Fine-Grained QoS Scheduling for PCM-based Main Memory Systems

Ping Zhou† Yu Du‡ Youtao Zhang‡ Jun Yang†

† {piz7, juy9}@pitt.edu
Electrical and Computer Engineering Department
University of Pittsburgh
Pittsburgh, PA 15261

‡ {fisherdu, zhangyt}@cs.pitt.edu
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260

***Abstract*—With wide adoption of chip multiprocessors (CMPs) in modern computers, there is an increasing demand for large capacity main memory systems. The emerging PCM (Phase Change Memory) technology has unique power and scalability advantages and is regarded as a promising candidate among new memory technologies. When scheduling a mix of applications of different priority levels, it is often important to provide tunable QoS (Quality-of-Service) for the applications with high priority. However due to the slow PCM cell access, and the destructive interferences among concurrent applications, existing memory scheduling schemes lack the flexibility to tune QoS in a wide range, in particular to the level close or equal to that of stand-alone execution. In this paper we propose a novel QoS scheduling scheme that utilizes *request preemption* and *row buffer partition* that enable QoS tuning at a fine-granularity. That is, they can tune the request queuing time and the PCM bank service time for the high priority requests. Our experimental results show that the proposed scheme achieves 1.7× ∼10× QoS tuning range while introducing negligible area and energy overheads.**

I. Introduction

The past decade has envisioned the wide adoption of chip multiprocessors (CMPs) in modern computer systems. In such systems, the main memory is a critical shared resource for serving multiple threads/applications running concurrently. The applications, ranging from high performance scientific computations, to streaming video processing, to data intensive data mining, often have large memory capacity requirements but different runtime memory access behaviors. The memory footprint of modern applications is usually large e.g. a database application requires large main memory to hold the index file for efficient query processing; and a web application requires large main memory to cache its recent accessed files. However the memory access behaviors are quite different at runtime e.g. a computation intensive application often has bursty memory accesses and low overall memory bandwidth requirement. Its performance is sensitive to the average processing latency of its memory accesses. In contrast, a memory intensive application has continuous accesses during the execution. Its performance is more sensitive to its acquired memory bandwidth at runtime.

Under the pressure to meet the aggregated capacity requirement of multiple applications, the main memory in future CMP systems has to scale large. As the technology enters the nanoscale regime, large DRAM-based main memory faces serious leakage power and cost limitations. Lefurgy *et al.* reported about 40% of power was consumed by the main memory system on a mid-range IBM eServer machine [5]. Due to these issues, several new memory technologies are drawing more attention in recent years, such as Phase Change Memory (PCM) and Spin-Transfer Torque RAM (STT-RAM) [4], [16], [28]. PCM is particularly promising due to its better write endurance over NAND flash, and better cost effectiveness over STT-RAM.

While non-volatile PCM-based main memory has many advantages, its memory accesses are slower than those of DRAM, which worsens the interferences among concurrently running applications. In particular, it greatly degrades the QoS tuning ability when scheduling a mix of applications of different priority levels. For example, requests from a critical application i.e. the one with the highest priority level, may arrive at the memory controller but find the bank is busy servicing requests from other applications. Due to long PCM access time, the critical application may suffer from large performance loss even if each of its requests can get serviced after finishing the current one in the bank. Existing memory scheduling schemes [12]–[14] focus on fairness and overall throughput when scheduling requests from multiple applications. They were designed for DRAM systems that have different physical characteristics, and lack the ability to tune the scheduling to meet the QoS requirement in a PCM based main memory system.

In this paper, we propose a novel memory scheduling scheme for achieving high performance and fairness on PCM-based main memory systems. In particular we target at broadening the QoS tuning range of critical applications. By identifying two major performance degradation sources — (1) increased request queuing time; and (2) increased row buffer misses, we propose *access preemption* and *row buffer partition* to enable QoS tuning at fine granularity. The non-volatility of PCM allows us preempting a being served request without destroying the row contents. Preempting a PCM read has no impact on PCM cells. While preempting a PCM write may leave corresponding cells in undetermined states, the raw data is buffered in the memory controller and thus can be resumed later. In addition, we employ the multi-entry row buffer organization for PCM based main memory [4], and dynamically partition the entries into two groups to service the critical and other applications respectively. Our experimental results show that the proposed scheme achieves 1.7× ∼10× QoS range improvement without significant area and energy

978-1-4244-6443-2/10/$26.00 ©2010 IEEE

overheads.

For the rest of the paper, we discuss the PCM basics, the structure of PCM based main memory, and its integration in a 3D stacked processor in Section II. We elaborate the design details of *access preemption* and *row buffer partition* in Section III. The experimental settings are discussed in Section IV with the results presented in Section V. More related work is discussed in Section VI. Section VII concludes the paper.

II. PCM BASED MAIN MEMORY

A. PCM Basics

Phase-change memory (PCM) is one type of non-volatile memory that exploits the unique behavior of chalcogenide alloy to store information. As shown in Figure 1, a PCM cell usually consists of a thin layer of chalcogenide adhered by two electrodes from each side. The chalcogenide such as $Ge_2Sb_2Te_5$ (GST) has two stable states, crystalline and amorphous, and can switch between them with the application of heat i.e. by injecting current into the PCM cell.

There are two write operations associated with GST cell. The `SET` operation gets GST heated above the crystallization temperature ($\sim 300^o$C) but below the melting temperature ($\sim 600^o$C) over a period of time. This turns the GST into the low-resistance crystalline state that corresponds to the logic '1'; the `RESET` operations gets GST heated above the melting temperature and quenched quickly. This places the GST in the high resistance amorphous state that corresponds to the logic '0'.

Due to repeated heat stress applied to the phase change material, a PCM cell has limited number of write cycles. Referred as *write endurance*, it is a key parameter in designing PCM-based memory systems. A typical SLC PCM cell can sustain $10^8 \sim 10^9$ writes before a failure can occur [1], [3], [27]. This is better than the write endurance of NAND flash i.e. 10^5 times, and worse than that of a DRAM cell i.e. 10^{15} times. To extend the life time of PCM based main memory system with limited cell write endurance, Zhou *et al.* proposed circuit innovations [28] to update only modified bits when a cacheline is written back to the PCM memory; Qureshi *et al.* [16] utilized a DRAM buffer to filter out frequent write operations; Lee *et al.* [4] proposed the multi-entry narrow row buffer design to extend the life time. Wear leveling techniques were also proposed to balance the number of write times across different cachelines, blocks, and segments in the main memory [28].

Reading the bit from a PCM cell involves sensing the resistance level of the cell. It is non-destructive and has negligible heat stress comparing to write operations. With technology advances, it is now possible to differentiate more than two resistance levels and thus store multiple bits in a PCM cell. This is referred as MLC (multi-level cell). The cell that stores one bit only is referred as SLC (single-level cell).

