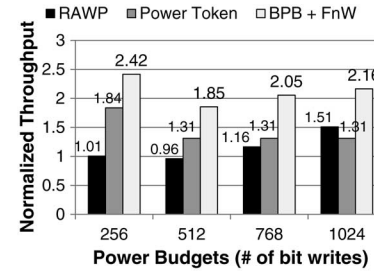


Fig. 17. Comparing throughput improvements with Power Token [30].

parallel writes would be throttled due to insufficient power budget. Hence, both `PT` and `BPB + FnW` are more effective with the latter being 58% better than the former. This is mainly due to the flexibility of `BPB + FnW` in choosing write configurations to better utilize power budget and our non-blocking bank design. The second important observation is that `PT` does not show more throughput improvement when power budget increases. This is because it solely relies on bit-level power consumption to increase throughput. When the maximal potential is achieved, no further opportunities are leveraged to continue increasing the throughput. As a result, its improvement levels off with more power budget. In contrast, `RAWP` and `BPB + FnW` can achieve higher throughput when power budget increases. `BPB + FnW` is superior to `PT` by up to 85% (1024) due to both our non-blocking bank design with scheduling enhancement, and the power budgeting algorithm.

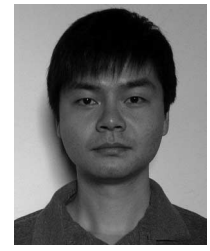
8 Conclusion

We proposed a novel PCM bank design to mitigate its low bandwidth problem due to long write latency. Our technique essentially allows a bank to serve up to 2 reads and 2 writes simultaneously which provides new opportunity of intra-bank parallelism. To make such design effective to throughput improvement, we proposed write-precedence intra-bank reordering schemes (AWP and RAWP) to fully take advantage of our non-blocking bank design. Our scheduling enhancements can achieve up to 61% throughput improvement over the original blocking bank design. Finally, we developed power budgeting schemes to maintain higher throughput under power budgets. We found that our bit-level power budgeting scheme can further increase throughput since power control is done in fine-granularity so more requests can be issued in parallel. Such increase is more evident when the available power budget is low. Also, under the same power budget, it is generally more beneficial to issue parallel writes, each with narrow bit width, than to issue single write with a wider bit width.

References

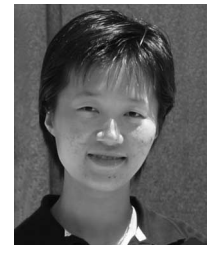
[1] K. Kim, "Technology for sub-50 nm DRAM and NAND flash manufacturing," in *Proc. Annu. IEEE Int. Electron Dev. Meeting (IEDM)*, 2005, vol. 5, pp. 323–326.
[2] C. Lefurgy et al., "Energy management for commercial servers," *Computer*, vol. 36, no. 12, pp. 39–48, Dec. 2003.
[3] S. Cho and H. Lee, "Flip-n-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2009.
[4] G. Dhiman et al., "PDRAM: A hybrid PRAM and DRAM main memory system," in *Proc. Des. Autom. Conf. (DAC)*, 2009, pp. 664–469.
[5] E. Ipek et al., "Dynamically replicated memory: Building reliable systems from nanoscale resistive memories," in *Proc. Int. Conf. Architectural Support Program. Languages Oper. Syst. (ASPLOS)*, 2010.

