2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture

Boosting Access Parallelism to PCM-based Main Memory

Mohammad Arjomand Mahmut T. Kandemir Anand Sivasubramaniam Chita R. Das
The School of Electrical Engineering and Computer Science
Pennsylvania State University, PA 16802, USA
mxa51@psu.edu {kandemir,anand,das}@cse.psu.edu

***Abstract*—Despite its promise as a DRAM main memory replacement, Phase Change Memory (PCM) has high write latencies which can be a serious detriment to its widespread adoption. Apart from slowing down a write request, the consequent high latency can also keep other chips of the same rank, that are not involved in this write, idle for long times. There are several practical considerations that make it difficult to allow subsequent reads and/or writes to be served concurrently from the same chips during the long latency write. This paper proposes and evaluates several novel mechanisms – re-constructing data from error correction bits instead of waiting for chips currently busy to serve a read, rotating word mappings across chips of a PCM rank, and rotating the mapping of error detection/correction bits across these chips – to overlap several reads with an ongoing write (RoW) and even a write with an ongoing write (WoW). The paper also presents the necessary micro-architectural enhancements nee-ded to implement these mechanisms, without significantly changing the current interfaces. The resulting PCM access parallelism (PCMap) system incorporating these enhancements, boosts the intra-rank-level parallelism during such writes from a very low baseline value of 2.4 to an average and maximum values of 4.5 and 7.4, respectively (out of a maximum of 8.0), across a wide spectrum of both multiprogrammed and multithreaded workloads. This boost in parallelism results in an average IPC improvement of 15.6% and 16.7% for the multi-progra-mmed and multi-threaded workloads, respectively.**

***Keywords*-Phase Change Memory, Write performance.**

I. Introduction

The continuing need to boost main memory capacities for handling big data problems is directing more attention on alternate storage technologies that can potentially replace DRAM. Even when idle, DRAM dissipates considerable power and can account for as much as 40% of the power consumed by a high-end server [1]. The problem exacerbates with increasing DRAM memory capacities, making such servers less energy proportional. DRAM is also difficult to scale down due to various limitations including device leakage and challenges in integrating large capacitors. Phase Change Memory (PCM) [2]–[4] is one promising technology that is gaining interest as a DRAM replacement to address these problems. Apart from non-volatility, PCM has several promising features – zero standby power, low read latencies, resilience to soft errors, and high densities that can scale better with technology, making it an attractive alternative for building main memories [4]. However, its endurance and write performance have been noted [2], [4], [5] as serious detriments to its widespread adoption. In this paper, we specifically focus on *write performance*, showing how, despite its inherent long latency, we can concurrently serve other requests to significantly boost PCM performance.

Performance of any memory system, including PCM, is determined by two attributes: latency and throughput. The former denotes the minimum time required to service a request, even when it is the only request in the system. Rather than speeding up a single request, this paper seeks to improve the throughput (the number of requests served per unit time) of the PCM system. Throughput can be enhanced by improving the raw latency required to serve each request. Equally important is the scheduling of these requests and maximizing the number of requests that can be serviced at any time. The latter artifact, referred to as concurrency or parallelism, is a crucial factor in determining how memory systems, whether it be DRAM or PCM, are organized, and is the subject of optimization in this paper.

Reads and writes are both slower on PCM than on DRAM; especially write latencies can be 2–8 times higher than DRAM [2]. Such a high write latency has an important implication – *if only a subset of chip(s) of a PCM bank is serving a write request*[1]*, the remaining chips of that rank will be idle for the long duration of this write* – making it even more imperative to avoid such idling in the case of PCM compared to DRAM. We need to look beyond this request, to subsequent read and write requests, that could potentially be serviced by such idling chips during the long latency write. As we will show, a majority of writes across a spectrum of applications need to update data in only a small number of chips of a PCM rank, mandating techniques for utilizing the remaining chips (we refer to these as *uninvolved* chips for serving a write request) to serve other requests in parallel. Even though prior work [2] has also identified that the "dirty" data which needs to be actually written back is at most a few words of the cache line being written back, it has only been leveraged for power savings and endurance related issues in the context of PCM. Our work, on the other hand, leverages this information to focus the writes on a subset of PCM chips in a rank, freeing up the others for potentially

[1]With differential writes [3], before a cache line is written, the old value is read out of the array, compared with the new data to be stored, and only the bits that need to change are then written in PCM. In a DIMM architecture, using differential writes can make a subset of chips idle.

1063-6897 2016

DOI 10.1109/ISCA.2016.66

servicing other requests.

When a write to a PCM memory is ongoing in a subset of chips, we could boost the Intra-rank-Level Parallelism (IRLP)[2] by concurrently serving (i) other write requests which are directed (or need changing) at the uninvolved chips of the ongoing write, (ii) several other read requests one after another (since each read takes only 15%–50% of the time for a write) even if each such read spans all the uninvolved chips, and (iii) possible combinations of the two. However, such servicing of other requests during an ongoing write poses several challenges:

- Reads to main memory are at cache line granularities, and the words of a cache line get striped across all chips of the rank (and not just uninvolved chips). Fetching the words from the uninvolved chips during the write may not help since the read has to still wait until the write is done to fetch the words from the chips servicing the write, to return the entire cache line.
- Even if we are selective about only writing to the chips where data changes, it is possible that the dirty words in multiple write-backs get assigned to the same chips limiting the potential of overlapping one write with another.
- In memories with error detection/correction support, data integrity needs to be checked for every read using some error detection/correction mechanism, and the corresponding bits (which are themselves stored in PCM chips) need to also be written during the write operation. Hence, even if we are able to successfully divert subsequent reads and/or writes to uninvolved chips during an ongoing write, all of these accesses could potentially contend for the PCM chip(s) holding the error detection/correction bits, thereby defeating the purpose.

This paper introduces several novel mechanisms to tackle these challenges when boosting parallelism in the PCM ranks during long latency writes. Using detailed multicore platforms simulated in the full system mode, and a wide spectrum of both multi-programmed (from SPEC CPU 2006) and multi-threaded (from PARSEC and STREAM) workloads, this paper makes the following specific **contributions**:

- We show that only a few selective words in each cache line need to be written back to PCM (in agreement with prior observations) freeing up the remaining (uninvolved) chips to service other accesses. Across the entire set of workloads, we show that 14–52% of the write-backs have only 1 word dirty, and 77–99% of the write backs have less than 4 words (50% of a cache line) dirty. If the uninvolved chips are to be kept idle during writes, we can only expect an average intra-rank-level parallelism of 2.37 during such writes for a rank with 8 chips, where each word of the cache line maps to a different chip.
- Given the large number of writes that need to update only 1 word of a cache line, we propose to serve subsequent reads, for the bank currently involved in single word writes, by leveraging a simple correction mechanism (i.e., parity) to reconstruct the word that would have been provided by the chip involved in the write. The error checks, which would need the actual data from that chip, do not need to be in the critical path. We find that, on average, 42% of the reads that come soon after an ongoing write, can be serviced in parallel with this enhancement, providing 9.8% performance benefit on the average across the workloads.
- To relieve the contention of dirty words written by successive write-backs on the same chips, we rotate the allocation of words of a cache line to the chips of a rank. Consequently, the same word offset in successive cache lines would map on to different chips. Allowing such successive writes to overlap with ongoing writes gives an average 5.3% additional improvement compared to just allowing successive reads to overlap with the ongoing writes. To avoid the need for book-keeping information for rotation, the controller uses the cache line address to determine the rotation offset. Therefore, the rotation offset of each cache line is fixed during reads and writes.
- Reserving a single PCM chip to hold the error detection/correction bits of all cache lines of that bank creates undue contention for it. Successive write-backs cannot be accommodated (even if their dirty words map to different chips) since they all need to update the detection/correction bits (unless those updates are propagated in the background during idle periods). We find that allowing writes to overlap with ongoing writes gives only 6.7% improvement without addressing this problem. Further, subsequent reads may also contend with the writes for these chips (even if error checking is postponed to the background and does not fall within the critical path of the read latency). Hence, allowing writes and reads to both overlap with ongoing writes, gives only 10.6% improvement on the average, because of contention for the error detection/correction chips. We consequently propose rotating the error detection/correction bits across the chips (similar to the rotating parity of RAID-5). This rotation significantly boosts the parallelism, giving 5.2% additional performance improvement, compared to confining all such bits to a single chip.