B. The Memory Banks and Memory Controller

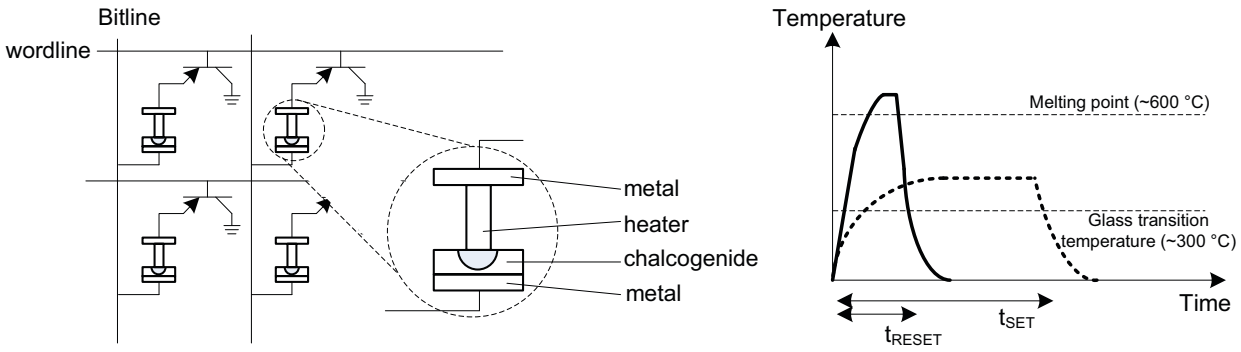
PCM cell array structure is organized similar to that used by DRAM (Figure 1(a)). It has peripheral logic such as decoders, sense amplifiers, and write drivers. The cells are grouped into sub-blocks, blocks, and banks in a similar way as DRAM.

Figure 2 illustrates a 3D stacked CMP architecture with PCM-based on-chip main memory. Our design is based upon this architecture while it can be extended to offchip PCM main memory as well. The 3D processor consists of a core layer that is close to the heat sink, an interface face layer, and several layers of PCM memory banks. Comparing to other stacking alternatives, the memory-on-core stacking is simple and is getting popular in recent years [2], [6], [25]. The core layer employs a crossbar interconnect to connect four cores together. It is placed close to the heat sink for better heat dissipation. The memory controller is integrated on the interface layer. We adopt the hybrid PCM/DRAM organization and assume a DRAM buffer below the last level cache. In this paper we consider 4 PCM memory layers (i.e. chips) stacked on top of the interface layer. The data are interleaved into four layers to support parallel access, which corresponds to one channel, one rank organization that supports 8 simultaneous PCM accesses. Memory banks utilize the multi-entry narrow row buffer design to exploit access locality and achieve better energy and performance efficiency [4].

To serve a memory request, the memory controller sends a sequence of *micro* commands to the memory banks. For DRAM chips, in the case if a memory access cannot be served by the row buffer, a *precharge* command needs to be issued to write back a row buffer entry such that the new row can be loaded [11], [13]. That is, a read access may need to wait for the completion of a write back of the dirty row and a read of the new row. This is not desired for PCM based memory since PCM write is slow, the *precharge* command shall considerably prolong the read access and hurt the performance. Instead PCM write requests bypass the row buffer and write to the cells directly, as shown in [28]. A potential concern of this design is that consecutive writes are not merged in the row buffer and thus increase the number of write operations. It is not a problem in our organization as such writes have been captured by the DRAM buffer below the last level cache.

The memory accesses, according to if they can be served by the row buffer, can be categorized into the following types.

- *Read row-hit*. If the read request can be served using an entry in the row buffer, then there is no need to access the memory bank. The memory controller only need to send a *read* command to the lower level bank. The access latency is the row buffer latency $\mathtt{T}_{rh}$.
- *Read row-miss*. If the request cannot be served using any entry in the row buffer, then the memory controller needs to issue an *activate* command to move the data to the empty row and then a *read* command to get the data. The latency contains both the cell read access latency $\mathtt{T}_{cr}$ and the row buffer access latency $\mathtt{T}_{rh}$.
- *Normal write*. For a write operation, the memory controller issue the *write* command directly and send the data to the bank through data bus. We adopt the *redundant write removal* technique in [28]. If there are modified bits



(a) PCM cell. (b) PCM SET and RESET pulses.

Figure 1. The PCM cell and its SET/RESET pulses.

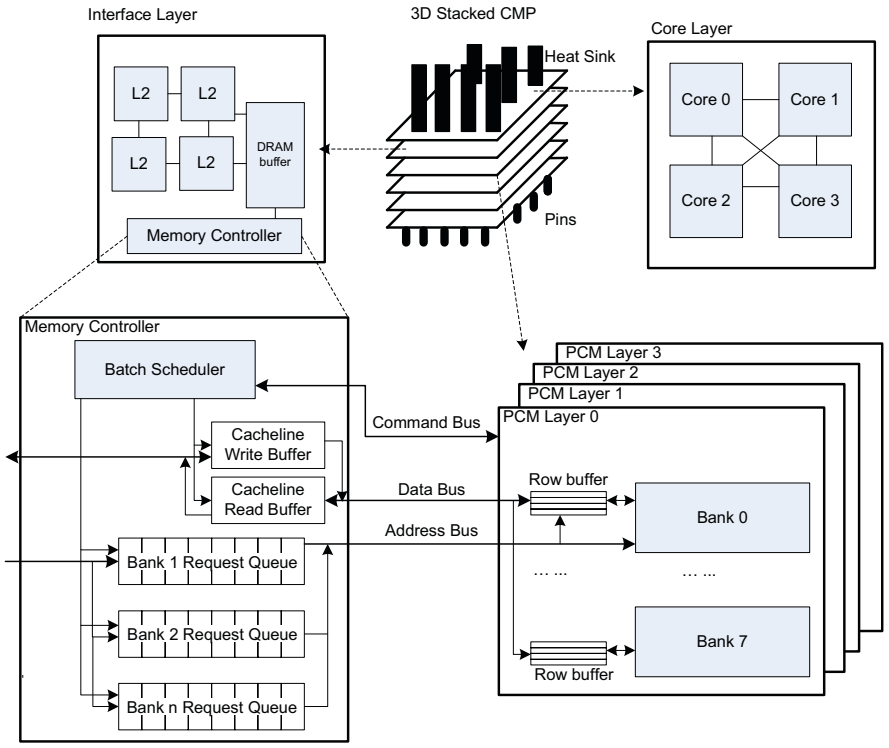


Figure 2. A 3D stacked processor with on-chip PCM based main memory.

to be written, then the write operation does not complete until these bits are successfully written to the cells. The latency is the the cell write latency $\mathtt{T}_{cw}$.

- *Redundant write*. For a write operation, if all bits are the same, the write is redundant and thus can be terminated early. The actually latency is $\mathtt{T}_{cr}$. We assume $\mathtt{T}_{cw}$ to achieve uniform write interface and simplify the memory controller design. It has energy advantages.

The multi-entry row buffer in our design is organized as a fully associative cache with tags stored in the memory controller i.e. on the interface layer. We exploit LRU replacement policy to manage the entries in the row buffer. For a hit in the row buffer, the corresponding entry index is sent to bank to fetch the desired data. In the case if it is a miss, then the index of the selected entry to be replaced is also sent to the bank.

C. Fair/QoS Memory Scheduling

As a shared system resource, the main memory needs to serve memory requests from multiple applications on different cores. If the memory controller receives multiple requests asking for the data from the same bank, it needs to determine a fair order such that the overall system performance is sustained while no application gets severely penalized.

Since our scheme is motivated by *parallelism-aware batch scheduling* (PAR-BS) [12], we briefly summarize PAR-BS as follows. PAR-BS groups a number of DRAM requests into a batch and ensures that all requests from the current

batch are serviced before the next batch is formed. For the requests within one batch, they are ordered according to their thread IDs such that requests from one thread are likely serviced in parallel. By grouping DRAM requests into batches and servicing them in order, PAR-BS achieves fairness and exploits the thread-level parallelism. PAR-BS has simple priority scheduling tuning ability. When scheduling a mix of applications with different priority levels, PAR-BS enhances the scheduling in two ways: (1) Requests from a higher priority application are marked more often e.g. they are marked for every batch while the others are marked every other batch; (2) Requests from a higher priority application are scheduled before other requests within one batch.

