Adaptive Burst-Writes (ABW): Memory Requests Scheduling to Reduce Write-Induced Interference

HSIANG-YUN CHENG and MARY JANE IRWIN, Pennsylvania State University
YUAN XIE, University of California, Santa Barbara

Main memory latencies have become a major performance bottleneck for chip-multiprocessors (CMPs). Since reads are on the critical path, existing memory controllers prioritize reads over writes. However, writes must be eventually processed when the write queue is full. These writes are serviced in a burst to reduce the bus turnaround delay and increase the row-buffer locality. Unfortunately, a large number of reads may suffer long queuing delay when the burst-writes are serviced. The long write latency of future nonvolatile memory will further exacerbate the long queuing delay of reads during burst-writes.

In this article, we propose a run-time mechanism, Adaptive Burst-Writes (ABW), to reduce the queuing delay of reads. Based on the row-buffer hit rate of writes and the arrival rate of reads, we dynamically control the number of writes serviced in a burst to trade off the write service time and the queuing latency of reads. For prompt adjustment, our history-based mechanism further terminates the burst-writes earlier when the row-buffer hit rate of writes in the previous burst-writes is low. As a result, our policy improves system throughput by up to 28% (average 10%) and 43% (average 14%) in CMPs with DRAM-based and PCM-based main memory.

Categories and Subject Descriptors: B.3.1 [Memory Structures]: Semiconductor Memories

General Terms: Design, Performance

Additional Key Words and Phrases: Memory controller, memory subsystem, memory request scheduling, writeback-aware management

ACM Reference Format:

Hsiang-Yun Cheng, Mary Jane Irwin, and Yuan Xie. 2015. Adaptive Burst-Writes (ABW): Memory requests scheduling to reduce write-induced interference. *ACM Trans. Des. Autom. Electron. Syst.* 21, 1, Article 7 (November 2015), 26 pages.

DOI: <http://dx.doi.org/10.1145/2753757>

1. INTRODUCTION

In modern chip multiprocessors (CMPs), main memory access latencies have become the major performance bottleneck. The limited on-chip caches cannot always store all the data requested by the running applications, and a last-level cache (LLC) miss can stall a requesting core for hundreds of cycles. As more cores on a chip contend for the limited memory bandwidth, this bottleneck is expected to grow. Since reads are on the critical path for program execution, many previous studies have focused on memory scheduling of reads to optimize system throughput and fairness [Kim et al. 2010a, 2010b; Mutlu and Moscibroda 2007, 2008; Awasthi et al. 2010; Rixner et al. 2000]. Nevertheless, writes often compete with reads for the precious memory resources,

This work is supported in part by NSF grants 1500848, 1461698, 1213052, 1017277, and Department of Energy under Award Number DE-SC0005026.

Author's addresses: H.-Y. Cheng, Computer Science and Engineering Department, Pennsylvania State University; email: hoc5108@cse.psu.edu; M. J. Irwin, Computer Science and Engineering Department, Pennsylvania State University; email: mji@cse.psu.edu; Y. Xie, Electrical and Computer Engineering Department, University of California at Santa Barbara; email: yuanxie@ece.ucsb.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2015 ACM 1084-4309/2015/11-ART7 \$15.00

DOI: <http://dx.doi.org/10.1145/2753757>

increasing the queuing delay of reads. The longer write latencies in future nonvolatile memories, such as Phase Change Memory (PCM) [Lee et al. 2009; Qureshi et al. 2009; Raoux et al. 2008; Lam 2008; Kwon et al. 2012; Niu et al. 2010], will further increase the delay of reads. This write-induced interference has a significant impact on system performance [Lee et al. 2010a] and has become a critical design issue in the memory system.

The write-induced interference mainly comes from two sources, the write-to-read turnaround time (tWTR) and the write recovery time (tWR) [Lee et al. 2010a, 2010b; Jacob et al. 2007]. Since the I/O gating resource is shared between reads and writes, a read request after writes needs to wait for an additional tWTR delay to change the direction of the data bus. Additionally, if the request following a write accesses a different row, an additional tWR delay is required to propagate the write data into the memory arrays. These tWTR and tWR delays in a state-of-the-art DDR3 DRAM system are 7.5 and 15 ns [Micron-DDR3], translating to 30 and 60 processor cycles on a 4GHz processor. Reducing the number of transitions between reads and writes can eliminate the tWTR penalty, while the tWR penalty can be reduced by increasing the row-buffer locality.

To handle the write-induced interference, previous studies proposed mechanisms to service writes in a burst [Chatterjee et al. 2012, Lee et al. 2010a, 2010b, Zhou et al. 2012, Stuecheli et al. 2010]. In conventional memory controller, reads are prioritized over writes. Since writes are not on the critical path, they are buffered at the write queue in the processor's memory controllers until the write queue is nearly full (reaches a high watermark). To amortize the write-to-read turnaround delay, a large number of writes are serviced in a burst until the write queue is nearly empty (reaches a lower threshold). This process is called write queue drain (WQD) [Chatterjee et al. 2012, Lee et al. 2010a, 2010b]. In addition to eliminating the tWTR penalty, WQD also increases the row-buffer locality by exposing more writes together. However, servicing a larger number of writes in a burst incurs a longer queuing delay to the reads waiting in the read queue. This queuing delay introduced by long WQD hurts system performance, especially when the arrival rate of reads is high.

In this article, we observe that the better number of writes that should be serviced in a burst varies for different applications. Although longer WQD eliminates the tWTR and tWR penalties, it may hurt performance when the row-buffer miss rate of writes and the arrival rate of reads are high during the WQD processes of an application. We then propose an Adaptive Burst-Writes (ABW) mechanism that includes two run-time techniques, WQD early termination (ET) and history-based WQD early termination (hET), to dynamically select a better WQD length (the number of writes serviced in each WQD process) for different workloads according to the row-buffer locality of writes and the arrival rate of reads. The hET further utilizes the row-buffer miss rate in the previous WQD process to predict whether the current WQD process should be terminated earlier for shorter queuing delay of reads, and to thereby improve system performance.

In summary, this article offers the following contributions.

- We show that a longer WQD is not always a better choice for all applications. The benefit of a longer WQD is determined by the row-buffer hit rate of writes and the arrival rate of reads during the WQDs. To the best of our knowledge, there is no previous study that analyzes the pros and cons of a longer WQD for different workloads.
- We design a novel mechanism with low hardware overhead to select a better WQD length for workloads with different characteristics. Our run-time mechanism changes

Fig. 1. Baseline CMP and memory system (MC = Memory controller, RQ = Read queue, WQ = Write queue).

the WQD length for different program phases according to the number of row-miss writes and the arrival rate of reads.

- We evaluate our proposed ET and hET techniques, with cycle-accurate simulations and detailed memory models. Overall, our mechanism can provide up to 28% (average 10%) and 43% (average 14%) throughput improvement in CMPs with DRAM-based and PCM-based main memory, compared to the conventional WQD policy.
- We analyze the scalability of our mechanism to the systems with different write-queue implementations, higher number of cores, and PCM-based main memory. The high throughput improvement implies that our proposed scheme is promising for future many-core systems with emerging memory technologies.

The rest of the article is organized as follows. Section 2 describes the typical organization of the memory system and the pros and cons of a longer WQD. Our Adaptive Burst-Writes (ABW) mechanism, including both ET and hET, is explained in detail in Section 3. Section 4 and Section 5 include the evaluation method and results, followed by a summary of related work in Section 6. Finally, we conclude the article in Section 7.

2. BACKGROUND AND MOTIVATION

In this section, we first introduce main memory background and terminologies that are related to write-induced interference. We then use some motivating examples to illustrate that not all the workloads benefit from longer WQDs, and the better WQD length depends on the row-buffer hit rate of writes and the arrival rate of reads.

2.1. Main Memory Background and Terminologies

Figure 1 illustrates our baseline CMP and memory system. Each core has its own local cache, and all the cores share a last-level cache. To maximize memory bandwidth, there are multiple memory channels in the system. Each channel is managed by one memory controller (MC), and is connected to one or more DIMMs, each containing multiple DRAM chips. These DRAM chips are logically arranged into multiple ranks. Internally, each DRAM chip is partitioned into banks that can be accessed in parallel. Three steps are required to access a data element in the bank. First, a precharge command is sent to precharge the bank's bitlines. Second, an activate command is sent to open the target row through the sense amplifier (row buffer). The row buffer keeps the last accessed memory row in the bank. Finally, a read or write command is scheduled to access the target column from the row data in the row buffer. If the subsequent request accesses the same row in the bank, this row-hit request can be performed simply by accessing the target column from the currently opened row. Otherwise, these three steps (precharge, activate, read/write) must be performed again to complete the row-miss request.