With all these enhancements, our *PCM access parallelism (PCMap)* system gives an overall intra-rank-level parallelism of 4.5 (up to 7.4) during the writes, compared to 2.3 for the default PCM-based main memory, which has a rank with 8 chips. Examining a 8-core processor with 8GB main memory capacity (connected by 4 channels), this boost in parallelism leads to an overall IPC improvement of 15.6% and 16.7% over the baseline PCM main memory for multi-threaded and multi-programmed workloads, respectively.

[2]We use the term "intra-rank-level parallelism during a write" (IRLP) as the number of chips in the rank that are actively serving some request during that period. IRLP gives an indication of how well we are able to utilize the uninvolved chips during the writes to a rank.

II. Background

A. PCM Memory Architecture

We describe a typical PCM memory architecture that is same as DRAM memory in modern processors. Our discussion in this paper will focus on the dominant memory architecture today, i.e., JEDEC-style DDR3.

A high-performance processor typically implements up to four memory controllers. Each memory controller handles a 64-bit DDR3 data channel with an address/command bus with a typical width of 23 bits [6]. Multiple DIMMs (Dual In-line Memory Modules) can be accessed via a single channel and memory controller. Each channel typically supports 1–4 ranks. A rank is a collection of PCM chips that together feed the 64-bit data bus, i.e., in the common case, a rank may contain 8 $\times$8 chips, or 4 $\times$16 chips, or 16 $\times$4 chips ($\times N$ refers to a chip chip with N data bits of input/output on every clock edge). DDR3 has a minimum burst length of 8, i.e., a request results in eight 64-bit data transfers on the bus. To fetch a cache line, the memory controller first issues an `Activate` command, followed by a `Column-Read`. Each `Column-Read` results in a burst of 8 from each chip in that rank, yielding a 64-byte cache line. The `Active` command brings an entire row of data (about 8 KB) into a row-buffer. Adjacent cache lines in the row can be fetched with multiple `Column-Read` without requiring additional `Actives`. Each rank is itself partitioned into 8 banks in DDR3. The 8 banks are independently controlled and have their own row-buffers.

In a DIMM memory with ECC support (ECC-DIMM), typically a Hamming code with seven check bits (or a similar code) is used to provide single bit error correction (SEC) for 64 bits of data. In addition to the SEC code, an additional parity bit is provided to enable double-bit error detection (DED). This introduces 8 bits of overhead per 64 bits of data, which is implemented by adding a ninth chip to the DIMM. All 72 bits are read in parallel and the 8 bits of SECDED coding are used to check and possibly correct an error in one of the 64 bits; this does not increase latency.

B. Scheduling at Memory Controller

The memory scheduler has to consider resource availability and several timing constraints when issuing commands. Generally, the memory scheduler prioritizes reads over writes, accesses to open rows, and older requests over younger ones. Below, we describe these policies in details:

Read-over-Write Priority: Memory writes are generated as a result of write-back from the LLC. Since writes are not on the processor's critical path, the memory-controller is not required to complete the write operation immediately and buffers the data in a write queue (this is a common feature in DRAM and PCM controllers). This is even more important in PCM memories where, unlike DRAM, access latencies are *asymmetric* for read and write operations (writes are 2–4 times slower than reads). One of the many timing constraints is the write turnaround delay that is incurred every time the bus changes direction when switching from a write to a read. Writes and reads are generally issued in bursts to amortize this delay overhead. To conclude, memory controllers usually buffer writes till the write queue is $> \alpha\%$ full (or there are no pending reads); the bus is then turned around and writes are drained until a lower threshold is reached. At this point, the controller resumes servicing read requests and changes the bus direction. This policy ensures that read requests are given priority, but write requests eventually get a chance to be serviced.

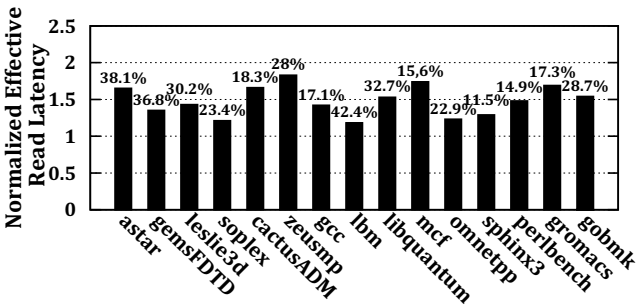

Figure 1: Percentage of read requests delayed when serving a write (shown on top of the bars) and the read latency of an asymmetric PCM memory normalized to that of a symmetric PCM (it has equal read and write latencies).

First-Ready-First-Come-First- Served (FR-FCFS) Priority: Regardless of the memory technology (DRAM or PCM), the controller prioritizes accesses to open rows because their data is ready in the row-buffer and accessing the row-buffer takes less time than accessing the memory array. Among those accesses in the row-buffer, the controller also services the older requests before the younger requests.

III. Problem and Motivation

A. Write Problem in PCM

A write request in PCM negatively affects performance from two aspects.

1) With the memory architecture described in Section II, any read or write request makes the bank busy till its completion. Since PCM write latency is considerably larger than that of a read, when a bank is busy servicing an ongoing write, the read requests arriving later should wait, and this increases the effective read latency. To show how write latency affects the service time of a read in PCM, we performed an experiment. We assumed a 8-core processor with 8GB PCM main memory (detailed configuration setting in Section V), and measured the access latency for each memory request for our analysis. Figure 1 shows the results for a set of programs from SPEC CPU 2006 – the program set includes a spectrum of applications with diverse memory access intensities and read-write ratios. This graph plots (1) the percentage

of read requests whose service times are affected because of servicing a write (it is not only due to the queuing latency of the read queue), and (2) the (average) effective read latency *normalized* to a case where the write latency is equal to the read latency (symmetric PCM). This plot shows that the large latency writes can degrade the performance of 11.5% to 38.1% of all read requests by increasing their effective latencies by 1.2 to 1.8 times. Consequently, in a PCM main memory, one *cannot* afford to ignore write performance by simply assuming that they are non-critical compared to reads; in fact, writes can also slow down critical reads. Note that the same issue presents in DRAM, but it gets magnified in PCM.

2) PCM chips have severely limited write bandwidth. For example, a prototype $20nm$ 8Gb PCM chip [7] has a read bandwidth of 800Mbit/Sec/pin and a write bandwidth of 40MB/Sec (it can be as high as 133MB/Sec by applying a higher external voltage). DRAM, on the other hand, has a maximum bandwidth of 12.8GB/Sec in 800MHz DDR3. Two factors limit the write bandwidth of PCM compared to DRAM. First, PCM write latency is greater than that of DRAM, and second, writing to a PCM cell takes more energy than a DRAM write. To provide the same write bandwidth as DRAM, PCM would require five times more power than the latter [8].

B. Motivation

It is well-known that, a significant portion of the writes in main memory are redundant. That is, in most cases, a write into a word does not change its value, either because (i) that word is not written into the cache line by the CPU, but at least one word of the same cache line is updated (in a non-sectored-cache a write-back forces all words in a cache line to be updated in main memory), or (ii) even if it is written, the content values have not changed (i.e., silent stores [9]). These writes are hence unnecessary. A few prior works [2], [3] that have considered this redundancy for power optimization or lifetime improvement have attempted to study data redundancy at the granularity of bits or cache lines. Instead, in this work, we focus on another redundant update opportunity that has not been leveraged by prior works. Specifically, we study write redundancy of a sub-block of a cache line that is assigned to one PCM chip. We call it a *word* without loss of generality. For instance, with a 64B cache line size and 8 chips in one rank, the granularity of our redundant data analysis would be 8B (word size). Here, we assume that every 8 adjacent bytes of a cache line are mapped to the same chip (starting from bytes 0–7 in the first chip and ending at bytes 56–63 in the eighth chip). Later in Section IV-A2, we will describe this data layout.