III. Fine-Grained QoS Scheduling

A modern CMP computer system often executes a mix of applications of different QoS requirements. Among these application, one (or several) is of more importance e.g. a user may be editing an image file while several background applications are extracting emails, and/or downloading software patches etc. For the application of the highest priority, we usually prefer its performance to be the same or close to that when running the application alone on the CMP. However the long memory latency of PCM accesses can significantly degrade the QoS tuning ability when applying existing memory scheduling schemes.

To study the memory slowdown due to interferences, Figure 3 compares the average read latency breakdowns of critical applications when running alone and running with other applications (details of the mixes are summarized in Table III). We chose average read latency as it is an good indicator of the performance impact from the memory system — the write operations are not on the critical path due to the large DRAM buffer below the last level cache. Each application mix has four applications while one or two applications are assigned to be of the highest priority. We collected the results based on the 3D processor architecture shown in Figure 2 and adopted PAR-BS scheduling with priority levels. More configuration details can be found in Section IV.

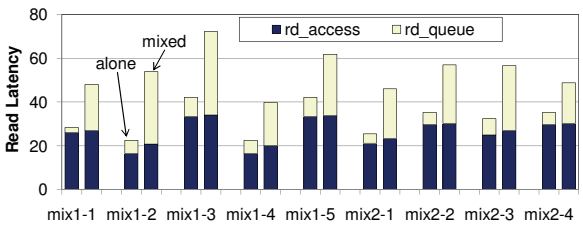


Figure 3. Read latency breakdown of critical applications.

As we can see from the figure, the slowdown comes mainly from the prolonged queuing — the requests from critical applications spend more time waiting in the request queue. This is because the PCM banks have more requests to handle and accessing PCM is slow. In addition the critical requests have more row buffer misses and on average spend 6% more time in accessing PCM banks.

A. Request Preemption

Based on the above observation, our first design targets at reducing the queuing time of critical requests. Since the priority-based batch scheduling still needs to complete the current batch before the next batch is formed, a critical request, when it arrives at the memory controller, may experience a long queuing time. If this request can preempt the current being serviced requests, then the queuing time could be greatly reduced. We only perform read preemption as they are the requests on the critical path.

Batch preemption. A simple preemption is to insert incoming critical requests into the current batch i.e. the current batch can dynamically accept grow and mark more requests. Within the batch, the incoming requests are scheduled behind those with the same or higher priority levels. If no such request exists, then the incoming requests are serviced right after the completion of the current request.

Batch preemption may starve low priority requests if they are blindly scheduled after the critical requests, and there are continuous incoming critical requests. This is similar to the starvation problem in network-fair-queuing [14]. We adopt the start time queuing mechanism [18] to regulate the critical requests to utilize their pre-assigned bandwidth share.

Batch preemption without regulation resembles the *opportunistic* scheduling in [12] i.e. low priority requests are serviced only if there is available bandwidth.

Cell-access preemption. Even with batch preemption, critical requests still need to wait for the completion of the current PCM access even if the batch has no other high priority request. Given PCM accesses are slow, in particular PCM writes are 2.5× slower than PCM reads, and 10× slower than row buffer hits, we still observed large performance losses.

To further reduce the request queuing time, we exploit the non-volatility of PCM cells and allow a high priority request preempting the current low priority PCM access. That is, the memory controller sends a *preempt* command to stop the ongoing PCM access, and then an *activate/write* command starts a new one. As a comparison, it is impossible to preempt a DRAM access due to its destructive read — DRAM read destroys the contents of a whole row. Preempting a DRAM write is also problematic as it first reads a row into the row buffer and then updates. If a DRAM read or write is preempted, then the old contents might be lost.

The current access can be one of row buffer hit, row buffer read miss (i.e. PCM cell read), or PCM write operations. The memory controller never preempts a row buffer hit due to its low latency. Since PCM reads are non-destructive, preempting a PCM read leaves no change to the PCM cells. The preempted read can be restarted at a later time. Preempting a PCM write is a bit complicated. With the *redundant bits removal* design [28], a write operation includes a read of the cells for comparison. The cells are unchanged if preemption happens before real write operation starts, and left in undetermined

states if preemption happens when the cells are being modified. In our current design, we adopted the pessimistic assumption — the cell states are undetermined, and the memory controller needs to resend the cacheline to restart.

While it is safe to preempt a PCM read or write at any time, it is not always a better decision, in particular, preempting a PCM access that is close to finish introduces both performance and energy overheads. Another drawback is, a write operation may perform several times before its success, which reduces the write endurance of affected cells. In our design, we set a threshold `T` and disable preemption if the current access can finish in `T` cycles. A preemption threshold helps to tune the trade-off between overheads and performance impacts to the critical applications.

B. Partitioning Row Buffers for Priority

While preemption helps to reduce the long queuing time of critical applications, the out-of-order scheduling of requests from multiple applications destroys the row buffer locality and results in more row buffer misses. Figure 4 illustrates an increased impact from row buffer misses. Using preemption, there is a 9% bank access time increase for the critical applications. The increase was 6% using the priority-based batch scheduling.

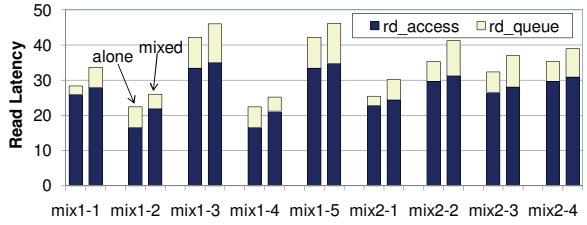


Figure 4. Read latency breakdown of critical applications with preemption.

To provide fine-grained QoS tuning ability, we propose to control row buffer misses by dynamically partitioning the row buffer entries among concurrent applications. Our design is motivated by the utility based partitioning for caches [15]. We divide the application into two groups: the prioritized group (PG) that contains critical applications; and the normal group (NG) that contains all other applications. Each row buffer entry belongs to one group at a time while the number of entries allocated to each group is dynamically determined by evaluating their utility of the entries.

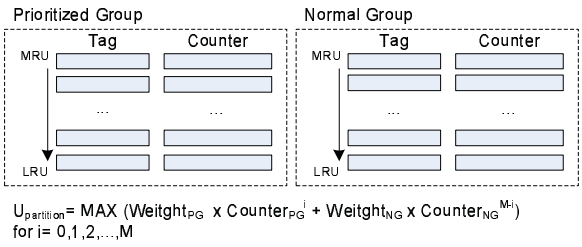


Figure 5. Partitioning the row buffer entries based on their utility.

Figure 5 illustrates the monitoring logic in the memory controller. For an M-entry row buffer, each group maintains an M-entry tag array and an M-entry counter array. Both are ordered according to their recency positions i.e. from MRU (most recent used) to LRU (least recent used). The hit counters count the number of hits at each position for each group at runtime, and thus provide the utility information when different number of entries are allocated to each group. Due to the inclusive property of LRU replacement policy, that is, an access hit in a 2-entry row buffer is also a hit in a 3-entry row buffer, we can evaluate all possible partitions and choose the one that gives the maximal benefits. For the example in Figure 5, there are M+1 possible cases: one group gets 0,1,2, ..., and M entries respectively while the other group has all the rest entries. We divide the execution into epochs and periodically adjust the partition for the next epoch based on the utility of the past epoch.