Read Queue (RQ) and Write Queue (WQ). In each memory controller, there is one read queue and one write queue to buffer the outstanding memory requests. A scheduler is

associated with the queues to determine which request should be serviced next. For each channel, only one request can be sent to the memory through the bus at each clock cycle. Although different banks can service requests in parallel as long as the bus is not occupied, each individual bank can only service one request at one time. Therefore, writes in the WQ often compete with reads in the RQ for the limited bus bandwidth and bank resources. When a write request is serviced instead of a critical read request, the read request needs to satisfy two timing constraints after the write is serviced. These two major timing constraints are tWR and $tWTR$.

The write recovery latency (tWR) and the write-to-read turnaround delay ($tWTR$). The tWR latency is the minimum latency from a write data burst to a precharge command in the same memory bank. When a subsequent precharge command is scheduled to open a different row after a write to a bank, the precharge command needs to wait until the modified data in the row buffer is completely written back to the corresponding row in the bank. This tWR timing constraint is required so that the memory avoids the loss of modified data. It can be eliminated if subsequent write requests access the same row in the bank. While the tWR latency specifies the constraint to access different rows in the same bank, the $tWTR$ latency guarantees the signal integrity when sharing the I/O gating between reads and writes in the same rank. The minimum latency from a write data burst to a column-read command is the $tWTR$ penalty. This latency is required to change the direction of the data bus between the read and write states. If the number of transitions between the read and write requests can be reduced, the $tWTR$ penalty can be eliminated.

To reduce the $tWTR$ and tWR penalty, a write queue drain (WQD) policy is applied in the memory controller to service a large number of writes in a burst [Chatterjee et al. 2012; Lee et al. 2010a, 2010b]. Since writes are not on the critical path, they are buffered at the WQ until the WQ is nearly full (the number of pending writes reaches a specified higher threshold, WQHT). When the number of pending writes is larger than WQHT, writes in the queue have to be serviced until the WQ is nearly empty (number of pending writes is less than a specified lower threshold, WQLT). This WQD process can amortize the $tWTR$ penalty by reducing the number of switches between reads and writes. Furthermore, it is more likely to find a row-hit write for eliminating the tWR penalty as the WQD length (the number of writes serviced in a burst) increases. However, the reads have no choice but to wait at the RQ when the burst-writes are being serviced. This queuing delay introduced by long WQD hurts system performance, especially when the arrival rate of reads is high. To our best knowledge, this problem is not discussed in previous work.

2.2. Motivating Examples

We first illustrate that a longer WQD (servicing more writes in a burst) can increase the row-buffer hit rate of writes by exposing more writes in the WQ to find a row-hit write. With higher row-buffer locality, each WQD process takes fewer cycles by eliminating the tWR penalty. Figure 2(a) shows the row-buffer hit rate of writes when a different number of writes are serviced in a burst during each WQD process. The experiments are running on a four-core system with DRAM-based main memory as specified in Section 4, and each core runs one copy of the SPEC CPU2006 benchmark. As the number of writes serviced in a burst increases from 4 to 32, the row-buffer hit rate also increases. In some workloads, such as *gromacs*, *leslie3d*, *milc*, and *mcf*, the improvement in row-buffer hit rate is high, as more writes are serviced in a burst. However, the row-buffer hit rate of some other workloads, such as *astar* and *zeusmp*, only slightly changes when the WQD length increases. If the arrival rate of reads is

Fig. 2. (a) Row buffer hit-rate of writes and (b) average number of read requests from LLC during each WQD under different WQD lengths (WQD4 to 32: service 4 writes to 32 writes in a burst).

Fig. 3. Performance under different WQD lengths (StaticOptimal: statically profile the optimal WQD length).

high during the WQD processes in these low-hit-rate workloads, servicing more writes in a burst may hurt performance by introducing a longer queuing delay of reads.

Although the row-buffer hit rate of writes often increases as more writes are serviced in a burst, the number of reads delayed by each WQD also increases, especially when the arrival rate of reads is high. Figure 2(b) shows that the average number of reads enqueued by the LLC during each WQD (inR) increases when the WQD length increases. In some workloads, such as *hmmmer*, *zeusmp*, and *leslie3d*, the inR increases dramatically when more writes are serviced in a burst. For these workloads, a longer WQD may hurt performance since a larger number of critical reads are delayed for a longer time. On the other hand, in some workloads, such as *libquantum*, *mcf*, and *gobmk*, the inR only increases slightly as the WQD length increases. These workloads with lower arrival rate of reads are more likely to benefit from a longer WQD.

Since different workloads receive varying row-buffer-hit benefits and suffer different queuing penalties from a longer WQD, they perform better at different settings of the WQD length. Figure 3 shows that different applications perform best when a different number of writes are serviced in a burst. When the WQD length decreases from 32 to 16, some workloads, such as *astar*, *hmmmer*, *zeusmp*, and *bwaves*, perform better. For these workloads, increasing the WQD length only slightly improves the row-buffer locality, but seriously delays more reads, as shown in Figure 2(b). On the other hand, some workloads, such as *libquantum* and *gobmk*, perform worse at a shorter WQD, since the number of reads delayed by WQD only slightly increases when the WQD length increases. We statically run the workloads with different WQD lengths, and select the WQD length that provides the highest performance as the StaticOptimal. Figure 3 shows that the StaticOptimal provides about 7% performance

Fig. 4. Architecture of adaptive burst-writes (pRBMw: hardware counter for previous WQD; RBMw, inR, nW, and ArvR: hardware counters for current WQD).

improvement when running duplicate workloads in the four-core system with DRAM-based main memory. When the workloads are composed of a mixture of applications, the performance benefit brought by the StaticOptimal further increases to 13% in DRAM-based main memory, as shown in Section 5.

In summary, a longer WQD provides higher row-buffer hit rate of writes, but also delays a higher number of reads. The better WQD length for different workloads is determined by the row-buffer hit rate of writes and the arrival rate of reads. Therefore, a run-time mechanism that adjusts the WQD length according to different program behavior is required to achieve better system performance. In the following section, we will describe two run-time techniques to tackle this problem.

3. ADAPTIVE BURST-WRITES (ABW)

In this article, we propose an ABW mechanism, which includes two low-cost techniques, WQD early termination (ET) and history-based WQD early termination (hET), to dynamically adjust the number of writes serviced in a burst for different workloads. Both ET and hET select the WQD length according to the row-buffer locality of writes and the arrival rate of reads. The hET further terminates the WQD earlier if the row-buffer miss rate of writes in previous WQD is high.

Figure 4 shows the architecture of the proposed scheme. Some additional counters are required to capture the memory access behavior. To estimate the row-buffer locality, an additional counter, RBMw, is required to monitor the number of row-miss writes during the current WQD. The counter inR tracks the number of reads that are enqueued by the LLC during the current WQD, and nW monitors the number of writes that are already drained in the current WQD process. Thus, the arrival rate can be calculated as inR/nW (the relative request and service rate between reads and writes), and updated to the register ArvR. In addition to the access behavior in the current WQD, hET further utilizes the row-buffer miss rate of writes in the previous WQD process (recorded by pRBMw) to predict the row-buffer locality in the current WQD. If pRBMw is high, the hET terminates WQD earlier to reduce the queuing delay of reads.

Our run-time mechanism only determines whether a read or write should be issued, and it is orthogonal to other scheduling policies of reads [Kim et al. 2010a, 2010b, Mutlu and Moscibroda 2007, 2008, Awasthi et al. 2010, Rixner et al. 2000]. After our mechanism decides to schedule from the RQ or WQ, the scheduler follows the scheduling policies proposed in previous studies, such as FR-FCFS [Rixner et al. 2000], to select one read or write request to issue. In the following sections, we describe these two run-time techniques in detail, and use an example to show the difference between conventional WQD, ET, and hET. The hardware counters and thresholds we used are summarized in Table I.¹

¹Row-buffer miss rate and arrival rate of reads can be represented by fixed point numbers with 2^{-6} accuracy.

Table I. Hardware Counters and Thresholds Summary

Name	Description	Storage Overhead
RBMw	Row-buffer write misses in current WQD	$\log(WQlength) = 6\text{-bits}$
pRBMw	Row-buffer miss rate of writes in previous WQD	6-bits
inR	Reads enqueued by LLC in current WQD	$\log(RQlength) = 6\text{-bits}$
nW	Writes already drained in current WQD	$\log(WQlength) = 6\text{-bits}$
ArvR	inR/nW, arrival rate of reads during the WQD	6-bits
RMth	Threshold for row-buffer write misses	$\log(WQlength) = 6\text{-bits}$
pRMth	Threshold for row-buffer miss rate of writes	6-bits
Qth	Threshold for the arrival rate of reads	6-bits
minWL	Minimum number of writes in each WQD	$\log(WQlength)\text{-bits} = 6\text{-bits}$

Fig. 5. WQD early termination (ET) (RBMw and ArvR: hardware counters; RMth and Qth: thresholds).