[6] B. C. Lee et al., "Architecting phase change memory as a scalable DRAM alternative," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, 2009, pp. 2–13.
[7] M. K. Qureshi et al., "Morphable memory system: A robust architecture for exploiting multi-level phase change memories," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, 2010.
[8] M. K. Qureshi et al., "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2009, pp. 14–23.
[9] M. K. Qureshi et al., "Scalable high performance main memory system using phase-change memory technology," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, 2009, pp. 24–33.
[10] S. Schechter et al., "Use ECP, not ECC, for hard failures in resistive memories," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, 2010.
[11] N. H. Seong et al., "Security refresh: Prevent malicioius wear-out and increase durability for phase-change memory with dynamically randomized address mapping," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, 2010.
[12] W. Zhang and T. Li, "Characterizing and mitigating the impact of process variations on phase change based memory systems," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2009.
[13] W. Zhang and T. Li, "Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures," in *Proc. Parallel Architectures Compilation Tech. (PACT)*, 2009, pp. 101–112.
[14] P. Zhou et al., "A durable and energy efficient main memory using phase change memory technology," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, 2009, pp. 14–23.
[15] M. K. Qureshi et al., "Improving read performance of phase change memories via write cancellation and write pausing," in *Proc. High Perform. Comput. Architecture (HPCA)*, 2010.
[16] A. P. Ferreira et al., "Increasing PCM main memory lifetime," in *Proc. Des. Autom. Test Eur. (DATE)*, 2010.
[17] C. Villa et al., "A 45 nm 1 GB 1.8v phase-change memory," in *Proc. Int. Solid-State Circuits Conf. (ISSCC)*, Feb. 2010.
[18] *DDR2 SDRAM* [Online]. Available: http://en.wikipedia.org/wiki/DDR2_SDRAM.
[19] N. C. T. Office. (2009, Sep.) *Phase Change Memory and Its Impacts on Memory Hierarchy*, Carnegie Mellon [Online]. Available: http://www.pdl.cmu.edu/SDI/2009/slides/Numonyx.pdf.
[20] K.-J. Lee et al., "A 90 nm 1.8v 512 mb diode-switch PRAM with 266 mb/s read throughput," *IEEE J. Solid-State Circuits*, vol. 43, no. 1, pp. 150–162, Feb. 2008.
[21] S. Kang et al., "A 0.1-m 1.8-v 256-mb phase-change random access memory (PRAM) with 66-mhz synchronous burst-read operation," *IEEE J. Solid-State Circuits*, vol. 42, no. 1, pp. 210–218, Jan. 2007.
[22] Y. Agata et al., "An 8-ns random cycle embedded RAM macro with dual-port interleaved dram architecture (d2ram)," in *Proc. Int. Solid-State Circuits Conf. (ISSCC)*, 2000.
[23] TOSHIBA. *Toshiba Develops 32 mb Read While Write Nor Memory Devices* [Online]. Available: http://www.toshiba.com/taec/news/press_releases/2000/to-077.jsp.
[24] H. Zheng et al., "Mini-rank: Adaptive dram architecture for improving memory power efficiency," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2008.
[25] J. H. Ahn et al., "Future scaling of processor-memory interfaces," in *Proc. IEEE Conf. Supercomput. (SC)*, 2009.
[26] S. Rixner, "Memory controller optimizations for web servers," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2004.
[27] S. Rixner et al., "Memory access scheduling," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, 2000.
[28] O. Mutlu and T. Moscibroda, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, 2008, pp. 63–74.
[29] Y. Kim et al., "Thread cluster memory scheduling: Exploiting differences in memory access behavior," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2010.
[30] A. Hay et al., "Preventing PCM banks from seizing too much power," in *Proc. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, 2011.
[31] K.-J. Lee et al., "A 90 nm 1.8v 512 mb diode-switch PRAM with 266 mb/s read throughput," in *Proc. Int. Solid-State Circuits Conf. (ISSCC)*, 2007, pp. 472–616.
[32] MICRON. *2 GB DDR2 SDRAM Component* [Online]. Available: http://www.micron.com/products/partdetail?part=MT47H256M8HG-25.
[33] F. Bedeschi et al., "A bipolar-selected phase change memory featuring multi-level cell storage," *IEEE J. Solid-State Circuits*, vol. 44, no. 1, pp. 217–227, Jan. 2009.
[34] H. Chung et al., "A 58 nm 1.8 V 1Gb PRAM with 6.4 MB/s Program BW," in *Proc. Int. Solid-State Circuits Conf. (ISSCC)*, 2011.
[35] A. S. University. *Predictive Technology Model (PTM)* [Online]. Available: http://www.eas.asu.edu/ptm/.
[36] A. Snavely and D. Tullsen, "Symbiotic jobscheduling for a simultaneous multithreading processor," in *Proc. Int. Conf. Architectural Support Program. Languages Oper. Syst. (ASPLOS)*, 2000.
[37] J. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Tech. Committee Comput. Architecture (TCCA)*, 1995.
[38] S. Woo et al., "The splash-2 programs: Characterization and methodological considerations," in *Proc. Int. Symp. Comput. Architecture (ISCA)*, 1995.
[39] P. S. Magnusson et al., "Simics: A full system simulation platform," *Computer*, vol. 35, no. 2, pp. 50–58, Feb. 2002.
[40] M. K. Martin, et al., "Multifacet's general execution-driven multiprocessor simulator toolset," *Comput. Archit. News*, 2005.



Ping Zhou received the BS and MS degrees in computer science from Shanghai Jiao Tong University, China, in 2001 and 2004, and the PhD degree in computer engineering from the University of Pittsburgh, Pennsylvania, in 2012. His research interests include emerging memory technologies and 3D CMP. He was with Intel (Shanghai), working on media SoC software between 2004 and 2007. He is now a Firmware engineer at Intel Corporation.

Bo Zhao received the MS degree from University of Pittsburgh, Pennsylvania, in 2009, and the BS degree from Beihang University, Beijing, in 2007. He graduated from University of Pittsburgh with PhD degree in Computer Engineering in 2013. His research interests include VLSI circuits and microarchitectures, memory systems, and modern processor architectures. He is now a Circuit Design Engineer at Apple, Inc.



Jun Yang received the BS degree in computer science from Nanjing University, China, in 1995, the MA degree in mathematical sciences from Worcester Polytechnic Institute, Massachusetts, in 1997, the PhD degree in computer science from the University of Arizona in 2002. She is an associate professor with the Electrical and Computer Engineering Department, University of Pittsburgh, Pennsylvania. She is the recipient of US NSF Career Award in 2008. Her research interests include low power, and temperature-aware microarchitecture designs, new memory technologies, and networks-on-chip.

Youtao Zhang received the PhD degree in computer science from the University of Arizona in 2002. He is an associate professor in Computer Science Department, University of Pittsburgh, Pennsylvania. His research interests include the areas of computer architecture, program analysis, and optimization. He is the recipient of US NSF Career Award in 2005, the distinguished paper award of the IEEE/ACM International Conference on Software Engineering (ICSE) conference in 2003, and the most original paper award of the International Conference on Parallel Processing (ICPP) conference in 2003. He is a member of the ACM.