IV. Experiment Methodology

A. Settings

We evaluated our proposed scheme through trace-driven and execution-driven simulations. Detailed traces are collected from a Simics-based simulator [9] and processed with load-store queue to eliminate read-after-write dependencies. We used an in-house trace-driven simulator to evaluate the MCPI (memory cycles per instruction), average read latency and unfairness indexes of different schemes. A GEMS/Simics-based execution-driven simulator [7] was then used to evaluate the average IPC (instruction per cycle). The settings are summarized in Table I. We modeled a 3D stacked processor (as shown in Figure 2) with 4 cores. Each core is 4-issue, out-of-order with private L1 and L2 caches. The cores and L1 caches are integrated on the core layer. The interface layer integrates L2 caches, the memory controller as shown in [16]. The main memory consists of four PCM layers and is shared by all cores. The PCM memory is organized into 8 banks (i.e. up to 8 requests can operate in parallel).

We assembled a set of applications with different memory bandwidth requirements from SPEC SPEC2006 [21], SPLASH2 [26], and STREAM [10] benchmark suites. For each application, we skipped its warm-up phase, and simulated 100 million instructions for memory access and IPC studies. Table II lists the characteristics of the simulated phase for each application. They are categorized into two groups based on MCPI (memory cycles per instruction): the memory non-intensive applications that include bzip2, fmm, and raytrace, and the memory intensive applications that include others. Memory intensive applications have much higher MCPI than memory non-intensive ones.

We then mixed these applications to construct a set of multiprogramming workloads to evaluate scheduling effectiveness. As shown in Table III, each mix contains 4 applications. Applications are picked based on their intensiveness (MCPI). Three groups of mixes are formed. For each `mix1-*` workload, there is one critical application while for each `mix2-*` mix,

Components	Parameters
Processor Core	4 cores, each core is 4-issue, out-of-order, run at 1GHz
L1 Cache	32K I-cache and 32K D-cache, 64B cacheline size, 4-way set associative, 3-cycle cache hit latency
L2 Cache	512KB per core, 64B cacheline size, 16-way set associative, 6-cycle hit latency
Main Memory	3D stacked, 4 PCM layers, 8 banks/layer, 2GB memory in total, T_{rh}=10ns, T_{cr}=30ns, T_{cw}=100ns

Table I
THE BASELINE CONFIGURATION.

Benchmark	MCPI	Type	Comment
bzip2	0.085	Int	SPEC2006, non memory-intensive
fmm	0.297	FP	SPLASH2, non memory-intensive
raytrace	0.044	Int	SPLASH2, non memory-intensive
gemsfdtd	1.190	FP	SPEC2006, memory-intensive
lbm	3.618	FP	SPEC2006, memory-intensive
leslie3d	1.281	FP	SPEC2006, memory-intensive
mcf	3.927	Int	SPEC2006, memory-intensive
milc	9.816	FP	SPEC2006, memory-intensive
stream	2.534	Int	STREAM, memory-intensive

Table II
BENCHMARK CHARACTERISTICS.

there are two critical applications. The critical applications are in bold font in the figure. While critical programs are usually memory non-intensive in practical, we also evaluated the cases in which critical applications are also memory-intensive (`mix3-*`).

Mix	Applications (critical ones are in bold font)
mix1-1	mcf, milc, stream, **raytrace**
mix1-2	mcf, milc, stream, **bzip2**
mix1-3	mcf, milc, stream, **fmm**
mix1-4	lbm, leslie3d, gemsfdtd, **bzip2**
mix1-5	lbm, leslie3d, gemsfdtd, **fmm**
mix2-1	milc, lbm, **raytrace, bzip2**
mix2-2	lbm, milc, **fmm, raytrace**
mix2-3	lbm, milc, **fmm, bzip2**
mix2-4	gemsfdtd, milc, **fmm, raytrace**
mix3-1	mcf, stream, gemsfdtd, **leslie3d**
mix3-2	mcf, milc, stream, **gemsfdtd**

Table III
WORKLOAD MIXES.

B. Evaluation

To evaluate the effectiveness of our proposed scheduling scheme, we implemented and compared the following three approaches in our experiments.

- `PAR-BS (baseline)`. This implements the PAR-BS scheduling scheme [12] without priority levels (i.e. all applications have same priority).
- `PAR-BS/P`. This implements the PAR-BS scheduling with priority levels. We assign the highest priority level to the critical application(s) such that within each batch, the requests from critical programs are serviced before those from other applications.
- `BatchP`. This scheduling scheme allows the insertion of critical requests into the current batch. While a fair queuing mechanism is needed to avoid starvation, in evaluating this scheme, we removed regulation to resemble the *opportunistic* scheduling [12].
- `FullP`. This scheduling scheme implements batch preemption, cell-access preemption, and row-buffer partitioning. We restricted the number of times a request can be preempted to avoid excessive impact on non-critical applications.

We focused on evaluating the read access latency as it determines the memory impact on application performance. With the DRAM buffer between the last level cache and the PCM memory, the write operations are not on the critical path. The metrics that we used are similar to those in evaluating DRAM systems [12], [13].

$$Memory\_SlowDown = \frac{MCPI_i^{shared}}{MCPI_i^{alone}}$$

$$Weighted\_Speedup = \sum_i \frac{IPC_i^{shared}}{IPC_i^{alone}}$$

$$Hmean\_Speedup = \frac{NumApplications}{\sum_i \frac{1}{IPC_i^{shared}/IPC_i^{alone}}}$$

V. EXPERIMENTAL RESULTS

A. Parameters

There are two important parameters in our scheme that can be used to tune the slowdown of critical applications. The first parameter is `preemption threshold T`. In cell-access preemption, current PCM access cannot be preempted if it will finish in T cycles. Using a larger T results in less preemption while using a smaller T results in more aggressive preemption. In particular, if T = 0, then the critical read can always preempt; if T = 100 (latency of write requests), then there is no preemption. Note that instead of using "remaining percentage", we use "remaining cycles" instead for more accurate control. The reason is that PCM read and write have different latencies, and therefore same remaining percentage means different remaining cycles for read and write.

The second parameter is `row buffer partition weight ratio W`. This is the weight ratio between prioritized group and normal group, that is:

$$W = \frac{Weight\ of\ the\ Priority\ Group}{Weight\ of\ the\ Normal\ Group}.$$

By using these parameters, we can control the slowdown of critical application in fine granularity (details of analysis are presented in Section V-E and Section V-F). We also use these analysis to decide the optimal settings (T=10, W=4 for mix1-* and W=2 for mix2-*) which are used in rest of our experiments.

B. Tunable QoS Range

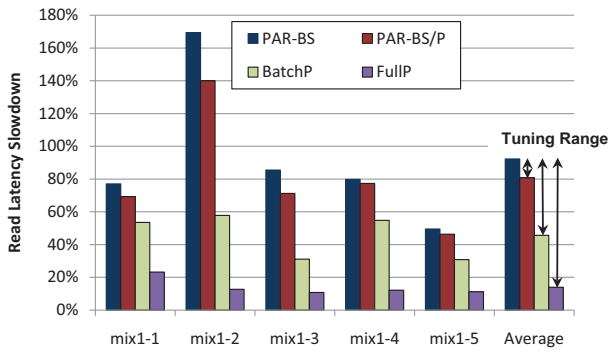
We first studied the tunable QoS ranges when using different memory scheduling schemes. Since memory read is more critical to application's performance, we use *increase of read access latency (read slowdown)* as the metrics for tunable QoS range. In other words, our goal is to achieve a broader tunable range of critical application's read latency increase. In Figure 6(a), we set the baseline by using the PAR-BS without priority, and observed an average 92% increase of read latency for the critical application. When the critical application is assigned with the highest priority, `PAR-BS/P` reduces its slowdown to 81% on average. That is, `PAR-BS/P` has a tunable QoS range [81%, 92%]. `BatchP`, and `FullP` reduce the average slowdown to 46% and 14%, and achieve tunable QoS ranges [46%, 92%] and [14%, 92%] respectively. The tunable QoS range of `FullP` is 7× and 1.7× that of `PAR-BS/P` and `BatchP` respectively.