3.1. WQD Early Termination (ET)

Although a longer WQD provides a higher row-buffer hit rate of writes, more reads will suffer from long queuing delay, especially when the arrival rate of reads is high. To tackle this problem, we propose ET to terminate WQD earlier (before the write queue is empty) if the number of row-miss writes and the arrival rate of reads are high.

Figure 5 shows the flow of ET, and the detail is described in Algorithm 1. At the beginning of each WQD, the hardware counters are initialized (lines 2 to 4 in Algorithm 1). When issuing each write during the WQD process, ET will first check whether the number of row-buffer write misses (RBMw) is high in the current WQD process. If RBMw is lower than a predefined threshold, RMth, ET will continue WQD and issue the next write request. Otherwise, ET will then check whether the arrival rate of reads (ArvR) is high. If the arrival rate of reads is lower than a threshold, Qth, ET will continue the WQD process and update the hardware counters when the write is being serviced by the main memory (lines 9 to 15 in Algorithm 1). Otherwise, ET will predict that the row-buffer locality in the following writes is low and the queuing penalty of reads will be high. Thus, ET terminates WQD and starts to issue reads (lines 6 to 8 in Algorithm 1).

When updating the ArvR, the division of inR and nW can be implemented by table lookup [Parhami 2010] to eliminate the latency overhead. Other counters can be updated at background when the memory request is being serviced, and the comparison with thresholds before issuing from the read or write queue only takes one

Fig. 6. Late termination problem in ET. ET terminates the WQD after 6 writes are serviced, while the preferred WQD length is 4.

ALGORITHM 1: WQD early termination (ET)

```

if number of writes in WQ > WQHT then
  nW = 0;
  RBMw = 0;
  inR = 0; //inR and ArvR will be updated when new reads enter RQ
  while number of writes in WQ > WQLT do
    if RBMw ≥ RMth and ArvR ≥ Qth and nW > 0 then
      //high row-miss and queuing penalty
      break; //stop WQD and issue reads from RQ
    else
      issue one write ReqW from WQ; //continue WQD
      nW = nW + 1;
      ArvR = inR/nW; //calculated by table lookup
      if ReqW is row-miss then
        RBMw = RBMw + 1;
      end
    end
  end
else
  issue reads from RQ;
end

```

single cycle. These counters and thresholds are stored in the memory controller and the storage overhead is negligible, as indicated in Table I. Note that when selecting the next write to issue during WQD, ET follows the original scheduling policy, such as FR-FCFS [Rixner et al. 2000], to select a row-hit write in the WQ. The value of the thresholds, RMth and Qth, are related to the memory timing constraints, and is not workload-dependent. For the memory technologies with longer tWR latency, such as PCM, RMth should be set to a lower value to prevent the reads from being delayed by a large number of row-miss writes. In this article, we empirically determine the values of RMth and Qth, as detailed in Section 4, so that the average performance of all workloads is maximized.

3.2. History-Based WQD Early Termination (hET)

The ET determines the WQD length by monitoring the row-buffer locality of writes and the arrival rate of reads in the current WQD process. However, if a workload prefers a short WQD, ET may terminate the WQD too late since it needs to spend some time on monitoring the memory access behavior. For example, suppose that the RMth is set to 3 and the workload prefers to drain only 4 writes for each WQD, as shown in Figure 6. For simplicity, we assume that there are only 12 entries in each RQ and

Fig. 7. History-based WQD Early Termination (hET) ($pRBMw$, $RBMw$, and $ArvR$: hardware counters; $RMth$, and Qth : thresholds).

WQ. Figures 6(b) and (c) show the status in the RQ and WQ after the first and second row-miss writes are issued. The WQD is terminated after the third row-miss write is issued, as illustrated in Figure 6(d). The ET policy detects the high row-miss rate (50%) until the seventh write is about to be issued. In this example, the WQD length selected by ET is 6, which is longer than the preferred WQD length, and thus four additional reads are delayed.

To solve this late-termination problem, we propose hET to determine the WQD length by both the access behavior in the previous and the current WQDs. Since the data access pattern in the same program phase is similar, we use the row-buffer miss rate of writes in the previous WQD to predict the row-buffer locality in the current WQD. For the benchmarks we evaluated in Section 5, the difference in the row-buffer miss rate of consecutive WQDs is within 15% most of the time ($>80\%$ program execution). By using the prediction, the hET can solve the late-termination problem of ET and terminate the WQD earlier for the applications that prefer a shorter WQD.

The flow of hET is shown in Figure 7, and the detail is explained in Algorithm 2. To avoid frequent transitions between servicing reads and writes, each WQD first drains $minWL$ writes (lines 5 to 12 in Algorithm 2). The hET then checks whether the row-buffer miss rate of writes in the previous WQD process ($pRBMw$) is high. If $pRBMw$ is higher than or equal to a predefined threshold, $pRMth$, the hET predicts that the row-buffer locality will be low in the following writes and immediately terminates WQD (lines 14 to 16 in Algorithm 2). Otherwise, it follows the same flow in ET to determine whether to continue WQD or stop the WQD process according to the number of row-buffer write misses and the arrival rate of reads (lines 18 to 28 in Algorithm 2). Once the WQD is terminated, the $pRBMw$ is updated by calculating $RBMw/nW$ (lines 31 in Algorithm 2). Note that only simple comparison is required to decide whether to issue

ALGORITHM 2: history-based WQD early termination (hET)

```

if number of writes in WQ > WQHT then
  nW = 0;
  RBMw = 0;
  inR = 0; // inR and ArvR will be updated when new reads enter RQ
  while nW < minWL do
    // avoid frequent switch between reads/writes
    issue one write ReqW from WQ;
    nW = nW + 1;
    if ReqW is row-miss then
      RBMw = RBMw + 1;
    end
  end
  while number of writes in WQ > WQLT do
    if pRBMw ≥ pRMth then
      // predict low row-buffer locality
      break; // stop WQD and issue reads from RQ
    else
      // follow the same flow in ET
      if RBMw ≥ RMth and ArvR ≥ Qth and nW > 0 then
        break; // stop WQD and issue reads from RQ
      else
        issue one write ReqW from WQ;
        nW = nW + 1;
        ArvR = inR/nW; // calculated by table lookup
        if ReqW is row-miss then
          RBMw = RBMw + 1;
        end
      end
    end
  end
  pRBMw = RBMw/nW;
else
  issue reads from RQ;
end

```

from the read queue or write queue, and the counters are updated out of the critical path when the memory request is being serviced. The division of *inR* and *nW* can be implemented by table lookup [Parhami 2010]. Since *inR* and *nW* are only $\log(RQlength)$ -bits and $\log(WQlength)$ -bits, the storage overhead is small and the table access is fast. Thus, our ET and hET schemes incur at most negligible latency overhead. The setting of the thresholds, *pRMth*, *RMth*, and *Qth*, is closely related to the memory timing constraints, and the best value does not differ drastically among variant workloads. In this article, we empirically determine the values of *pRMth*, *RMth*, *Qth*, and *minWL*, to maximize the average performance of all workloads, as detailed in Section 4.

3.3. Example

Figure 8 illustrates the difference between conventional WQD, ET, and hET. For simplicity, we assume that there are only 12 entries in each RQ and WQ. The conventional WQD starts when there are 12 writes in WQ, and drains writes until the WQ is empty (*WQHT*=12, *WQLT*=0). In this example, *Qth*=1/3, *RMth*=3, *minWL*=2, and *pRMth*=50%.

Fig. 8. Scheduling example with different policies (Queuing penalty of hET < ET < conventional WQD. a to f: row address).

Figure 8(a) shows that when WQD starts, there are two reads in the RQ, the counters (nW, inR, RBMw) are initialized to zero, and the row-buffer miss rate in the previous WQD is high (70%). When selecting write requests from WQ, we follow the FR-FCFS policy [Rixner et al. 2000] to prioritize row-hit writes. After issuing two writes to row *a* (the first write is row-miss and the second is row-hit), hET will terminate WQD immediately since it predicts that the row-buffer locality of writes is low ($pRBMw$ (70%) $\geq pRMth$ (50%)), as shown in Figure 8(b). When the two writes are being serviced, there is now one more outstanding read request from the LLC. Thus, there are three reads delayed by WQD and the WQD length of hET is equal to two. On the other hand, ET will continue the WQD process as the number of row-buffer write misses is low ($RBMw < RMth$). Figure 8(c) shows that after draining seven writes, there are three row-buffer write misses. However, ET will continue to issue more writes as the arrival rate of reads is low ($inR/nW = 2/7 < Qth$ (1/3)). The ET will continue to issue writes until the arrival rate of reads ($inR/nW = 4/10$) is higher than Qth (1/3), as shown in Figure 8(d). Since ET spends some time to monitor the row-buffer locality and the arrival rate of reads, six reads are delayed by the WQD and the WQD length is 10. The conventional WQD will continue to issue writes and delay eight reads during the whole WQD process, as shown in Figure 8(e).