The graph in Figure 2 plots the redundancy of 8B word write-backs for a set of SPEC CPU 2006 programs (Section V gives details of the examined configuration). We monitored memory writes over the entire execution of each

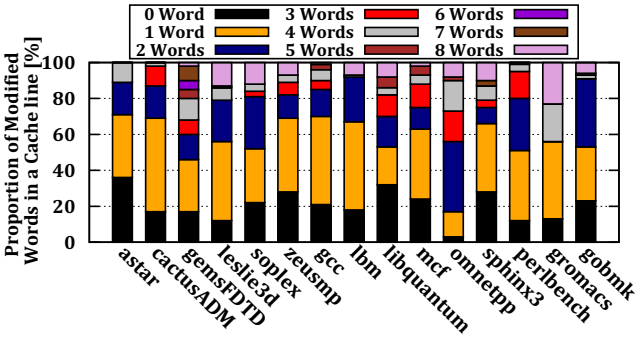

Figure 2: Percentage of cache line writes (each 64B) that need updating only i 8B-words ($0 \leq i \leq 8$). For each application, we have a stacked bar where the segment at the bottom is for 0-Word-update, the second segment from the bottom is for 1-Word-update, and so on.

program for this analysis. In this figure, we plot the percentage of *memory writes* (cache lines that are identified as *dirty* on eviction) where only *i* words are actually modified (among all its 8 words). *i* can have a value between 0 and 8; it is zero when a cache line is completely unmodified (although it is marked as dirty in the last-level cache, hence it is a silent store) and it is 8 when all its eight words have values different from those stored in the main memory. We call the writes related to these *i* dirty words, the *"essential"* writes. In this paper, we do not consider optimizations related to the cases where the update of a cache line is completely redundant ($i = 0$), since prior work [2] have already investigated this case. Rather, we are mainly interested in cases where only a few words need to be updated. One of the valuable observations that this figure reveals and is extensively exploited in the rest of paper is that the percentage of write requests that would update only one word in main memory is high, at least 14% (`omnetpp`) and up to 52% (`cactusADM`).

Further, the essential words in 76.6% of the write requests (on average) are less than 4 in majority of applications. To sum up, the actual amount of data to be updated on each cache line write is limited to few words (1 to 4) of the write requests[3]. This observation means that most of the write requests update only 1 to 4 words in the main memory and the intra-rank-level parallelism during these writes is only around 2.3 on the average across these applications. Motivated by this result, the rest of this paper sets out to answer the following question: *by confining the "essential" writes to a subset of PCM chips, can one free up the other chips to serve the other writes and/or reads?* If this can be

[3]We repeated the same experiment without considering *silent stores*: For the applications in Figure 2, the average percentage of 0-Word-update is 17.2%; 29.5% for 1-Word-update; 14.1% for 2-Words-update; 7.2% for 3-Words-update; 12.9% for 4-Words-updates; 5.8% for 5-Words-update; 1.8% for 6-Words-update; 2.3% for 7-Words-update; and 9.2% for 8-Words-update. It is still highly probable that only one word out of 8 in a cache line needs update (29.5%) or up to 4 out of 8 words need update (66%).

achieved, the serialization penalty imposed by long latency write operations can be reduced significantly.

IV. PCM Access Parallelism (PCMap)

We propose a memory system that enables the PCM chips *uninvolved* in a write operation to serve other requests to the same bank. It is named as PCM access parallelism or *PCMap*. The basic idea behind PCMap is that, when a write is involving a subset of PCM chips in a rank, we could boost the intra-rank-level parallelism (IRLP) by (i) concurrently serving other write requests that can be served using the chips not participating at the ongoing write, (ii) concurrently serving several read requests, one at a time, though this requires satisfying a constraint that will be discussed later, and (iii) possible combinations of the two.

PCMap is based on two novel mechanisms: *Read over Write (RoW)* and *Write over Write (WoW)*. RoW enables the concurrent service of a read request and a write request, and WoW enables the concurrent service of multiple writes. Both these mechanisms are built upon two architectural techniques: (i) a technique that determines which words of a cache line write-back are essential, and (ii) a technique that enables a fine-grain write, which is a cache line write-back that only involves chips on essential words, and leaves the other chips idle (so, the main memory controller can use the idle chips for serving other read or write requests). We discuss the details of these two architectural support before describing our RoW and WoW techniques.

A. Architectural Supports for PCMap

1) Finding essential words: We need a mechanism to find out which words need to be updated in main memory (finding essential words). Equivalently, we are looking for a technique to detect redundant updates, especially silent stores [9]. To tackle this, we may take three different approaches that are discussed in the following along with their advantages and disadvantages.

1) **Extended dirty flag in cache:** At the last-level cache (LLC), we can extend dirty flag from one bit per line (e.g., 64B) to one bit per word (e.g., 8B). With this organization, every write request to the cache is preceded by a read to detect the word(s) that have been changed. This approach requires modification to LLC, and can increase the write traffic to it. In addition, in cases where the LLC is made up of DRAM (i.e., a common choice in studies on PCM main memories [10]–[12] and we also used in our system set-up), read-before-write could be more expensive because it requires changing the bus direction (bus reads to bus writes or vice versa).
2) **Read-before-write at memory controller:** We can rely on read-before-write issued by the memory controller. In this approach, the memory controller first reads the old version of a cache line from the PCM DIMM, and then finds the modified word(s) based on the difference between the updated and old versions of the cache line. This approach does not require any modification to LLC and can be designed to minimize the number changes in bus direction, but can increase memory traffic.
3) **Read-before-write on PCM chip:** One could assume read-before-write happens inside the PCM chips. This is a reasonable assumption for PCM write optimization. In fact, the PCM sub-array already reads the cache line first and only writes into the flipped bits [3], [13]. For chips with non-modified words, this takes only the read latency. For other chips, however, this includes write latency as well. As a result, different chips might experience different write service times. If each chip can notify the memory controller when it is done serving a write, then the memory controller can send a new write/read job and utilize the idle chips. However, this approach increases the number of wires from the PCM-based DIMMs to the memory controller. Another option would be to set a flag in the register located on DIMM indicating the completion of writes. The memory controller could poll over these registers in different chips of all DIMMs to find those with completed writes.

In this paper, PCMap employs the third approach (*Read-before-write on PCM chip*) where the completion of writes on PCM chips is stored in a DIMM register, and the controller pulls them up for finding idle chips and performing intelligent scheduling. The third approach is preferred because it does not increase LLC complexity and its load (like the first approach), and does not increase main memory traffic (like the second). More discussion on the DIMM register and commands that the controller issues to check for idleness of chips are given in Section 4.4.

2) Fine-grained writes: After finding the essential words, we need a mechanism to perform *fine-grained* writes (i.e., writing each word separately). This can be done at the DIMM register by sending separate RAS/CAS commands to different chips in order to isolate the update of a chip (these per-chip writes can take place in different rows). Our mechanism is similar to *rank subsetting* [14] with slight modifications in the data layout. We begin by describing the conventional rank subsetting architectures, and then explain the design of a single DIMM in PCMap.