When there are two critical applications, `PAR-BS/P`, `BatchP`, and `FullP` have tunable QoS ranges [67%,72%], [48%,72%], and [20%,72%] respectively. `FullP`'s range is 10× and 2.2× that of `PAR-BS/P` and `BatchP` respectively. Using `FullP`, the critical applications still have performance losses comparing to their stand-alone execution. The reason is that there are non-removed interferences from other applications e.g. a close-to-finish PCM request cannot be preempted due to energy and overall performance efficiency considerations.

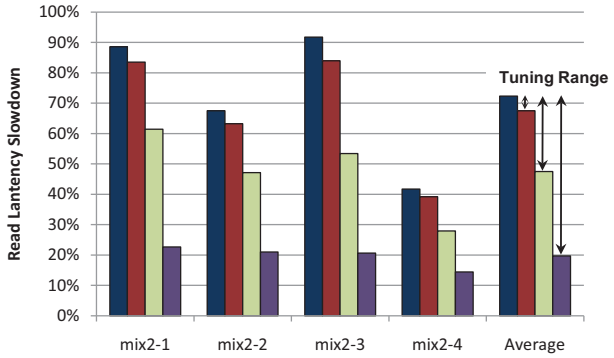
In summary, by providing a wider QoS tunable range, our scheme enables better system control for the critical programs running in multiprogramming CMP environment.

C. Fairness and Throughput

Since the main memory is shared, improving the performance of critical applications tends to slowdown other applications. Figure 7 presents the average read latencies for critical and non-critical applications respectively. Our scheme successfully reduces the read latency of critical applications without incurring large overhead to other applications. For example, when there is one critical application, the average read latency of the critical application reduces 35% while that of the non-critical applications increases around 4%. This is because the critical applications are not memory intensive

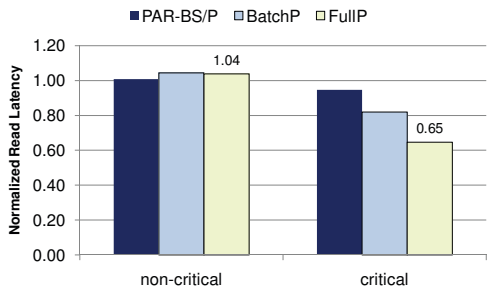


(a) Mix1-* workloads.

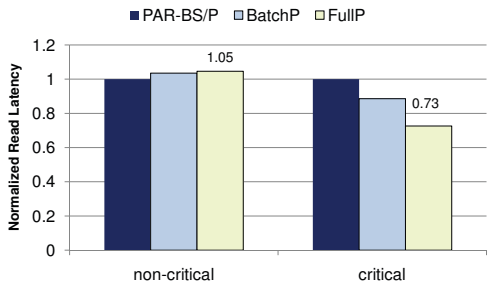


(b) Mix2-* workloads.

Figure 6. Tunable ranges of read latency increase (for critical programs).

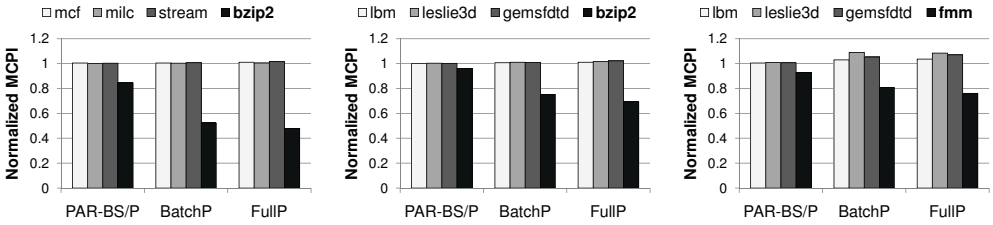


(a) Mix1-* workloads.

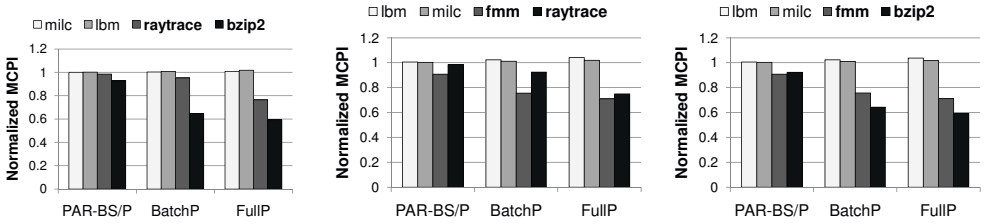


(b) Mix2-* workloads.

Figure 7. Average read latency of critical and non-critical applications.

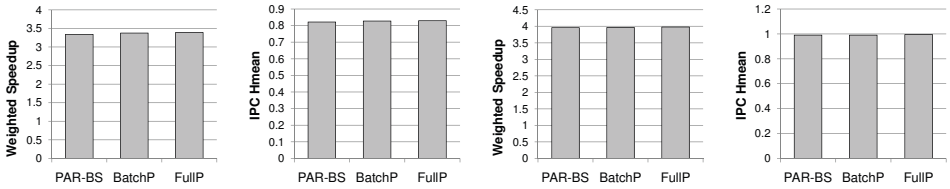


(a) Mix1-2 workload. (b) Mix1-4 workload. (c) Mix1-5 workload.



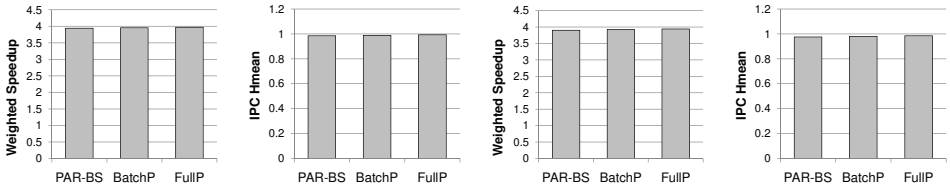
(d) Mix2-1 workload. (e) Mix2-2 workload. (f) Mix2-3 workload.

Figure 8. Comparing normalized MCPI using different scheduling schemes.



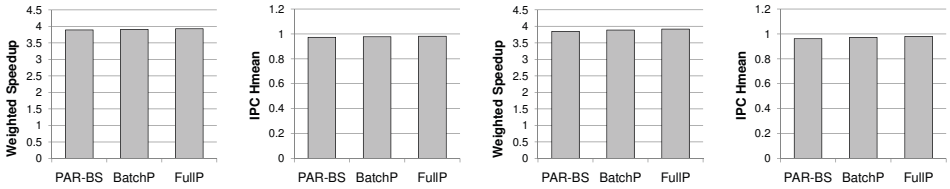
Weighted speedup Average IPC Weighted speedup Average IPC

(a) Mix1-2 workload. (b) Mix1-4 workload.