In this example, the row-buffer miss rate of writes (58% in Figure 8(e)) and the arrival rate of reads (50% in Figure 8(e)) are high when all writes in the WQ are serviced in a burst. Therefore, a shorter WQD is preferred as the queuing penalty of reads outweighs the reduction in the tWR and tWTR penalties. From the example, we can see that the queuing penalty of reads is lowest in hET, since hET terminates the WQD at the earliest stage. Although hET may miss-predict the row-buffer locality when the program behavior changes frequently, we will show in Section 5 that hET outperforms ET and the conventional WQD in most of the workloads.

Table II. System Configuration

Processor	4/8 cores; 3GHz; Out-of-order; issue width=8; 192-entry reorder buffer; 32-entry load-store queue
L1 I&D-cache	Private; 16KB per core; 2-way;
L2 cache	Shared; 1MB total; 16-way; 128-entry MSHR; 64-byte block size
Memory Controller	FR-FCFS with WQD; 1 channel; 48-entry RQ/WQ; WQHT=32; WQLT=0
DRAM Configuration	8GB; 2 ranks/DIMM; 8 chips/rank; 8 banks/chip; MT41J512M8 DDR3-1300 (667MHz) [Micron-DDR3]
PCM Configuration	8GB; 4 ranks/DIMM; 4 chips/rank; 8 banks/chip; LPDDR2-800 interface (400MHz) [Micron-LPDDR2]

Table III. DRAM and PCM Timing Specifications

Latency	Symbol	DRAM	PCM
Precharge	tRP	15ns	20ns
Activate to read/write	tRCD	15ns	55ns
Read column address strobe	tCL	15ns	15ns
Column write to data valid on bus	tCWD	13ns	13ns
Burst length	tBL	6ns	20ns
Between Column address strobes	tCCD	6ns	5ns
Activate to activate (different bank)	tRRD	6ns	10ns
Four activate windows	tFAW	30ns	50ns
Write to read (bus-turnaround)	tWTR	7.5ns	7.5ns
Write recovery	tWR	15ns	200ns

4. EXPERIMENTAL SETUP

Simulation infrastructure. We evaluate our designs using the cycle accurate gem5 simulator [Binkert et al. 2011], and model the main memory system in detail by using NVMain [Poremba and Xie 2012]. NVMain is a cycle-accurate main memory simulator designed to simulate both conventional DRAM-based main memory and emerging non-volatile memories. We modified the memory controller in NVMain to implement our ABW mechanism. The baseline configuration for the CMP system is shown in Table II, and the timing parameters in the main memory are described in Table III. Our baseline processors are 8-way out-of-order cores with a 192-entry reorder buffer and a two-level cache hierarchy. The L1 instruction and data caches are 2-way 16KB each, while the L2 cache is a unified 16-way 1MB cache. Since we focus on environments that stress the memory controller, we use an L2 cache size that is smaller than typical LLC in existing multi-core machines. Our ABW provides little performance benefit to systems with light memory traffic (large LLCs and compute-intensive workloads). Both DRAM-based and PCM-based main memory are evaluated. The DRAM device model and timing parameters are derived from the Micron DDR3-1333 data sheet [Micron-DDR3]. For PCM, we use the Micron LPDDR2-800 data sheet [Micron-LPDDR2] to derive the timing parameters, and use NVSim [Dong et al. 2012] to model the read/write-related latency of PCM with 20nm diode-switched PCM cells [Choi et al. 2012].

Evaluated policies. Table IV summarizes the scheduling policies we evaluated and the parameter settings for different designs. The baseline scheduling policy (WQD32) in the memory controller is FR-FCFS [Rixner et al. 2000] with WQD that drains the

Table IV. Scheduling Policies in the Memory Controller

WQD32 (baseline)	Conventional WQD [Lee et al. 2010a, 2010b; Stuecheli et al. 2010; Jiang et al. 2014]. Drain 32 writes during each WQD process; WQHT = 32; WQLT = 0
WQD16	The WQD policy applied in [Chatterjee et al. 2012]. Drain 16 writes during each WQD process; WQHT = 32; WQLT = 16
WQDopt	The WQD length that provides the highest average performance among workloads. Drain opt writes during each WQD process; 4-core DRAM: opt = 12; 8-core DRAM: opt = 8; 4-core PCM: opt = 8
AHB	Adaptive history-based memory scheduling [Hur and Lin 2004, 2007]. 2-bits read-write history. Select reads or writes according to: (1) Read-write ratio of applications (30% probability) or (2) Access latency (70% probability).
Wcancel	Write cancellation policy [Qureshi et al. 2010]. Abort the processing of a write request if a read request arrives to the same bank. If a write request is >75% complete or the number of queued writes > WQHT, continue to process the write. Reads and writes are scheduled by FCFS.
RT	Naive policy that determines the WQD length by the read queue occupancy. Drain writes until the write queue is empty or reads in the read queue > Rth 4-core DRAM: Rth = 6; 8-core DRAM: Rth = 12; 4-core PCM: Rth = 6
ET	WQD early termination policy. 4-core DRAM: RMth = 3; Qth = 10%; 8-core DRAM: RMth = 2; Qth = 5% 4-core PCM: RMth = 1; Qth = 10%
hET	History-based WQD early termination. 4-core DRAM: pRMth = 60%; RMth = 3; Qth = 10%; minWQDL = 4 8-core DRAM: pRMth = 60%; RMth = 2; Qth = 5%; minWQDL = 4 4-core PCM: pRMth = 70%; RMth = 1; Qth = 10%; minWQDL = 4
StaticOptimal (SO)	Statically profile the performance with different WQD length (from 1 to 32) and select the best one for each individual workload.

WQ until it is empty [Lee et al. 2010a, 2010b; Stuecheli et al. 2010; Jiang et al. 2014]. We also evaluate the policy that drains only half of the write queue (WQD16). In addition, the best static and fixed choice of WQD length that provides the highest average performance (WQDopt) is also evaluated. The best static and fixed WQD length is shorter in 8-core and PCM-based systems, compared to the 4-core DRAM-based system, because the heavy request traffic in the 8-core system and the long write latency in the PCM incur long queuing delays when large numbers of writes are serviced in a burst. We also compare ABW to adaptive history-based memory scheduling (AHB), which schedules reads and writes according to the read-write request ratio from the application to avoid the long queuing delay [Hur and Lin 2004, 2007]. The write cancellation policy [Qureshi et al. 2010] that reduces the queuing latency of reads by aborting the processing of writes when reads arrive is also evaluated. To illustrate that both the arrival rate of reads and the row buffer hit rate of writes need to be considered when scheduling reads and writes, we compare that to a naive approach (RT) that services writes until the write queue is empty or the number of reads in the read queue is higher than a predefined threshold, Rth. Note that the Rth is set to a higher value in the 8-core system, because the higher request rate from eight cores would easily achieve a low Rth threshold, resulting in frequent transitions between reads and writes. The static optimal WQD length for each individual workload is also evaluated to indicate the effectiveness of our ABW mechanism.

Table V. Workload Combinations

	Benchmarks	MPKI	RBH%	Qocc%	Pred%
4cS1	hmmmer(4)	2.14	37.83	19.80	97.19
4cS2	libquantum(2),hmmmer(2)	3.49	62.12	16.64	84.49
4cS3	astar,zeusmp,gobmk,hmmmer	5.55	62.26	28.26	87.37
4cS4	hmmmer,leslie3d,lbm,gobmk	10.31	68.25	30.61	76.87
4cS5	gobmk,stream,mcf,milc	24.17	75.03	25.86	74.45
4cL1	libquantum(4)	6.95	68.87	14.39	94.59
4cL2	libquantum(2),gobmk(2)	12.69	85.32	39.25	85.92
4cL3	libquantum(2),lbm(2)	14.36	59.51	24.43	92.82
4cL4	stream(2),lbm(2)	33.45	47.52	28.20	93.80
4cL5	gobmk(4)	38.40	85.78	58.40	82.73
8cS1	hmmmer(5),astar,zeusmp,gobmk	3.13	37.75	44.68	99.61
8cS2	hmmmer(4),gobmk,astar,mcf,milc	3.34	45.00	47.76	99.20
8cS3	hmmmer(2),gobmk(2),astar,zeusmp,mcf,milc	4.89	48.23	51.36	98.14
8cS4	hmmmer(3),libquantum(2),leslie3d,gobmk,lbm	5.81	52.12	42.28	91.40
8cL1	libquantum(8)	6.95	76.21	30.28	81.58
8cL2	libquantum(6),lbm(2)	9.42	62.79	26.61	82.81
8cL3	libquantum(4),lbm(2),gobmk(2)	13.61	65.95	56.99	81.19
8cL4	libquantum(4),gobmk(4)	13.84	78.67	73.52	75.36

RBH: row-buffer hit rate; Qocc: average read queue occupancy at the end of WQDs; Pred: prediction accuracy of row-miss writes in hET.