Different forms of rank subsetting have been introduced in recent years [14]–[16] to improve the energy consumption and performance of a rank. Rank subsetting partitions a 64-bit rank into smaller sub-ranks (e.g., 8 sub-ranks) where each can be independently controlled with the same single command/address bus on the channel. A RAS command only applies to one sub-rank at any time, i.e., it only activates a subset of the PCM chips in the rank, and limits the amount of data brought into the row-buffer. Figure 3 shows the concept of rank subsetting in a DIMM made of eight $\times 8$ chips. Figure 3.(a) shows how two cache lines, A and B, each having 8 words ($A0$ to $A7$, $B0$ to $B7$), are mapped

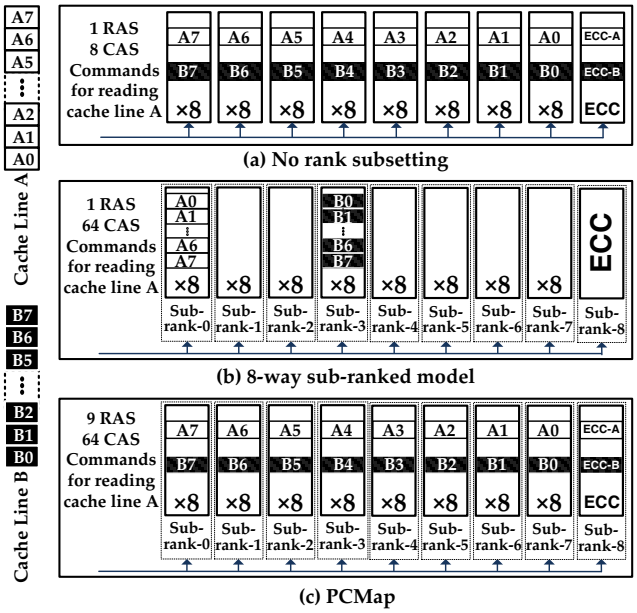

Figure 3: Data layout in (a) a DIMM memory with no rank subsetting, (b) conventional 8-way sub-ranked model and (c) the rank subsetting and data layout in a DIMM in PCMap.

to different chips in the baseline that has no rank subsetting (it has only one rank): words $A0$ ($B0$) to CHIP0, word $A1$ ($B1$) to CHIP1 and so on. Figure 3.(b) depicts mapping of the same cache lines in a 8-way subranked DIMM (i.e., our assumption in this paper where each subrank corresponds to one chip), using the model given in [16]: words of cache line A are only mapped to CHIP0 (sub-rank0) and cache line B is completely stored in CHIP1 (sub-rank1). By rank subsetting, chips in different sub-ranks can be addressed independently, meaning that the number of active cache lines increases. This leads to higher rank-level parallelism, which in turn brings shorter queuing delays and higher data bus utilization [15], [16]. The drawback is the need for additional CAS commands to fetch a cache line, e.g., the number CAS commands are 8 times higher in 8-way subranked model compared to a non-subranked DIMM (8 and 64 CAS commands needed in Figures 3.(a) and 3.(b), respectively). This increases the access latency of a cache line.

Our PCMap architecture uses the rank subsetting scheme introduced by Ahn et al. [16]. This specific implementation is DDR3 compliant, and uses multiplexers and a buffer chip on the DIMM board to activate an appropriate sub-rank on every command. Figure 3.(c) depicts the data layout for cache lines A and B in our memory model, which is exactly the same as baseline ($A0$/$B0$ are mapped to CHIP0, $A1$/$B1$ are mapped to CHIP1, and so on). The difference from the baseline is that, with sub-ranked DIMM, each word can be accessed independent of the others, and more importantly, it has the capability to serve words of different cache lines simultaneously (they should be mapped to different chips);

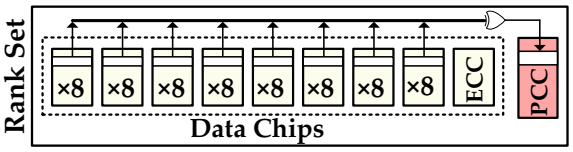

Figure 4: ECC-DIMM organization with PCC chip.

e.g., writing word $A0$ to CHIP0 and writing word $B1$ to CHIP1. With this memory architecture, (i) we can implement *fine-grained* writes, i.e., update only essential words of a cache line and leaving the uninvolved chips free, and (ii) we can concurrently access words from different cache lines residing in the same bank.

The problem with fine-grained write is that we need to send separate RAS commands to control each chip (word or sub-rank). For example, accessing all words of the cache line A in Figure 3.(c) needs 8 RAS and 64 CAS commands. Note that we do *not* need this fine-grained access on read operations; so, serving a read request is done via a *coarse-grain access* (one rank of 8 chips like in conventional DIMMs). This reconfigurable access mechanism, i.e., supporting both the *coarse- and fine-grained accesses*, can be simply provided by updating the busy status of the chips in the DIMM register [16] (discussed in Section 4.4).

B. RoW (Read over Write)

Recall from Section III that a majority of writes need to update only few chips (words). In PCMap, these updates are performed by the fine-grained write scheme, as described above (Section IV-A2). When a write is on-going, some chips are busy and the rest are idle (we talk about the ECC chip later). On the other hand, reads to main memory are at a cache line granularity (i.e., 64B), and partially fetching a cache line from the uninvolved chips during a write might not be helpful since these partially fetched data cannot be sent back till the write is done and the rest of the cache line (missing in the first step) is read. The basic idea behind RoW is that, from the read queue perspective, these chips are not available as if they are faulty (similar to the concept of faulty chips in Chipkill [17], [18]). If PCM ranks are augmented with a mechanism correcting one chip failure, it might be possible to service a read request by using this correction mechanism to *reconstruct* the data in busy chips. Depending on the number of busy chips and the strength of the correction mechanism used, we might be able to serve a read in parallel with writes.

We use a simple error correction scheme; our correction code needs one extra chip and the focus is on cases where *only one chip is involved in a write*. The assumed architecture is a rank with ten $\times 8$ PCM chips, as shown in Figure 4. In addition to having the ECC chip (ninth chip) for SECDED protection, we employ a tenth $\times 8$ chip, called PCC (Parity Correction Code) from now on, to *reconstruct* the missing bits of the read request assigned to the write-involved chip. Data reconstruction from the PCC code is

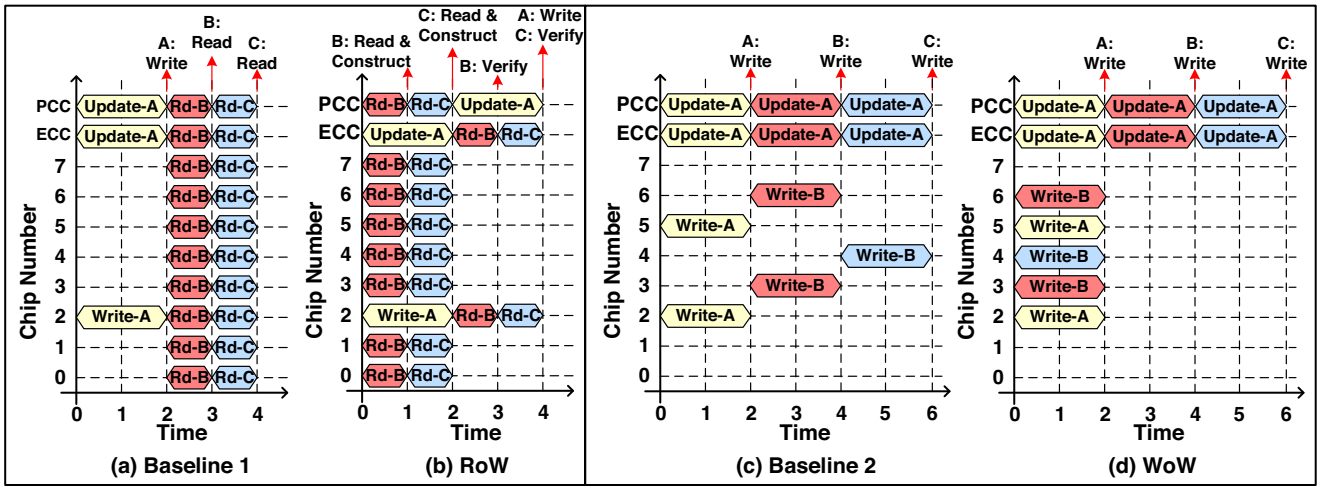

Figure 5: (a)–(b) Example for RoW. (c)–(d) Example for WoW. We assumed that the write latency is twice the read latency.

trivial, i.e., a simple XOR of all other data chips, because the controller already knows which chip is busy with a write (or equally faulty) and a simple correction scheme suffices. This approach can be used to reconstruct the missing data stored in the write-involved chip. The only limitation of this scheme is that we have to restrict the service of a write to one chip at a time, if we decide to concurrently service a read from the same bank as well. Note that we cannot use the SECDED ECC chip as a replacement for PCC in RoW, because RoW needs correction of a whole word missing on read, while SECDED only corrects one bit failure. As a result, it is mandatory to have an extra mechanism (one extra PCC chip that adds a storage overhead of 11.11%).