Weighted speedup Average IPC Weighted speedup Average IPC

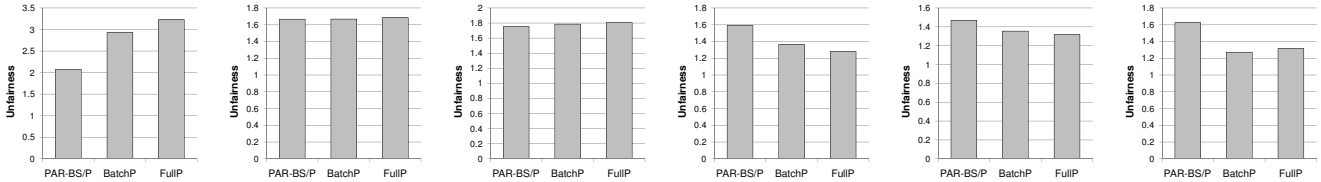
(c) Mix1-5 workload. (d) Mix2-1 workload.



Weighted speedup Average IPC Weighted speedup Average IPC

(e) Mix2-2 workload. (f) Mix2-3 workload.

Figure 9. Comparing weighted throughput and IPC using different scheduling schemes.



(a) Mix1-2. (b) Mix1-4. (c) Mix1-5. (d) Mix2-1. (e) Mix2-2. (f) Mix2-3.

Figure 10. Comparing unfairness using different scheduling schemes.

in the mixes. Our scheduling scheme did not allocate more bandwidth than its preassigned share.

Figure 8, Figure 9 and Figure 10 compare the MCPI (memory cycle per instruction), weighted throughput, IPC and fairness of individual applications in selected mixes (others are similar and omitted due to space limit). We normalized the MCPI to PAR-BS without priority. Our scheme has small impacts on MCPI on non-critical applications. The loss is within 2% for mix1-* and mix2-* workloads. The changes to the overall throughputs are also small. It has a realtively large impact on fairness. For example, our fairness is worse than the PAR-BS/P due to large performance improvement of the critical application. When there are two applications, there is small fairness improvement since the two critical applications amortize the improvement from each other.

D. Prioritizing Memory-Intensive Applications

While critical programs are typically memory non-intensive, we also evaluated the cases in which critical applications are also memory-intensive for completeness (Figure 11). Due to increased number of memory requests, prioritizing memory-intensive application results in more significant impacts on non-critical applications. For example, MCPI of non-critical application in mix3-1 is increased by 13% on average.

Another note is that from `BatchP` to `FullP`, critical application's MCPI does not reduce as much as its average read latency. Taking mix3-2 as example, critical application's MCPI is reduced from 0.60 to 0.53, while its read latency is reduced from 0.77 to 0.58. The underlying reason is that in `FullP`, only read requests from critical application can do cell-access preempt, meaning that write requests from critical application are still scheduled in the same way as they are in `BatchP`. Since memory-intensive applications tend to have more write requests (e.g. 42% of memory requests are writes in `gemsfdtd`), and PCM write takes much longer time than read, the MCPI reduction from `BatchP` to `FullP` is not as significant as average read latency.

E. Preempt Threshold

As described in Section V-A, preemption threshold T is used to determine the aggressiveness of preemption. We assume one memory cycle penalty for each cell-access preemption, so smaller preemption threshold does not always result in best average latency. Therefore, the trade-off needs to be studied to decide the optimal threshold value.

Figure 12 plots the normalized read latency of critical applications using different preemption thresholds. From our experiments, we achieved best trade-off when T = 10 cycles, which was the threshold we chose in other experiments. As a comparison, if T = 100 cycles i.e. there is no preemption, the normalized read latency of the critical application in mix1-2 workload increases from 1.13 to 1.44.

F. Weight Sensitivity

The row buffer partition weight ratio W affects how row buffer is partitioned between Priority Group and Normal Group. Figure 13 shows the average read latencies of all applications and critical applications with different weight ratios. Figure 14 illustrates the row buffer hit rates for the critical application when using different weight ratios.

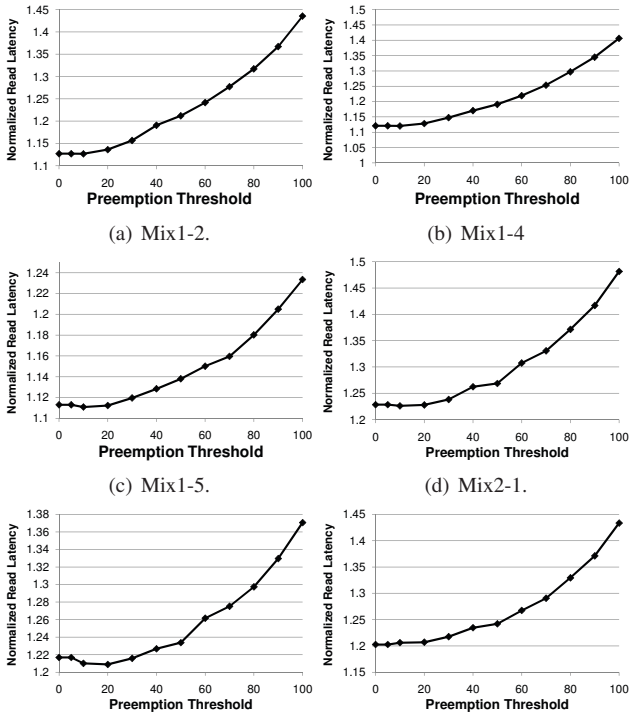


Figure 12. The normalized read latency changes for critical application(s) with different preemption thresholds.

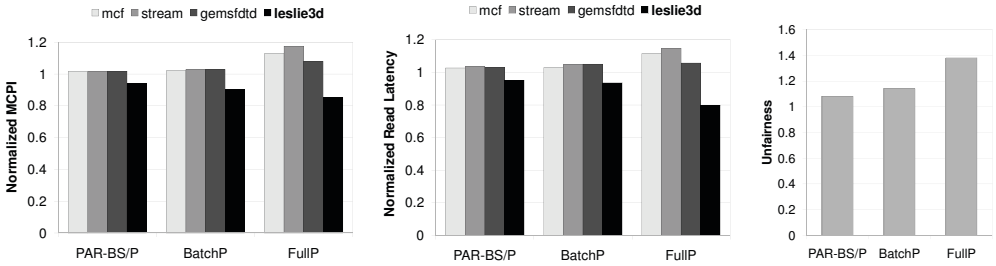
When there is one critical application i.e. for the workloads mix1-*, W=4 gives the best trade-off — the read latency averaged over all programs in each mix keeps low while the read latency of critical applications has a drop for 3 out of 5 mixes. The row buffer hit rates jump for 3 of 5 mixes. When there are two critical applications, i.e. for the workloads mix2-*, W=2 and W=4 are potential candidates. W=2 weights more on the overall system performance, while W=4 weights more on critical applications. In our experiments, we chose W=4 for mix1-* and W=2 for mix2-* workloads respectively.

G. Number of Entries

Next we studied the impact when having different number of row buffer entries. We assumed the same row buffer budget in the study i.e. for banks with the same index in four PCM layers, the total row buffer size is 2KB. If the row buffer has 4 entries, then each row is 512B, while if using 16 rows, then each row is 64B. Figure 15 shows the normalized read latency of critical applications when using different number of entries. We observed small improvements when more entries are used. In our other experiments, we used 8-entry row buffer.