Workloads. We use a variety of benchmarks from SPEC2006, and STREAM [McCalpin 1995]. The selected benchmarks from SPEC2006 are memory-intensive programs with larger than 0.5 miss-per-kilo-instructions (MPKI>0.5) when running the reference input set on a 4-core system. We chose 10 and 8 multi-programmed workload combinations for our 4-core and 8-core CMP configurations.² The characteristic of these multi-programmed workloads is shown in Table V, and we classify the workloads into two categories by their preference to shorter (4cS1-4cS5, 8cS1-8cS4) or longer (4cL1-4cL5, 8cL1-8cL4) WQDs. Workload combinations with low to high RQ occupancy (Qocc) are all covered. Furthermore, the row-buffer miss rates of writes in consecutive WQDs are highly correlated, as illustrated by the high prediction accuracy (>75%) in the table. We fast forward the simulation by 500 million instructions on each core, and run each application for 100 million instructions.³ When reporting performance results, we use two metrics: overall throughput and weighted speedup [Snively and Tullsen 2000], which are defined as follows.

$$Throughput = \sum IPC_i. \quad (1)$$

$$Weighted Speedup = \sum (IPC_i / IPC_i^{single}). \quad (2)$$

²We also evaluated multi-threaded workloads by running the entire PARSEC benchmark. The average performance improvement is smaller in PARSEC than in SPEC2006 since most of the PARSEC benchmarks are compute-intensive and do not put pressure on the memory controller. Nevertheless, there is still non-negligible performance gain (>5%) in some memory-intensive applications, such as fluidanimate, facesim, and streamcluster.

³We also ran longer simulations (300M instructions) for half of the workloads (cover different types), but the result showed that running 100 million instructions is representative enough. Thus, we choose to run 100 million instruction simulations for faster evaluation.

Fig. 9. Throughput improvement in the 4-core system with DRAM-based memory.

Fig. 10. Speedup improvement in the 4-core system with DRAM-based memory.

Fig. 11. Average number of writes serviced in a burst.

In the equations, the IPC_i is the IPC of $program_i$ when running with the rest of the workload, while IPC_i^{single} represents the IPC of the same program when it runs alone on one core of the CMP system (other cores are idle).

5. EXPERIMENTAL RESULTS

In this section, we first analyze the performance impact of our ABW mechanism in the 4-core system, and study the queuing latency of reads that are delayed by WQDs. To understand the effectiveness of our schemes, we further analyze the throughput improvement for the 4-core systems with different WQ parameters, different threshold settings, different number of channels, and the 8-core system with serious memory contention. We then show that our mechanism also improves the throughput for the systems with a PCM-based main memory.

5.1. 4-Core Performance

Figures 9 and 10 show the impact of ET and hET on the throughput and weighted speedup in the 4-core system. For the workloads that prefer shorter WQDs (4cS1-4cS5), ET and hET perform better than both WQD32 and WQD16 by terminating WQDs earlier to reduce the queuing latency of reads. As shown in Figure 11, the WQD lengths selected by ET and hET are less than 32 and 16 in these workloads, approaching the WQD length reported by the StaticOptimal. The performance of ET and hET are also better than WQDopt, which drains 12 writes at each WQD process, as the average WQD lengths in ET and hET are lower than 12 and closer to the StaticOptimal. The hET provides better performance than ET, because it utilizes the row-buffer locality in the previous WQD to terminate the current WQD earlier when the queuing penalty of reads outweighs the benefit of a higher row-buffer hit rate. For the workloads that prefer longer WQDs (4cL1-4cL5), ET and hET also provide better performance, except

in 4cL2, compared to WQD32 and WQD16. As illustrated in Figure 11, our mechanism overestimates the queuing penalty of reads for 4cL2 and conservatively selects the WQD lengths that are much lower than the one selected by StaticOptimal. Nevertheless, ET and hET perform better than the StaticOptimal at 4cL1 and 4cL5, since our mechanism can dynamically adjust the WQD length for different program phases. On average, ET and hET provide 8.5% and 10.2% throughput improvement, compared to the baseline WQD32. Our mechanism also provides balanced improvement on each individual program, thus improving the weighted speedup by 7.4%.

Figure 9 also compares the throughput improvement of our schemes and the previous work, AHB [Hur and Lin 2004, 2007]. The AHB schedules reads or writes to memory according to the read-write ratio of the application. Three scheduling patterns (2R1W, 1R1W, and 1R2W) are provided by the AHB scheduler. When AHB is applied, the writes are not serviced in a burst and the memory controller frequently switches between servicing reads and writes. Thus, AHB provides high throughput in some workloads (4cS1-4cS5 and 4cL2) due to the reduced queuing penalty of reads. However, the frequent switches between reads and writes decrease the row-buffer hit rate and increase the tWTR penalty. Therefore, AHB performs worse than the baseline WQD at some workloads (4cL1, 4cL4, and 4cL5). Our ET and hET provide better trade-offs between the row-buffer miss penalty and the queuing penalty of reads. As a result, our schemes outperform the AHB approach by about 5% on average.

We also compare the performance improvement of our schemes with the write-cancellation policy (Wcancel) [Qureshi et al. 2010], as illustrated in Figure 9. The Wcancel policy aborts normal writes to service critical reads, so it helps to reduce the queuing latency of reads, especially in the workloads that prefer a shorter WQD (4cS1-4cS5). However, for the workloads that prefer a longer WQD, the Wcancel policy does not benefit from the higher row-buffer hit rate in longer WQDs. Moreover, the rescheduling of aborted writes wastes memory bandwidth, and these rescheduled writes may delay the read requests. Since the Wcancel policy does not eliminate tWR and tWTR when servicing these writes, our policy performs better at the workloads that prefer a longer WQD (4cL1-4cL5) and provides 4% higher throughput improvement than Wcancel on average.

In Figure 9, we also show the performance of RT that determines the WQD length by the read queue occupancy. Since RT does not take the row-buffer locality into consideration and always terminates WQDs as long as there are more than six pending reads, the WQD length chosen by RT is always short, as shown in Figure 11. For the workloads, such as 4cS3, 4cS4, and 4cS5, that prefer shorter WQDs and benefit less from the row-buffer locality of burst-writes, RT performs slightly better than ET and hET, since the WQD length selected by RT is close to the StaticOptimal. However, for the workloads that prefer longer WQDs (4cL1-4cL5), RT performs worse than both ET and hET due to its frequent switches between reads and writes. The performance degradation of RT is further exacerbated at some workloads, such as 4cL1, as the row-buffer miss penalty outweighs the benefit of shorter queuing delay when only few writes are serviced in a burst.

5.2. Queuing Delay of Reads

The performance improvement in ET and hET comes from the reduction in the queuing penalty of reads. Figure 12(a) shows the average number of reads that are delayed by each WQD in the 4-core system with DRAM-based main memory. Since ET and hET terminate WQD earlier when the row-buffer write misses and the arrival rate of reads are high, the average number of reads delayed by each WQD decreases, compared to the baseline WQD32. The history-based approach further reduces the number of delayed reads by utilizing the row-buffer locality in the previous WQD to terminate

Fig. 12. (a) Average number of reads delayed by each WQD, (b) average queuing latency of reads delayed by WQDs, (c) percentage of reads delayed by WQDs, and (d) read latency normalized to WQD32 in the DRAM-based main memory.

the WQDs that hurt performance at an earlier stage. Figure 12(b) shows that not only is the number of delayed reads reduced, but also the queuing latency of reads caused by WQDs. A higher number of WQD processes is required to service all the writes when fewer writes are serviced in each WQD process. The additional WQD processes may delay more reads in the workloads. Therefore, a higher portion of reads is delayed by writes when the WQD length is reduced from 32 to 16, as shown in Figure 12(c). Our ET and hET policies dynamically adapt the WQD length according to the row-buffer locality and arrival rate of reads, so the percentage of reads delayed by WQDs is higher than WQD32 but lower than WQD16. Although more reads are delayed by a higher number of shorter WQDs, the benefit of the reduction in queuing delay outweighs the penalty of the increase in delayed reads. As a result, the average access latency of reads is reduced, as shown in Figure 12(d). The reduction in the workloads (4cS1-4cS5) that prefer a shorter WQD is higher than other workloads, since our mechanism terminates WQDs earlier in these workloads to reduce the queuing penalty of reads. On average, the access latency of reads is decreased by 9% and 10% when ET and hET are applied.

5.3. Sensitivity and Scalability

To analyze the scalability of our ABW mechanism on different DRAM-based systems, we evaluate the throughput improvement for the 4-core systems with different baseline WQD thresholds (different WQHT) and an 8-core system with serious memory contention. We also evaluate the impact of different threshold settings, and analyze the performance improvement in the 4-core systems with a different number of channels.