One of the key points is that with RoW, it is possible to service multiple reads (if we have in read queue) in parallel with one write. The reason is that servicing a write takes 2–8 times longer than a read. This creates the potential of servicing more than one read (though sequentially) in parallel with servicing a write.

1) Updating ECC and PCC chips: In the assumed memory architecture, writing into (at least) one chip also requires updating both the SECDED ECC and PCC chips. This defeats our purpose. In order to use the RoW mechanism in this situation, we *delay* the write to only the PCC chips, and commit it later as new chip update. When the controller decides to schedule for RoW, the original ongoing cache line write (with only one essential word in this case) is broken into two fine-grained writes running in serial: (*Step 1*) writing the data chip (i.e., for essential word) and the SECDED ECC chip, and (*Step 2*) writing to the PCC chip. Along with each partial writes, the controller can schedule one or more reads. We note that in order to avoid the need for data recovery and simplify the controller design, the memory controller must schedule the second step of a write right after finishing the first step (with no interrupt). The PCMap controller considers this when handling a request.

2) Example of RoW: Figure 5 illustrates how RoW operates with an example. Figure 5.(a) shows the baseline scheduling, where cache line A is written and followed by reading cache lines B and C. Figure 5.(b) depicts how RoW works: it breaks writing cache line A in a series of chip-level updates, and concurrently reads cache lines B and C (one at a time) with the update of CHIP3 and ECC chips for A. After reading B (except from CHIP3) as well as the PCC data, the controller *reconstructs* the missing data bits of B (Word 3). The same is done for reading and reconstructing C. Later, the controller schedules another write to update A's PCC data. With RoW, it is clear that reading cache lines B and C is not stalled by long write latency of cache line A.

3) Recovering failures on data read: Since we do not simultaneously read all data and error correction bits, SECDED is not available when performing RoW to correct potential one-bit errors on reads – this can affect memory reliability. To address this issue with RoW, the controller passes the read cache line to the CPU and we allow the CPU to progress with the currently read data by assuming that the data is fault-free. Later, when the missing word is read (e.g., word 3 of B and C in Figure 5), the controller checks the correctness of data. If the initially read data has errors, the controller corrects it using SECDED ECC and resends it to the processor for fault-free execution. This process can be costly if the read data was previously used by the CPU for processing. However, our experiments on a wide spectrum programs shows that 98.7% of all cache lines read by RoW are *not* committed by the processor before the data correctness check at the memory controller. Consequently, we do *not* rollback. In case of faulty execution of the program (i.e., 1.3% of all read lines given by RoW), we rollback to the point where the read data is processed by the CPU. Later we discuss the cost of this rollback process, and show that even though we need rollbacks as high as 5.8% of all data reads, we never see performance drop in a

system with RoW compared to the baseline.

4) Employing RoW for writes with more than one chip updated: Utilizing RoW has a constraint: it works only when the on-going write access has only one essential word. But, as shown by Figure 2, in 75% of write accesses, more than one chip update is necessary and, more specifically, in 25% of all writes, two chips need to be updated. To utilize RoW in such scenarios, the controller may decide breaking the write access into two (or more) partial writes (each should necessarily update one essential word). To conclude, RoW can be scaled to writes with more than one essential word, but doing so may increase the access time of a cache line write. In our experiments in this paper, we only employed RoW for writes with only one chip update in order to (1) keep the write latency at a reasonable bound and (2) reduce the complexity of scheduler.

C. WoW (Write over Write)

Our fine-grained write approach (Section IV-A2) narrows down the cache line write to a few chip updates (holding the essential words). With this in place, the remaining chips will be free to serve another write request(s). We can concurrently serve two (or more) write requests from the same bank, if they have *non-overlapping* sets of modified words (involved PCM chips). This is the basic idea behind WoW. The main objective of WoW is to enhance the PCM write bandwidth such that, in a latency equal to a PCM write, PCMap serves as many cache line writes as possible.

1) Example of WoW: Figure 5 further illustrates the application of WoW with an example: three write requests are targeting one bank. The first write request (A) updates Word 2 and Word 5; the second one (B) updates Word 3 and Word 6 and the third (C) updates only Word 4. Figure 5.(c) shows the baseline where all writes are serialized. Figure 5.(d) shows one possible application of WoW where all writes can be consolidated and performed at the same time because they do not contend for the same chips. Note that, because of the contention at ECC and PCC chips, the updates to ECC data for all the requests should still be serialized. This problem (updating ECC and PCC chips) can limit the write bandwidth improvement, as discussed later.

2) Chip-level conflicts in WoW: WoW concurrently serves multiple writes, if their chip updates have *non-overlapping* sets of modified words. However, it is possible that the essential words get clustered in the same chips in multiple write-backs, limiting the potential of parallelizing one write with another. To reduce such chip contentions, we *rotate* the allocation of words of a cache line to the chips. The rotation is based on the cache line address – if we assume that beginning address of a cache line is `Address`, the module calculation determines the rotation offset. More precisely, with eight data chips in a rank and cache line size of `L`, `Address` modulo (8 × `L`) gives the rotation offset. Figure 6 shows with this address-based rotation scheme, how the same word offset in successive cache lines in the memory would be mapped to different chips (the cache line X is stored with no offset, the cache line $X + 1$ is stored with one offset, and so on). This address-based rotation scheme has two advantages. First, when accessing a cache line, the memory controller easily calculates its rotation offset by having its address. So, it *does not* need to keep any book-keeping information with each cache line. Another premise (which we observed in our workloads to be 32% on average) is that there is higher a likelihood of write-backs with successive addresses being dirty at the same offsets in their respective words.

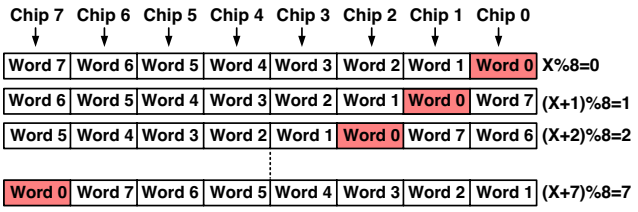

Figure 6: An example of rotating 8 consecutive cache lines (the address of the first cache line is X) over data chips.

Another potential source of contention in WoW is the update of the ECC and PCC chips on every write. That is, if all the error detection/correction data are mapped to the same chip, successive write requests will always contend for that chip, limiting the potential of WoW because the updates of ECC and PCC chips have to be serialized. We consequently propose *rotating the ECC and PCC bits* across the chips to address this problem. More precisely, in a PCMap DIMM with ten chips in a rank (eight data chips, one ECC chip and one PCC chip), the rotation offset is calculated as `Address` modulo (10 × `L`), where `Address` is the address of the cache line and `L` is the LLC line size.

By rotating the ECC and PCC chips along with data chips, the updates are not concentrated to few chips and are better balanced. Hence, although PCMap and wear-leveling schemes (e.g., Start-and-Gap [5]) are generally orthogonal, PCMap is expected to have better lifetime than the baseline.