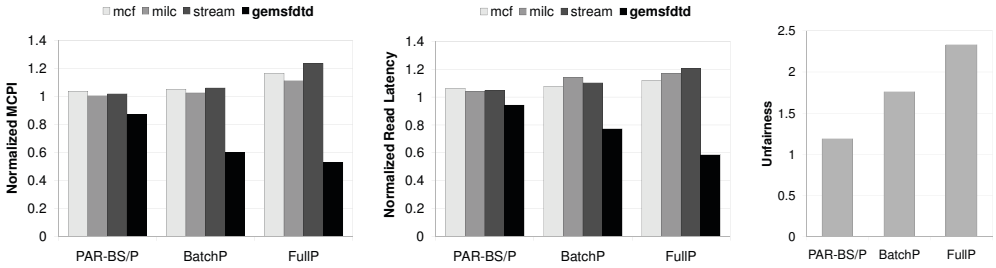
H. Overhead

Figure 16 plots the read/write access and energy overheads of our scheme. When a read or write operation was preempted,



MCPI Average read latency Unfairness

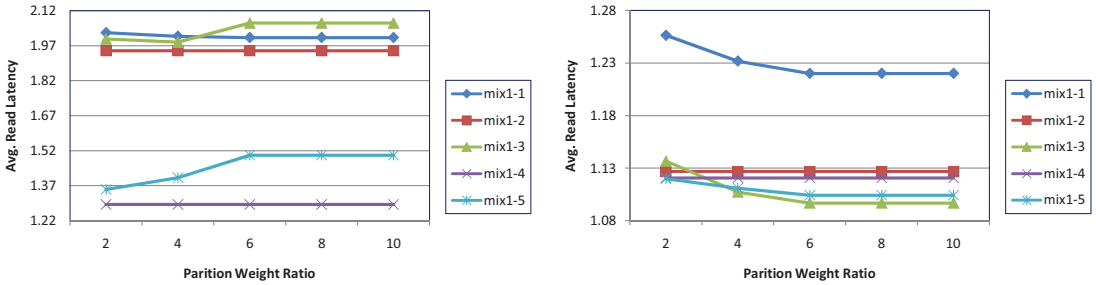
(a) Mix3-1 workload.



MCPI Average read latency Unfairness

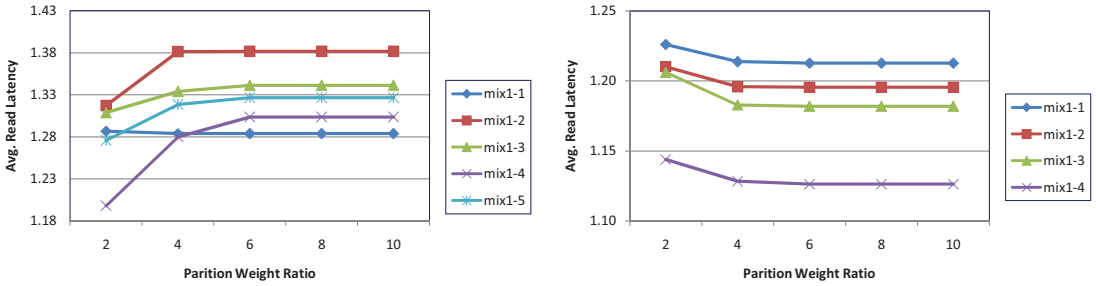
(b) Mix3-2 workload.

Figure 11. Comparing MCPI, read latency and unfairness when critical applications are also memory-intensive.



Average read latency (all applications). Average read latency (critical application).

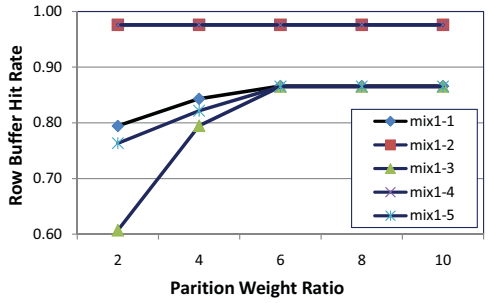
(a) Mix1-* workloads.



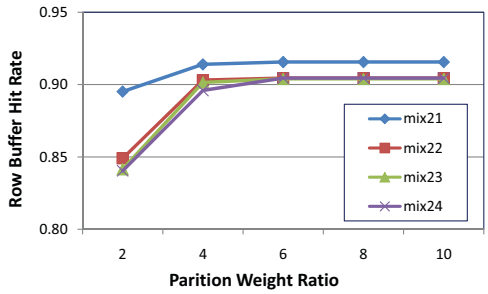
Average read latency (all applications). Average read latency (critical application).

(b) Mix2-* workloads.

Figure 13. Averaged read latency with different weight ratio W.



(a) Mix1-* workloads.



(b) Mix2-* workloads.

Figure 14. Row buffer hit rates with different weight ratio W.

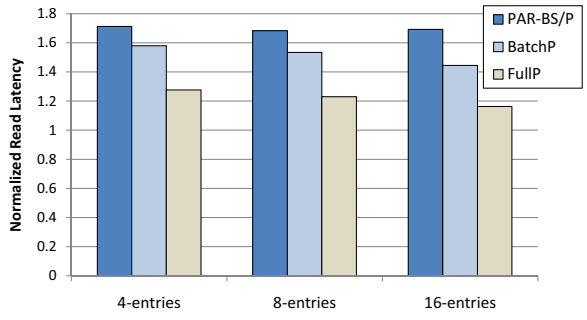


Figure 15. Number of row buffer entries.

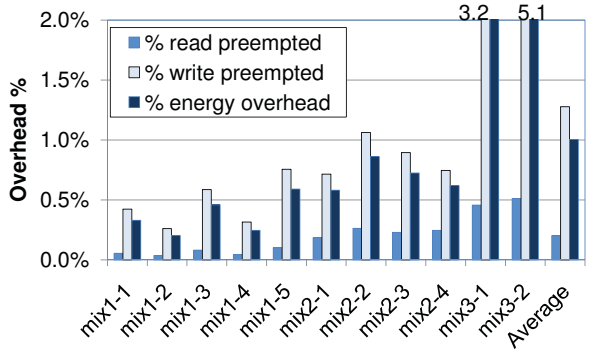


Figure 16. Preemption Overhead. Descriptions of mixes are summarized in Table III.

it has to be issued again and thus wastes the energy that it has consumed. From the figure, we preempted <1% read and write operations in most cases. This corresponds to about 1% energy increase on average. In this study we pessimistically assumed that a preempted PCM access wasted the energy to perform a full read or write.

The area overhead in our scheme comes mainly from supporting row buffer partition. We used two priority groups, and added tag and counter array for each group. Since the PCM controller can support 8 outstanding memory requests, the buffer size is 8× 2 groups × 8 entries/group × (3B/tag-entry + 4B/counter-entry) = 896B. This is negligible in modern processors.

VI. Related Work

In this section we discussed the related work in addition to those in previous sections.

Most of existing memory scheduling schemes are proposed for DRAM-based memory controllers. Among them, the simplest scheme is first-come-first-serve (FCFS). A simple enhancement to it is to schedule the requests that will get row buffer hit first, which is called first-ready FCFS (FR-FCFS) [19], [20]. Nesbit *et al.* proposed a network fair queuing (NFQ) scheme to achieve QoS and fairness in scheduling memory requests from multiple applications [14]. NFQ was based on virtual clocks that increment when requests get serviced. To solve the possible starvation problem in NFQ, Rafique *et al.* proposed the start-time fair queuing instead of the virtual finish time fair queuing [18]. Mutlu and Moscibroda proposed the stall-time fair memory (STFM) scheduling which provides fairness to different threads [13]. A more recent work, PAR-BS, was proposed by Mutlu et al. to improve overall throughput while still ensures fairness [12].