5.3.1. Write Queue Parameters. The choice of the high and low thresholds (WQHT and WQLT) for the write queue determines the duration and frequency of WQD. Our proposed scheme dynamically adjusts WQLT by terminating the WQD if the queuing penalty of reads outweighs the benefits of higher row-buffer write hits. On the other hand, the value of WQHT determines the number of writes serviced in a burst in the baseline WQD. For example, the memory controller drains 16 and 48 writes when WQHT is set to 16 and 48, if the baseline policy that drains the WQ until it is empty

Fig. 13. Throughput improvement of hET in the 4-core DRAM-based system with different WQHTs.

Fig. 14. Throughput improvement in the 8-core system.

is applied. A higher WQHT causes more reads to be delayed by each WQD, while the total number of WQD processes is decreased.

Figure 13 shows the throughput improvement when our history-based approach, hET, is applied to the systems with different WQHT settings. We choose to evaluate the hET policy because hET performs better than ET in most workloads. The result shows that our hET policy provides higher throughput improvement when the WQHT is higher, especially for the workloads (4cS1-4cS5) that prefer a shorter WQD length. For the workloads (4cL1-4cL5) that prefer a longer WQD length, this trend is less evident as our hET policy aims to shorten the WQD length. There is no throughput improvement at 4cL3 and the performance degrades slightly at 4cL2, when WQHT is set to 48. The reason is that these two workloads benefit more from the higher row-buffer locality of long burst-writes and the fewer number of WQD processes that delay a lower percentage of overall reads.

When WQHT is set to 16, the baseline WQD drains 16 writes in a burst and thus delays fewer reads for shorter latency. Therefore, the potential benefit of reducing the queuing penalty of reads by using our hET policy is reduced. Nevertheless, our policy still provides 6.1% throughput improvement on average. When the value of WQHT increases to 32 and 48, the baseline WQD drains a higher number of writes in a burst and suffers from a longer queuing delay of reads if the arrival rate of reads is high. Thus, our hET policy can provide higher throughput improvement by terminating the WQD earlier. The high throughput improvement in the system with high WQHT implies that our approaches can also provide performance benefit when the cache management scheme in LLC is modified [Lee et al. 2000; Stuecheli et al. 2010] or the write queue size is increased to create a longer burst of writes. On average, the throughput improvement is 10.2% and 11.4% when WQHT is set to 32 and 48.

5.3.2. 8-Core Results. To understand the scalability of our schemes, we evaluate ET and hET for an 8-core system with serious memory contention. Due to the contention from a higher number of cores, the arrival rate of reads and the row-buffer miss rate are higher in the 8-core system. Therefore, we use lower threshold settings for Qth and RMth, as shown in Table IV, to terminate the WQD earlier when the queuing delay of reads is high. Figure 14 shows that our ABW mechanism also provides high throughput improvement in the 8-core system. On average, our mechanism provides 9.2% and 10.3% performance improvement, compared to the baseline WQD32 that

Fig. 15. Performance improvement in the 4-core system with different (a) RMth, (b) Qth, and (c) pRMth settings.

drains the WQ until it is empty. The performance improvement of ET and hET in the workloads (8cS1-8cS4) that prefer a shorter WQD length is especially high, since these workloads suffer from a higher queuing penalty of reads when the baseline WQD32 is applied in the heavily contended 8-core system. For the workloads (8cL1-8cL4) that prefer longer WQDs, our ET and hET perform similar to or slightly worse than the baseline WQD32, because the drawback of more WQD processes and delayed reads in these workloads outweighs the benefit of a shorter queuing delay of reads. In the 8-core system, our ABW mechanism also performs better than AHBs that schedule reads or writes according to the pattern of arrival requests, because the frequent switches between reads and writes in AHB incur higher tWTR overhead and row-buffer miss rate at 8cS2, 8cS3, 8cS4, and 8cL4. When compared with the Wcancel policy, the Wcancel performs better at the workloads that prefer a shorter WQD, as the Wcancel policy aborts the processing of writes to service critical reads. However, for the workloads that prefer a longer WQD, Wcancel suffers from tWR and tWTR penalties when the aborted writes are eventually processed. Thus on average, our schemes perform better than the Wcancel policy. The RT policy that considers only the occupancy of the read queue when scheduling writes also performs worse than our ET and hET, especially at 8cL1 and 8cL4, as the frequent switches between reads and writes reduces the row-buffer hit rate and increases the total number of delayed reads.

5.3.3. Threshold Settings. To understand the performance impact of different threshold settings, we evaluate ET and hET policies with varying threshold settings. Figure 15(a) shows the throughput improvement of ET with different RMth settings. A lower RMth indicates that the WQDs would be terminated even at the execution phases with higher row-buffer locality. Therefore, the workloads that prefer a shorter WQD length, such as 4cS1, 4cS4, and 4cS5, perform better when RMth is set to one, while the throughput improvement of workloads that prefer a longer WQD length, such as 4cL1, 4cL2, and 4cL4, is higher when RMth is set to five. At 4cS2, 4cS3, 4cL3, and 4cL5, the peak performance is achieved when RMth is set to three, as the setting better captures the varying-row-buffer localities in different program phases. On average, RMth=3 provides the highest performance improvement.

Figure 15(b) illustrates the performance improvement of ET with different Qth settings. A lower Qth indicates that the WQDs would be terminated earlier even at the execution phases with a lower arrival rate of reads. Thus, most of the workloads that prefer a shorter WQD length perform better at lower Qth settings, while the performance improvement of the workloads that prefer a longer WQD length is higher when the Qth is set to a higher value. On average, Qth=10% provides the peak throughput improvement.

Fig. 16. Throughput improvement in the 4-core system with different numbers of channels.

Fig. 17. Throughput improvement in the 4-core system with PCM-based main memory.

The performance impact of different pRMth settings in the hET policy is shown in Figure 15(c). When the pRMth is set to a lower value, the WQDs would be terminated earlier even when the predicted row-buffer hit rate is high. Therefore, a lower pRMth setting is beneficial for the workloads that prefer a shorter WQD length. Nevertheless, the workloads that prefer a longer WQD length usually perform worse at lower pRMth settings, as the WQD processes are terminated too early, leading to a lower row-buffer hit rate. On average, pRMth=60% provides the best performance.

5.3.4. Multiple Channels. Figure 16 shows that our hET policy also provides performance improvement to the systems with multiple channels. Since each channel has its own read and write queues, increasing the number of channels can reduce the contention for the limited write queue capacity. Therefore, the WQD process is triggered less frequently and the queuing delay of reads is reduced in the baseline WQD32 policy. As a result, our ABW mechanism provides slightly smaller performance improvement in the 4-core system with two channels, compared to the single-channel system. Nevertheless, our proposed scheme can still provide 9% performance improvement on average.

5.4. PCM Results

Recent work proposes the use of PCM as a viable main memory replacement due to its high cell density and low leakage power [Lee et al. 2009; Qureshi et al. 2009; Zhou et al. 2009; Zhang and Li 2009; Lam 2008; Raoux et al. 2008; Dhiman et al. 2009; Park et al. 2011; Mirhoseini et al. 2012; Kwon et al. 2012; Niu et al. 2010; Kim et al. 2012a]. As shown in Table III, the tWR latency in PCM is much larger than in DRAM. The higher tWR penalty makes our mechanism more compelling. Figure 17 shows that ET and hET can provide 12.7% and 14.2% throughput improvement on average, compared to the baseline WQD policy. Our policies also improve the weighted speedup by 9.5% and 10.3%, as shown in Figure 18. The performance improvement is higher in the workloads (4cS1-4cS5) that prefer a shorter WQD than other workloads, since our mechanism dynamically adjusts to a shorter WQD length when the queuing penalty of reads is high. Although ET and hET overestimate the queuing penalty of reads at two workloads, 4cL3 and 4cL4, and select a shorter WQD length than the StaticOptimal, they outperform the StaticOptimal at 4cL5 due to their dynamic adjustment for different program phases. Note that the threshold setting for row-buffer locality (RMth) in both ET and hET are lower in PCM than in DRAM, as shown in

Fig. 18. Weighted speedup improvement in the 4-core system with PCM-based main memory.

Fig. 19. (a) Average queuing latency of reads delayed by WQDs, (b) percentage of reads delayed by WQDs, and (c) average read latency normalized to WQD32 in the PCM-based main memory.

Table IV, due to the higher tWR penalty in PCM. With lower RMth, a WQD process can be terminated earlier when the queuing penalty of reads is high.

In the 4-core system with PCM-based main memory, our ET and hET policies also perform better than AHB, Wcancel, and RT, as shown in Figure 17. The short burst-writes in AHB reduce the row-buffer locality of writes and incur more than 10% performance degradation at 4cL2 and 4cL5. The Wcancel policy provides better performance at the workloads that prefer a shorter WQD by aborting the process of writes when reads arrive. However, the rescheduling of writes still delays reads in Wcancel, and these writes do not benefit from the higher row-buffer hit rate in longer WQDs. Thus, our ABW performs better than the Wcancel on average, especially at 4cL1, 4cL2, 4cL3, and 4cL5. The RT policy always selects a short WQD length, and thus degrades performance by more than 10% at the workloads that prefer a longer WQD, such as 4cL2 and 4cL3. The performance degradation of RT comes from the decrease in the row-buffer hit rate and the increase in the total number of reads that are delayed by the higher number of short WQDs.