D. PCMap Memory Architecture

PCMap is designed upon two mechanisms: *rank subsetting* and *finding busy and idle chips*. Here, we discuss a typical implementation of a PCMap DIMM.

1) Implementation of PCMap: Figure 7 shows an exemplary DIMM memory channel along with one rank and its memory controller in our PCMap. As we previously described with *rank subsetting* in PCMap, each rank is divided into multiple sub-ranks, each sub-rank with only one chip. Physically, the same data bus as in a conventional memory channel is used (it has 80-bit width[4]), but it is now divided into ten logic data buses, each occupying one tenth of the

[4]We expect that the module standards for DIMMs which incorporate PCMap would include some number of pins for PCC chip.

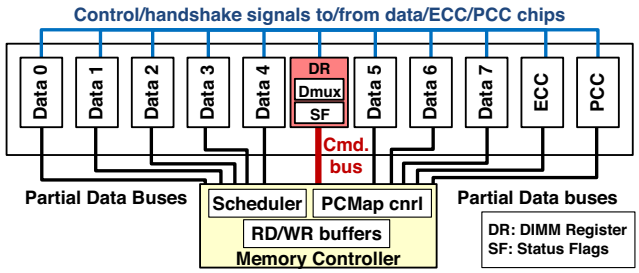

Figure 7: A typical implementation of PCMap DIMM.

physical bus. We have a register on DIMM (DIMM register) on each rank, which has two main functionalities: (1) it has a demultiplexer register which routes (demultiplexes) control signals to the proper chip to provide independent operations. Also, rank subsetting necessitates extra wires on DIMM board to connect the DIMM register to each chip independently (shown in Figure 7). (2) For each *bank* in the rank, it keeps one *status register* that shows busy/idle status of the chips when writing a cache block (one bit per each chip). If the chip is busy with writing a word (essential word), after comparing the old and new data at PCM chip (similar to [3])., the chip itself informs the DIMM register to set the status bit to "1" (busy); otherwise it should be "0". Later, the controller reads the status register by sending the `Status` command to understand which chips are busy for future scheduling. It takes 2 cycles ($0.8ns$) for the memory controller to send `Status` command and receive the status register's content for each bank. Also, setting/resetting the status register, which is performed by PCM chip, takes a variable time because of variable write latency and the needs for read before write operations at PCM chip.

2) Request Scheduling at PCMap Controller: Having the possibility of performing RoW and WoW in main memory, the scheduler in our PCMap tries to exploit them. Our scheduler is built on top of current PCM schedulers [11], where they always prioritize reads over writes unless write queue is $> \alpha$ full. In our memory scheduling, when a write is scheduled for service, the scheduler decides to service other requests in parallel, based on the following order:

1) If the write queue is $> \alpha$ full and the read queue is non-empty and on-going write has only one essential word, the scheduler selects the oldest read request in the read queue and performs RoW.
2) If write queue is $> \alpha$ full and on-going write has more than one essential word (regardless of having requests in the read queue), the scheduler selects one or more write requests in the write queue that can be parallelized with on-going write. The selection policy is again oldest-first.

V. Experimental Setup

We evaluated our proposed architecture, using a typical baseline multi-core system with configuration and parameters close to real systems and similar studies.

Table I: Main characteristics of the simulated system.

Processor, On-chip Caches	
Cores	8 cores, out-of-order, 2.5GHz, Solaris 10 OS
L1 caches	Split I and D, 32KB private, 2-way, 32B, LRU, write-through, 1-cycle hit, MSHR: 4 instruction & 32 data
Coherency	MOESI directory; 2×4 grid packet switched NoC; XY routing; 1-cycle router; 1-cycle link
L2 cache	8MB shared, 8-way, 64B, LRU, write-back, 7-cycle hit, MSHR: 32 (I and D)
DRAM cache	256MB shared, NUCA with 8 banks, 8-way, 64B, LRU, write-back, 100-cycle access latency, tag array is SRAM
Main Memory	
Controllers	4 controllers, 32×64B write queue, 8×64B read queue.
Memory	4 channels, 1 DIMM per channel, 1 rank per DIMM, 8 ×8 chips per rank, 8 banks per chip; PCM cell [19]: 60ns read, 50ns RESET, 120ns SET; Frequency: 400MHz, t_{RDC}=60 cycles*, t_{CL}=5 cycles, t_{WL}=4 cycles, t_{CCD}=4 cycles, t_{WTR}=4 cycles, t_{RTP}=3 cycles, t_{RP}=60 cycles, t_{RRDact}=2 cycles, t_{RRDpre}=11 cycles

* Here, *cycle* refers to memory cycle at 400MHz frequency.

Table II: Characteristics of the evaluated workloads.

Workload	RPKI	WPKI	Workload	RPKI	WPKI
8-Thread Multi-Threaded					
canneal	15.19	7.13	fluidanimate	5.54	1.51
dedup	3.04	2.072	freqmine	0.78	3.33
facesim	6.66	1.26	streamcluster	5.19	2.13
8-Application Multi-Program (MP)					
Workload				**RPKI**	**WPKI**
MP1: 2×mcf, 2×gemsFDTD, 2×astar, 2×sphin3				6.45	3.11
MP2: 2×mcf, 2×gromacs, 2×gemsFDTD, 2×h264ref×2				2.68	1.56
MP3: 2×gromacs, 2×h264ref, 2×astar, 2×sphin3×2				2.31	1.08
MP4: 8×astar				8.05	5.65
MP5: 8×gemsFDTD				4.15	2.6
MP6: 2×cactusADM, 2×soplex, 2×gemsFDTD, 2×astar				5.09	2.09

Evaluation platform: We simulate our target multi-core architecture using the Gem5 full-system simulator [20], and the main memory model is based on DRAMSim2 [21], modified for including PCM details.

Evaluated configurations: Target system is an 8-core processor with the configuration given in Table I. It has three levels of caches, with 256MB DRAM as the LLC. Main memory is SLC-based PCM. We consider four memory channels, one rank per each channel, eight ×8 chips per rank to achieve the standard 8B interface. Internally, each chip is organized into 8 banks. The target system has four on-chip memory controllers (each connected to one channel) where each controller maintains separate write and read queues (each with size of 32 and 4 entries, respectively) for banks.

We evaluate six different systems:

1) ***Baseline*** prioritizes always reads over writes, if WRQ is $< 80\%$ full.
2) ***RoW-NR*** applies RoW without rotation of words and without rotation of ECC.
3) ***WoW-NR*** uses WoW without rotation of words and without rotation of ECC.
4) ***RWoW-NR*** is a system with both RoW and WoW without rotation of words and without rotation of ECC/PCC.
5) ***RWoW-RD*** is a system with both RoW and WoW with rotation of words and without rotation of ECC/PCC.
6) ***RWoW-RDE*** is a system with both RoW and WoW with rotation of words and rotation of ECC/PCC.

Throughout our evaluation, we report the results for each system normalized to the *baseline system* explained above.

Workload characteristics: Table II lists all the workloads tested in this work, the application programs they contain, and the number of read and write requests to the PCM memory per thousand instructions (RPKI and WPKI) – the higher these numbers, the greater the memory intensity for the workload. For multi-threaded workloads, we used seven programs from PARSEC-2 [22] (in addition to the results for selected workloads, we give the average improvement for all 13 programs in PARSEC-2; it is shown with label Average(MT) for the results in Section 6). To construct multi-programmed workloads, we chose programs from SPEC CPU 2006 [23] that put enough pressure on the main memory in the simulated multicore system (Table I). All our benchmarks are compiled with full optimization and run with the large-sized input. Each workload is run on the simulated multicore system for one billion instructions, after 200 millions for cache warm-up phase.