Preempting a non-volatile memory access has been studied by Sun *et al.* in [24]. They performed read-preempt-write in a cache setting with the goal to improve performance only. Qureshi et al. also proposed write cancellation and write pausing to improve PCM's read performance [17].

Our multi-entry row buffer is similar to a fully associative cache. It have been observed that partitioning a shared cache for multiple applications in a CMP based system can improve the overall system performance. Both static schemes [22] and dynamic partitioning schemes [15], [23] have been proposed.

VII. Conclusions

In this paper we proposed a novel fine-grained QoS scheduling scheme for PCM-based main memory systems. By employing batch preemption, cell-access preemption, and row buffer entry partitioning, our scheme can tune the read slowdown of critical programs in a much larger range (7x larger than that of PAR-BS/P and 1.7x larger than that of BatchP) without significantly penalizing non-critical applications in most cases. Moreover, our scheme requires simple architectural innovations and incurs little overhead to the modern processor architectures.

ACKNOWLEDGMENT

This work is supported in part by NSF under CNS-0720595, CCF-0734339, CAREER awards CNS-0747242 and CCF-0641177, and a gift from Intel Corp. The authors thank anonymous reviewers for their constructive comments.

REFERENCES

[1] "The International Technology Roadmap for Semiconductors, Process Integration, Device and Structures," http://www.itrs.net/links/2007itrs/2007_chapters/2007_PIDS.pdf, 2007.

[2] T. Kgil, S. D'Souza, A. Saidi, N. Binkert, R. Dreslinski, T. Mudge, S. Reinhardt, K. Flautner, "PicoServer: Using 3D Stacking Technology to Enable a Compact Energy Efficient Chip Multiprocessor," *The 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, 2006.

[3] D.H. Kang, *et al.*, "Two-bit Cell Operations in Diode-Switch Phase Change Memory Cells with 90nm Technology," *IEEE symposium on VLSI Technology Digest of Technical Papers*, pages 98–99, 2008.

[4] B. Lee, E. Ipek, O. Mutlu, and D. Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," *The 36th International Symposium on Computer Architecture (ISCA)*, pages 2–13, 2009.

[5] C. Lefurgy, K. Rajamani, F. Rawson, W. Felter, M. Kistler, and T.W. Keller, "Energy Management for Commercial Servers," *IEEE Computer*, Vol. 36(12):39–48, 2003.

[6] C.C. Liu, I. Ganusov, M. Burtscher, S. Tiwari, "Bridging the Processor-Memory Performance Gap with 3D IC Technology," *IEEE Design and Test of Computers,* Vol. 22(6):556–564, 2005.

[7] M.M. Martin, D. J. Sorin, B.M. Beckmann, M.R. Marty, M. Xu, A.R. Alameldeen, K.E. Moore, M.D. Hill, and D.A. Wood, "Multifacet's General Execution-driven Multiprocessor Simulator (GEMS) Toolset," *Computer Architecture News*, pages 92–99, 2005.

[8] C. Macian, S. Dharmapurikar, and J. Lockwood, "Beyond Performance: Secure and Fair Memory Management for Multiple Systems on a Chip," *IEEE International Conference on Field-Programmable Technology (FPT)*, pages 348–351, 2003.

[9] P. S. Magnusson, et al., "Simics: A full system simulation platform," *IEEE Computer,* Vol. 35(2):50–58, 2002.

[10] J.D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA)*, Newsletter, December 1995.

[11] Micron. 1Gb DDR2 SDRAM Component: MT47H128M8HQ-25, May 2007. http://download.micron.com/pdf/datasheets/dram/ddr2/1GbDDR2.pdf.

[12] O. Mutlu, and T. Moscibroda, "Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems," *The 35th International Symposium on Computer Architecture (ISCA)*, pages 63–74, 2008.

[13] O. Mutlu, and T. Moscibroda, "Stall-time Fair Memory Access Scheduling for Chip Multiprocessors," *The 40th IEEE/ACM International Symposium On Microarchitecture (MICRO)*, pages 146–160, 2007.

[14] K.J. Nesbit, N. Aggarwal, J. Laudon and J.E. Smith, "Fair queuing Memory Systems," *The 39th IEEE/ACM International Symposium On Microarchitecture (MICRO)*, pages 208–222, 2006.

[15] M.K. Qureshi, and Y.N. Patt, "Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches," *The 39th IEEE/ACM International Symposium On Microarchitecture (MICRO)*, pages 423–432 2006.

[16] M.K. Qureshi, V. Srinivasan, and J.A. Rivers, "Scalable High Performance Main Memory System Using Phase-Change Memory Technology," *The 36th International Symposium on Computer Architecture (ISCA)*, pages 24–33, 2009.

[17] M.K. Qureshi, M. Franceschini and L. Lastras, "Improving Read Performance of Phase Change Memories via Write Cancellation and Write Pausing," *The 16th International Symposium on High-Performance Computer Architecture (HPCA)*, 2010.

[18] N. Rafique, W.T. Lim, and M. Thottethodi, "Effective Management of DRAM Bandwidth in Multicore Processors," *The 16th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 245–258, 2007.

[19] S. Rixner, "Memory Controller Optimizations for Web Servers," *The 37th IEEE/ACM International Symposium On Microarchitecture (MICRO)*, pages 355–366, 2004.

[20] S. Rixner, W.J. Dally, U.J. Kapasi, P. Mattson, and J.D. Owens, "Memory Access Scheduling," *The 27th International Symposium on Computer Architecture (ISCA)*, pages 128–138, 2000.

[21] http://www.spec.org.

[22] H.S. Stone, J. Turek, and J.L. Wolf, "Optimal Partitioning of Cache Memory," *IEEE Transactions on Computers*, Vol. 41(9):1054–1068, 1992.

[23] G.E. Suh, S. Devada, and L. Rudolph, "A New Memory Monitoring Scheme for Memory Aware Scheduling and Partitioning," *The 8th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 117–128, 2002.

[24] G. Sun, X. Dong, Y. Xie, J. Li, and Y. Chen, "A Novel Architecture of the 3D Stacked MRAM L2 Cache for CMPs," *The 15th International Symposium on High-Performance Computer Architecture (HPCA)*, pages 239–249, 2009.

[25] S. Thoziyoor, J.H. Ahn, M. Monchiero, J.B. Brockman, and N.P. Jouppi, "A Comprehensive Memory Modeling Tool and Its Application to the Design and Analysis of Future Memory Hierarchies," *The 35th International Symposium on Computer Architecture (ISCA)*, pages 51–62, 2008.

[26] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta, "The SPLASH-2 Programs: Characterization and Methodological Considerations," *The 22nd International Symposium on Computer Architecture (ISCA)*, pages 24–36, 1995.

[27] F. Yeung, *et al.* "$Ge_2Sb_2Te_5$ Confined Structures and Integration of 64Mb Phase-Changed Random Access Memory," *Japanese Journal of Applied Physics*, pages 2691–2695, 2005.

[28] P. Zhou, B. Zhao, J. Yang, and Y. Zhang, "A Durable and Energy Efficient Main Memory Using Phase Change Memory Technology," *The 36th International Symposium on Computer Architecture (ISCA)*, pages 14–23, 2009.