Figure 19(a) illustrates the average queuing latency of each read caused by WQDs. As the tWR latency in PCM is higher than that in DRAM, the drop in the queuing latency of reads is also increased when ET and hET are applied, compared to the DRAM-based main memory. With shorter WQDs, the total number of WQD processes increases and the percentage of reads delayed by WQDs in the workloads also increases, as shown in Figure 19(b). Our ET and hET dynamically adapt the WQD length according to the workload behavior, so the percentage of delayed reads is slightly lower than WQD16. The benefit of shorter queuing delay outweighs the penalty of the higher number of WQD processes and delayed reads. Therefore, the average access latency of reads is reduced, as shown in Figure 19(c). On average, the access latency of reads is decreased by 15% (higher than the 10% latency reduction in DRAM) when our mechanism is applied. As a result, the performance improvement provided by our mechanism is higher in the system with PCM-based main memory (14% as shown in Figure 17) than with DRAM-based main memory (10% as shown in Figure 9). We can conclude that our

ABW mechanism is promising in future systems that use new memory technologies with high write latency (tWR) as the main memory.

6. RELATED WORK

Many DRAM scheduling policies [Rixner et al. 2000; Mutlu and Moscibroda 2007, 2008; Kim et al. 2010a, 2010b; Zhang et al. 2012] have been proposed to improve system throughput and fairness. The FR-FCFS scheduling policy [Rixner et al. 2000] is widely used in existing systems to prioritize row-hit requests. Mutlu and Moscibroda [2007] propose a stall-time fair memory scheduler to eliminate the slowdown of each individual thread. Kim et al. [2010a] propose ATLAS to maximize system throughput by prioritizing the threads that have received the least service from the memory controllers. They further propose TCM [Kim et al. 2010b] to allocate a share of the memory bandwidth for latency-sensitive applications, and shuffle the priority periodically to guarantee fairness. Zhang et al. [2012] design a memory access scheduling algorithm for memory controllers with heterogeneous channel widths. However, these mechanisms focus on reads and do not consider write-induced interference. In contrast, we propose a run-time mechanism to dynamically manage the scheduling priority between reads and writes. As such, our techniques are orthogonal to these previous scheduling policies. After our mechanism decides whether reads or writes should be serviced, these previous policies can be applied to further improve system performance.

There are some proposals that discuss write-buffer management and scheduling policies for writes [Lee et al. 2000; Natarajan et al. 2004; Lee et al. 2010a; Shao and Davis 2007; Chatterjee et al. 2012; Hur and Lin 2004, 2007; Lai et al. 2014]. Many of them [Lee et al. 2000; Natarajan et al. 2004; Shao and Davis 2007] are based on the principle that scheduling writes when the bus is idle can reduce the contention between reads and writes. However, this principle suffers from significant tWTR and tWR penalties due to frequent transitions between reads and writes. Hur and Lin [2004, 2007] propose a different approach, which selects the read-write scheduling pattern according to the read-write ratio of the applications. Although the queuing penalty of reads is reduced, their approach relies on predetermined read-write patterns and suffers from lower row-hit rates due to frequent switches between reads and writes. The comparison to their approach was described in Section 5.1 and Figure 9 (their approach is referred to as the AHB approach). Lee et al. [2010b] have shown that servicing all writes when the write queue is full performs better since it reduces the write-induced interference. In this article, we use this WQD policy as our baseline scheduling policy. Chatterjee et al. [2012] add additional buffers near the I/O pads of DRAM chips to boost read-write parallelism by overlapping part of the read operations with writes. Our mechanism is orthogonal to their work and can combine with their technique to provide better performance. Lai et al. [2014] propose read-write reordering and read-write aware throttling for DRAM, but they target reducing power consumption rather than improving system performance.

For PCM systems, several studies propose methodologies to reduce the number of writes [Hu et al. 2011; Rodriguez-Rodriguez et al. 2013; Sun et al. 2011; Huang et al. 2011]. These papers reduce the number of writes to PCM by task scheduling [Hu et al. 2011], value recomputation [Hu et al. 2011], clean-preferred LLC replacement policy [Rodriguez-Rodriguez et al. 2013], exploiting frequent-value locality [Sun et al. 2011], and register allocation [Huang et al. 2011]. However, there are still many necessary writes that need to be serviced. Other studies propose techniques to reduce the write latency to PCM [Cho and Lee 2009; Li and Mohanram 2014; Li et al. 2014; Zhang et al. 2013; Yue and Zhu 2013; Kim et al. 2012b]. These papers reduce the write latency by reducing the number of changed bits [Cho and Lee 2009], exploiting the asymmetric write latency of “1” and “0” [Li and Mohanram 2014; Li et al. 2014; Zhang

et al. 2013], leveraging subarray-level parallelism [Yue and Zhu 2013], or by hiding the R drift latency in PCM cells [Kim et al. 2012b]. Critical reads would still be delayed when these shorter writes are being serviced, and our mechanism can help to reduce the queuing latency of reads. Qureshi et al. [2010] introduce write-cancelation and write-pausing to prevent reads from being stalled by writes. Nevertheless, these writes have to eventually be serviced and such retries may increase the overall bus occupancy of writes.

In addition to the scheduling policies in memory controllers, many studies have proposed cache management mechanisms to create a long burst of writes [Wang et al. 2012; Lee et al. 2000; Stuecheli et al. 2010; Lee et al. 2010a] with high row buffer hit rate [Zhou et al. 2012; Stuecheli et al. 2010; Lee et al. 2010a, 2010b]. Lee et al. [2000] introduce a mechanism to proactively send write-backs of dirty lines to the DRAM before they are replaced. However, their proposed scheme is not aware of row-buffer locality, so it may fill the write queue with many row-conflict writes. Stuecheli et al. [2010] propose a coordinated cache and memory management policy that expose part of the cache-lines in the LLC to the memory controllers, and hence provide it with more opportunities to find a long burst of row-hit writes. However, servicing a larger number of writes in a burst incurs longer queuing delays to the reads, especially when the arrival rate of reads is high. Our mechanism aims to reduce the queuing latency of reads that are delayed by writes. In this article, we use an LRU replacement policy in the LLC. In the future, we will evaluate the throughput improvement of our ABW when these cache management schemes are applied.

7. CONCLUSION

In this work, we analyze the pros and cons of servicing a long burst of writes in the main memory, and propose a run-time mechanism to schedule reads and writes for lower queuing delays of the critical reads and better system performance. We illustrate that servicing a higher number of writes in a burst can reduce the bus turnaround penalty and increase the row-buffer hit rate by exposing more writes together. However, the queuing latency of reads also increases. To tackle this problem, we propose WQD early termination (ET) and history-based WQD early termination (hET) scheduling policies to terminate the service of burst-writes earlier and switch back to service the critical reads when the number of row-buffer misses and the arrival rate of reads are high. We demonstrate through cycle-accurate simulation that the proposed schemes provide significant throughput improvement (average 10% in DRAM and 14% in PCM) by reducing the access latency of reads (10% in DRAM and 15% in PCM) with negligible hardware overhead. Therefore, we believe that our ABW mechanism is worth considering for the memory controller designed for both DRAM-based and PCM-based main memory.

REFERENCES

- Manu Awasthi, David W. Nellans, Kshitij Sudan, Rajeev Balasubramonian, and Al Davis. 2010. Handling the problems and opportunities posed by multiple on-chip memory controllers. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*.
- Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The gem5 simulator. *SIGARCH Comput. Archit. News* 39, 2.
- Niladrish Chatterjee, Naveen Muralimanohar, Rajeev Balasubramonian, Al Davis, and Norman P. Jouppi. 2012. Staged reads: Mitigating the impact of DRAM writes on DRAM reads. In *Proceedings of the 18th IEEE International Symposium on High Performance Computer Architecture*.