VI. Experimental Results

A. Impact on IRLP

Recall that the primary objective of the WoW mechanism is to enhance intra-rank-level parallelism (IRLP). Figure 8 plots the changes in IRLP with different configurations and rotation policies. We drive three main conclusions from this figure. First, for the multi-threaded workloads, IRLP in baseline is very low (less than 2, on average), and increases to about 3.5 when using WoW and the rotation policies (boosts to 6 for applications like `canneal` in RWoW-RDE, i.e., rotating data and ECC words). Second, the IRLP values in the multi-programmed workloads are slightly better than those of the multi-threaded workloads in the baseline, and also significantly increases with the WoW mechanism and rotation policy (it gets close to the maximum value, 8, in `MP1`, `MP2` and `MP3` workloads). Third, when using rotation, the likelihood of conflicts on few words/chips is reduced, which in turn leads to higher IRLP. Also, rotating the ECC word is more beneficial than just rotating the data words since the ECC chip has to be updated on every write request and is always a source of conflict, if not rotated.

B. Impact on the write throughput

One of the key performance metrics for a PCM memory is its write throughput, i.e., the number of write requests completed in a given time unit. Figure 9 plots the results of write throughput in different systems, normalized to the baseline. For all the workloads tested, the write throughput is enhanced when employing the WoW mechanism; for 5 out of 12 workloads, the improvement is above 1.2× over the baseline, and is still considerable for the majority of the workloads (¿10% improvement). Note that these results are directly related to enhancements in IRLP. For instance, since the RWoW-RDE system gives the maximum IRLP among other systems (by rotating data and ECC/PCC words over all chips), it also delivers the maximum write

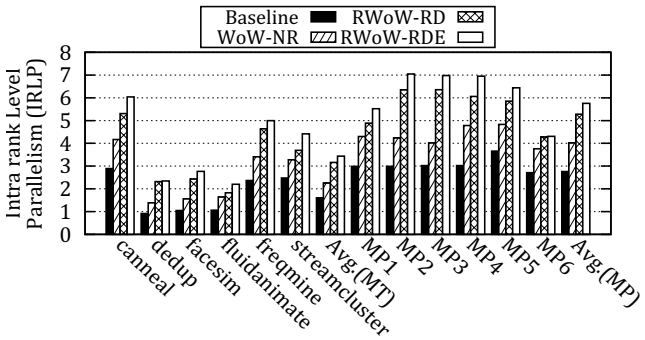

Figure 8: Comparative analysis of IRLP.

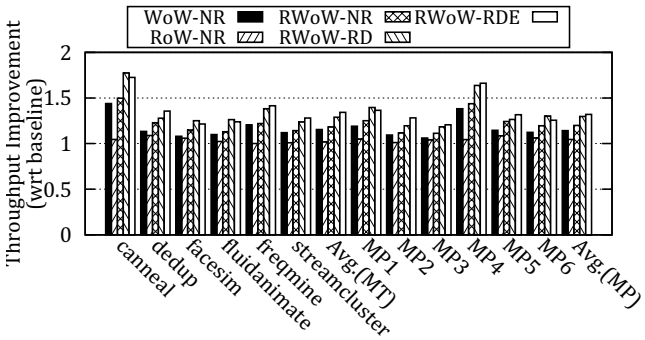

Figure 9: Analysis of write throughput enhancement.

throughput. Further, the combination of RoW+WoW (i.e., RWoW) is highly effective, and achieves an average of 33% improvement in the write throughput (compared to the results of RWoW-NR and WoW-NR). Thus, our proposal considerably improves the low write throughput in PCM.

C. Impact on the effective read latency

Figure 10 compares the effective read latency (i.e., the latency to complete a read operation) in our five configurations. Again all the results are normalized to the baseline. We can see that, by removing the possible contentions between reads and writes (with only one essential word), (i) the system with only RoW policy (RoW-NR) reduces the effective read latency by 6-14%, (ii) WoW also helps in reducing the read latency by parallelizing write requests and reducing the average latency penalty that writes impose on reads, and (iii) the latency penalty due to writes is further reduced by enhancing the IRLP in systems with the capability of rotating the data and ECC chips. Thus, our proposed RWoW-RDE (RoW+WoW with the capability of rotating the data chips and the ECC chip) policy reduces the average effective read latency by about 50% and 55% for multi-threaded and multi-programmed workloads, respectively.

D. Impact on the System Performance

Figure 11 plots the IPC improvement brought by the tested configurations over the baseline system. For all the workloads, the improvements in read latency and write

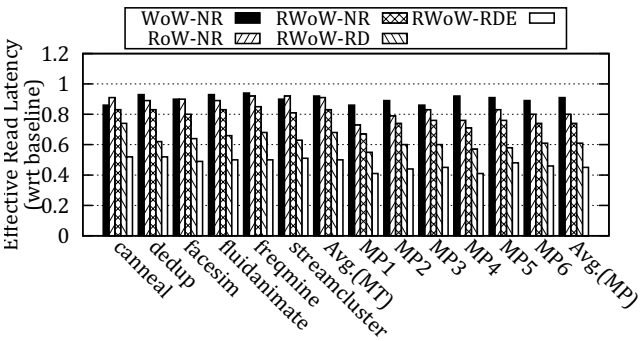

Figure 10: Analysis of the effective read latency.

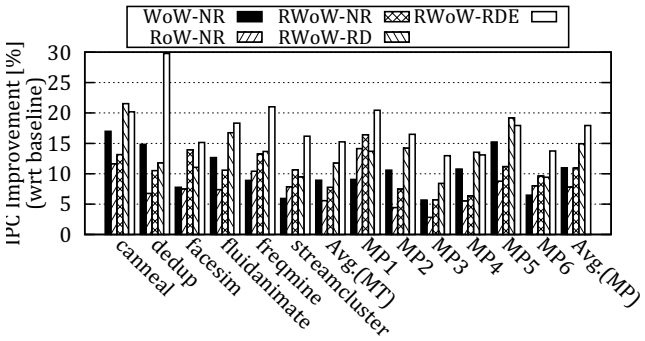

Figure 11: IPC improvement over the baseline.

throughput translate to significant IPC improvements. On average, there is a 6.1% improvement with WoW-NR over the baseline. RWoW-NR gives 9.95% improvement, while RoW-NR, RWoW-RD, and RWoW-RDE achieve, respectively, IPC improvements of 4.5%, 13.1%, 16.6%, considering both the multi-threaded and multi-programmed workloads together.

E. Impact of Write-to-Read Latency Ratio

In our evaluation so far, we assumed that a write incurs twice the latency of a read. We now analyze the sensitivity of our proposed scheme by varying the read latency. For this analysis, we assume a constant write latency of $120ns$, and variable read latency. Table III lists the IPC improvements for RWoW-NR and RWoW-RDE, as the latency ratio between write and read is varied from 2 to 8. RWoW-RDE improves performance by 18.7% and the upper bound is 29.7%, and exhibits only slight changes when the read-to-write latency ratio varies. For RWoW-NR performance improvement greatly depends on the write-to-read latency ratio, increasing from 11.3% at 2× to 24.7% at 8×.

Table III: Effect of write-to-read latency.

Write-to-read latency	2×	4×	6×	8×
RWoW-RDE	16.6%	18.7%	21.1	24.3%
RWoW-NR	11.3%	13.8%	18.8%	24.7%

F. Cost of Rollback for Data Correction in RoW

As we described earlier, to guarantee data correctness on reads in RoW mechanism, the processor may need to rollback if the initial read data was faulty. Rollback is not always necessary, but it may happen frequently, depending on the application nature. Table IV shows the IPC improvement given by PCMap with respect to the baseline (configuration in Table I) for five programs that experienced the maximum CPU rollbacks (among 12 programs tested) after applying RoW. This table gives two sorts of IPC improvement given by PCMap: the first is for a system that assumes data is always faulty and rollback is necessary when the initial readout data has been processed by the CPU; and the second assume that data is always non-faulty and rollback never happens. From this figure, one can deduce that: (1) RoW always leads to performance improvement, even though we need rollbacks as high as 5.8% of all data reads (i.e., for `canneal`), and (2) the cost of rollback (calculated as difference between IPC improvement in none-faulty and faulty systems) is variable and can be as high as 4.6%.

Table IV: IPC of RoW normalized to the baseline.

Workload	canneal	facesim	MP6	ferret
% of Max. Rollbacks	5.8%	4.1%	3.4%	2.2%
IPC Imp. in faulty system [%]	12.18%	1.1%	8.75%	12.37%
IPC Imp. in none-faulty system [%]	14.87%	4.29%	10.02%	16.8%

VII. Related Work

Write latency improvement. The *pre-set* architecture [24] alleviates the problem of slow writes by exploiting the fundamental property of PCM cells which indicates writes are slow only in one direction (SET) and are almost as fast as reads in the other direction (RESET). Therefore, a write operation to a line in which all memory cells have been SET prior to the write, will incur much lower latency. Similarly, Yue and Zhu [25] proposed to divide a write into stages: in the write-0 stage all zeros are written at an accelerated speed, and in the write-1 stage, all ones are written with increased parallelism, without violating power constraints. Qureshi et al. modeled MLC PCM access, and proposed *write cancellation* and *write pausing* [11] to let read operations preempt long-latency MLC writes and enhance the effective read latency of MLCs. Jiang et al. proposed *write truncation* [26] to identify the MLC cell that requires more iterations to write, and truncate their last several iterations to enhance write latency. To cover the erroneous data of truncated writes, they used an extra ECC code. Sampson et al. [27] proposed two mechanisms, one of which enables applications to approximately store data in PCM, leading to performance improvement. Also, their proposal allows errors in multilevel cells by reducing the number of programming pulses used to write them.

Write bandwidth improvement. A few prior studies [2], [10] explored hybrid DRAM/PCM main memory systems to address the problem of limited write bandwidth of PCM by directing most of the write traffic to DRAM. Du et al. [28] proposed a compression scheme (applied at last level cache)

to reduce the amount of data that should be updated on a PCM write. Zhou et al. [29] developed a non-blocking PCM bank design where subsequent reads or writes can be carried out in parallel with an on-going write. Xie et al. [30] explored the possibility of removing unmodified and joining the modified data in multiple writes to form a single write request, in order to improve the write bandwidth in PCM. The main difference between this system and PCMap is that PCMap consolidates multiple writes and reads proactively while this system tries to find the consolidation of writes.

VIII. Conclusion

Although PCM has many desirable features to replace DRAM as the main memory, its long write latency is a major concern that can affect application performance. In this paper, we propose two concurrent read and write mechanisms along with an ongoing write, called RoW and WoW, that allow subsequent read or write to the idle chips of the PCM rank accessed for a long write operation. We propose novel mechanisms such as re-constructing data from ECC bits to serve a read from an idle chip, and rotating word mapping and error detection/correction bits across PCM chips of a rank to enable concurrent read with a write and write with a write to improve PCM access parallelism. The proposed PCM access parallelism (PCMap) system is shown to improve the intra-rank-level parallelism (IRLP) from 2.37 to 4.5 (up to 7.4) across a wide range of workloads, and these parallelism improvements translate to average IPC improvements of 15.6% and 16.7% for the multi-programmed and mutli-threaded workloads, respectively.

Acknowledgment

This work is supported in part by NSF grants 1526750, 1302557, 1213052, 1439021, and 1302225 and a grant from Intel.

References

[1] C. Lefurgy *et al.*, "Energy management for commercial servers," *Computer*, vol. 36, no. 12, pp. 39–48, 2003.

[2] B. Lee *et al.*, "Architecting phase change memory as a scalable DRAM alternative," in *ISCA*, 2009, pp. 2–13.

[3] P. Zhou *et al.*, "A durable and energy efficient main memory using phase change memory technology," in *ISCA*, 2009, pp. 14–23.

[4] S. Raoux *et al.*, "Phase-change random access memory: A scalable technology," *IBM Journal of Research and Development*, vol. 52, no. 4.5, pp. 465–479, 2008.

[5] M. Qureshi *et al.*, "Enhancing lifetime and security of PCM-based main memory with start-gap wear leveling," in *MICRO*, 2009, pp. 14–23.

[6] Micron Technology Inc., "Micron DDR3 SDRAM Part MT41K1G8SN-107," http://www.micron.com/parts/dram/ddr3-sdram/mt41k1g8sn-107, 2015.

[7] Y. Choi *et al.*, "A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth," in *ISSCC*, 2012, pp. 46–48.

[8] S. Chen *et al.*, "Rethinking database algorithms for phase change memory," in *CIDR*, 2011, pp. 21–31.

[9] K. Lepak and M. Lipasti, "Silent stores for free," in *MICRO*, 2000, pp. 22–31.

[10] M. Qureshi *et al.*, "Scalable high performance main memory system using phase-change memory technology," in *ISCA*, 2009, pp. 24–33.

[11] M. Qureshi *et al.*, "Improving read performance of phase change memories via write cancellation and write pausing," in *HPCA*, 2010, pp. 1–11.

[12] W. Zhang and T. Li, "Characterizing and mitigating the impact of process variations on phase change based memory systems," in *MICRO*, 2009, pp. 2–13.

[13] L. Jiang *et al.*, "FPB: fine-grained power budgeting to improve write throughput of multi-level cell phase change memory," in *MICRO*, 2012, pp. 1–12.

[14] A. Udipi *et al.*, "Rethinking DRAM design and organization for energy-constrained multi-cores," in *ISCA*, 2010, pp. 175–186.

[15] H. Zheng *et al.*, "Mini-rank: Adaptive DRAM architecture for improving memory power efficiency," in *MICRO*, 2008, pp. 210–221.

[16] J. Ahn *et al.*, "Future scaling of processor-memory interfaces," in *SC*, 2009, pp. 1–12.

[17] X. Jian *et al.*, "Low-power, low-storage-overhead chipkill correct via multi-line error correction," in *SC*, 2013, pp. 24:1–24:12.

[18] T. J. Dell, "A white paper on the benefits of chipkill-correct ECC for PC server main memory," 1997.

[19] D.-H. Kang *et al.*, "One- dimensional heat conduction model for an electrical phase change random access memory device with an 8F2 memory cell (F=0.15um)," *Applied Physics*, vol. 94, no. 5, pp. 3536–3542, 2003.

[20] N. Binkert *et al.*, "The Gem5 simulator," *Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[21] P. Rosenfeld *et al.*, "DRAMSim2: a cycle accurate memory system simulator," *Computer Architecture Letter*, vol. 10, no. 1, pp. 16–19, 2011.

[22] C. Bienia and K. Li, "PARSEC 2.0: a new benchmark suite for chip-multiprocessors," in *MoBS*, 2009.

[23] C. D. Spradling, "SPEC CPU2006 benchmark tools," *Computer Architecture News*, vol. 35, no. 1, pp. 130–134, 2007.

[24] M. Qureshi *et al.*, "PreSET: improving performance of phase change memories by exploiting asymmetry in write times," in *ISCA*, 2012, pp. 380–391.

[25] J. Yue and Y. Zhu, "Accelerating write by exploiting PCM asymmetries," in *HPCA*, 2013, pp. 282–293.

[26] L. Jiang *et al.*, "Improving write operations in MLC phase change memory," in *HPCA*, 2012, pp. 1–10.

[27] A. Sampson *et al.*, "Approximate storage in solid-state memories," in *MICRO*, 2013, pp. 25–36.

[28] Y. Du, *et al.*, "Delta-compressed caching for overcoming the write bandwidth limitation of hybrid main memory," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 55:1–55:20, 2013.

[29] P. Zhou *et al.*, "Throughput enhancement for phase change memories," *IEEE Transactions on Computers*, vol. 63, no. 8, pp. 2080–2093, Aug. 2014.

[30] F. Xia *et al.*, "DWC: dynamic write consolidation for phase change memory systems," in *ICS*, 2014, pp. 211–220.