- Sangyeun Cho and Hyunjin Lee. 2009. Flip-N-Write: A simple deterministic technique to improve PRAM write performance, energy and endurance. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*.
- Youngdon Choi, Ickhyun Song, Mu-Hui Park, Hoeju Chung, Sanghoan Chang, Beakhyoung Cho, Jinyoung Kim, Younghoon Oh, Duckmin Kwon, Jung Sunwoo, Junho Shin, Yoohwan Rho, Changsoo Lee, Min Gu Kang, Jaeyun Lee, Yongjin Kwon et al. 2012. A 20nm 1.8V 8Gb PRAM with 40MB/s program bandwidth. In *Proceedings of IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*.
- Gaurav Dhiman, Raid Ayoub, and Tajana Rosing. 2009. PDRAM: A Hybrid PRAM and DRAM main memory system. In *Proceedings of the 46th Annual Design Automation Conference*. 664–669.
- Xiangyu Dong, Cong Xu, Yuan Xie, and N. P. Jouppi. 2012. NVSim: A circuit-level performance, energy, and area model for emerging nonvolatile memory. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 31, 7.
- Jingtong Hu, Wei-Che Tseng, C. J. Xue, Qingfeng Zhuge, Yingchao Zhao, and E. H.-M. Sha. 2011. Write activity minimization for nonvolatile main memory via scheduling and recomputation. *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.* 30, 4, 584–592.
- Yazhi Huang, Tiantian Liu, and C. J. Xue. 2011. Register allocation for write activity minimization on non-volatile main memory. In *Proceedings of 16th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 129–134.
- Ibrahim Hur and Calvin Lin. 2004. Adaptive history-based memory schedulers. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*.
- Ibrahim Hur and Calvin Lin. 2007. Memory scheduling for modern microprocessors. *ACM Trans. Comput. Syst.* 25, 4.
- Bruce Jacob, Spencer W. Ng, and David T. Wang. 2007. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publisher.
- Lei Jiang, Bo Zhao, Jun Yang, and Youtao Zhang. 2014. A low power and reliable charge pump design for Phase Change Memories. In *Proceedings of ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 397–408.
- Dongki Kim, Sungkwang Lee, Jaewoong Chung, Dae Hyun Kim, Dong Hyuk Woo, Sungjoo Yoo, and Sunggu Lee. 2012a. Hybrid DRAM/PRAM-based main memory for single-chip CPU/GPU. In *Proceedings of 49th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 888–896.
- Yoongu Kim, Dongsu Han, Onur Mutlu, and Mor Harchol-Balter. 2010a. ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture*.
- Yoongu Kim, Michael Papamichael, Onur Mutlu, and Mor Harchol-Balter. 2010b. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 43rd Annual IEEE/ACM International Symposium on Microarchitecture*.
- Youngsik Kim, Sungjoo Yoo, and Sunggu Lee. 2012b. Write performance improvement by hiding R drift latency in phase-change RAM. In *Proceedings of 49th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 897–906.
- Suknam Kwon, Dongki Kim, Youngsik Kim, Sungjoo Yoo, and Sunggu Lee. 2012. A case study on the application of real phase-change RAM to main memory subsystems. In *Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE)*. 264–267.
- Chih-Yen Lai, Gung-Yu Pan, Hsien-Kai Kuo, and Jing-Yang Jou. 2014. A read-write aware DRAM scheduling for power reduction in multi-core systems. In *Proceedings of 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 604–609.
- Chung Lam. 2008. Cell design considerations for phase change memory as a universal memory. In *Proceedings of International Conference on VLSI Technology, Systems and Applications*. 132–133.
- Benjamin C. Lee, Engin Ipek, Onur Mutlu, and Doug Burger. 2009. Architecting phase change memory as a scalable DRAM alternative. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*.
- Chang Joo Lee, Eiman Ebrahimi, Veynu Narasiman, Onur Mutlu, and Yale N. Patt. 2010a. DRAM-aware last-level cache replacement. HPS Tech. Rep. TR-HPS-2010-007.
- Chang Joo Lee, Veynu Narasiman, Eiman Ebrahimi, Onur Mutlu, and Yale N. Patt. 2010b. DRAM-aware last-level cache writeback: Reducing write-caused interference in memory systems. HPS Tech. Rep. TR-HPS-2010-002.
- Hsien-Hsin S. Lee, Gary S. Tyson, and Matthew K. Farrens. 2000. Eager writeback—A technique for improving bandwidth utilization. In *Proceedings of the 33rd Annual ACM/IEEE International Symposium on Microarchitecture*.

- Bing Li, ShuChang Shan, Yu Hu, and Xiaowei Li. 2014. Partial-SET: Write speedup of PCM main memory. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 1–4.
- Jiayin Li and K. Mohanram. 2014. Write-once-memory-code phase change memory. In *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*. 1–6.
- John D. McCalpin. 1995. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*.
- Micron-DDR3. 4Gb DDR3 SDRAM, MT41J512M8-64Megx8x8 banks. <http://www.micron.com/products/dram/ddr3-sdram>.
- Micron-LPDDR2. 4Gb LPDDR2 SDRAM, MT42L256M16D1-32Megx16x8 banks. <http://www.micron.com/products/dram/mobile-lpddram>.
- Azalia Mirhoseini, Miodrag Potkonjak, and Farinaz Koushanfar. 2012. Coding-based energy minimization for phase change memory. In *Proceedings of the 49th Annual Design Automation Conference*. 68–76.
- Onur Mutlu and Thomas Moscibroda. 2007. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*.
- Onur Mutlu and Thomas Moscibroda. 2008. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*.
- Chitra Natarajan, Bruce Christenson, and Fayé Briggs. 2004. A study of performance impact of memory controller features in multiprocessor server environment. In *Proceedings of the 3rd Workshop on Memory Performance Issues*.
- Dimin Niu, Yibo Chen, Xiangyu Dong, and Yuan Xie. 2010. Energy and performance driven circuit design for emerging Phase-Change Memory. In *Proceedings of 15th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 193–198.
- Behrooz Parhami. 2010. *Computer Arithmetic: Algorithms and Hardware Designs, 2nd Ed.* Oxford University Press.
- Hyunsun Park, Sungjoo Yoo, and Sunggu Lee. 2011. Power management of hybrid DRAM/PRAM-based main memory. In *Proceedings of 48th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 59–64.
- Matt Poremba and Yuan Xie. 2012. NVMain: An architectural-level main memory simulator for emerging non-volatile memories. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.
- Moinuddin K. Qureshi, Michele M. Franceschini, and Luis A. Lastras-montao. 2010. Improving read performance of phase change memories via write cancellation and write pausing. In *Proceedings of the IEEE 16th International Symposium on High Performance Computer Architecture*.
- Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*.
- S. Raoux, G. W. Burr, M. J. Breitwisch, C. T. Rettner, Y.-C. Chen, R. M. Shelby, M. Salinga, D. Krebs, S.-H. Chen, H.-L. Lung, and C. H. Lam. 2008. Phase-change random access memory: A scalable technology. *IBM J. Res. Dev.* 52, 4, 465–479.
- Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. 2000. Memory access scheduling. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*.
- Roberto Rodriguez-Rodriguez, Fernando Castro, Daniel Chaver, Luis Pinuel, and Francisco Tirado. 2013. Reducing writes in Phase-Change Memory environments by using efficient cache replacement policies. In *Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE)*. 93–96.
- Jun Shao and Brian T. Davis. 2007. A burst scheduling access reordering mechanism. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture*.
- Allan Snaveley and Dean M. Tullsen. 2000. Symbiotic job scheduling for a simultaneous multithreaded processor. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*.
- Jeffrey Stuecheli, Dimitris Kaseridis, David Daly, Hillery C. Hunter, and Lizy K. John. 2010. The virtual write queue: Coordinating DRAM and last-level cache policies. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*.
- Guangyu Sun, Dimin Niu, Jin Ouyang, and Yuan Xie. 2011. A frequent-value based PRAM memory architecture. In *Proceedings of 16th Asia and South Pacific Design Automation Conference (ASP-DAC)*. 211–216.
- Zhe Wang, Samira M. Khan, and Daniel A. Jimnez. 2012. Improving writeback efficiency with decoupled last-write prediction. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*.
- Jianhui Yue and Yifeng Zhu. 2013. Exploiting subarrays inside a bank to improve phase change memory performance. In *Proceedings of Design, Automation Test in Europe Conference Exhibition (DATE)*. 386–391.

- Guangfei Zhang, Huandong Wang, Xinke Chen, Shuai Huang, and Peng Li. 2012. Heterogeneous multi-channel: Fine-grained DRAM control for both system performance and power efficiency. In *Proceedings of 49th ACM/EDAC/IEEE Design Automation Conference (DAC)*. 876–881.
- Wangyuan Zhang and Tao Li. 2009. Exploring phase change memory and 3D die-stacking for power/thermal friendly, fast and durable memory architectures. In *Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques*.
- XianWei Zhang, Le Jang, Youao Zhang, Chuanjun Zhang, and Jun Yang. 2013. WoM-SET: Low power proactive-SET-based PCM write using WoM code. In *Proceedings of IEEE International Symposium on Low Power Electronics and Design (ISLPED)*. 217–222.
- Miao Zhou, Yu Du, Bruce Childers, Rami Melhem, and Daniel Mossé. 2012. Writeback-aware partitioning and replacement for last-level caches in phase change main memory systems. *ACM Trans. Archit. Code Optim., Special Issue on High-Performance and Embedded Architectures and Compilers*.
- Ping Zhou, Bo Zhao, Jun Yang, and Youtao Zhang. 2009. A durable and energy efficient main memory using phase change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*.

Received October 2014; revised February 2015; accepted March 2015