High Performance Memory Access Scheduling Using Compute-Phase Prediction and Writeback-Refresh Overlap

Yasuo Ishii†‡ Kouhei Hosokawa† Mary Inaba† Kei Hiraki†
The University of Tokyo, 7-3-1, Hongo Bunkyo-ku, Tokyo, Japan †
NEC Corporation, 1-10, Nisshin-cho, Fuchu-shi, Tokyo, Japan ‡
{yishii, khosokawa, mary, hiraki}@is.s.u-tokyo.ac.jp

ABSTRACT

In this paper, we propose two novel memory access scheduling algorithms: (1) Compute-Phase Prediction and (2) Writeback-Refresh Overlap. Compute-Phase Prediction is a fine-grain thread-priority prediction technique. It estimates the execution phase of the thread, whether compute-intensive or memory-intensive with fine granularity, and gives higher priority to the read requests from the thread in the compute-intensive phase. Writeback-Refresh Overlap issues pending write commands and a refresh command of a rank of multi-rank DRAM system at a time, so that a rank of DRAM is refreshed while the memory bus is occupied by the write requests of the other ranks. This eliminates the idle time of the memory bus on a multi-rank DRAM system because the memory controller issues write requests for the rank that is not refreshing during the time the other rank is refreshing.

We implement both features on an optimized memory access controller, which uses a 2469B budget. We evaluate the optimized memory controller using the memory scheduling championship framework. The optimized memory controller improves the execution time by 7.3%, the energy-delay product by 13.6% and the performance-fairness product by 12.2% over the baseline memory controller.

1. INTRODUCTION

As the semiconductor process technology has been improved, the performance gap between the processor chip and the DRAM memory system has increased. Modern multi-core processors generate multiple memory access sequences that have different processing context from each other. The memory controllers handle such complicated memory accesses with avoiding starvation, saving power consumption, and minimizing the overall running time of all processors. To achieve this goal, the memory controller should treat each thread differently depending on its executing phase. Existing work [3, 4] shows that it is effective to give a higher priority to the read commands of the compute-intensive phase, in which the corresponding thread mainly executes the calculation and issues few memory requests. Besides the executing phase, DRAM refreshing is the important problem of the modern memory controllers because the refreshing penalty increases as the DRAM density increases. Elastic Refreshing [6] shows that a reduction of the DRAM refreshing penalty can improve the system throughput.

Based on these previous works, we focus on two main research themes: Computation-Phase Prediction and Writeback-Refresh Overlap. First, Compute-Phase Prediction determines the thread-priority with fine-granularity based on the memory access frequency. When the execution phase of a thread switches from compute-intensive phase to memory-intensive phase, Compute-Phase Prediction detects the transition immediately and changes the thread priority. Compute-Phase Prediction improves the timeliness of the phase transition detection, while the existing methods take millions of cycles. Second, Writeback-Refresh Overlap issues pending write commands stored in the write queue while a rank is in a refreshing. Previously, the refresh commands are issued to all ranks simultaneously, so the bandwidth is wastefully left idle in the refreshing period. Writeback-Refresh Overlap strategy can utilize such a wasteful bandwidth, leading to the effective processing of the write queue.

This paper is organized as follows. We describe the problem descriptions in Section 2. In Section 3, we propose Compute-Phase Prediction and Writeback-Refresh Overlap. In Section 4, we introduce a detail design of the memory controller for the memory scheduling championship. In Section 5, we evaluate our optimized memory controller. Finally, in Section 6, we conclude this paper.

2. PROBLEM DESCRIPTION

2.1 Thread-Priority Control

First-Ready First-Come-First-Serve (FR-FCFS) [5] memory scheduling exploits row-buffer locality by prioritizing the memory requests that access an already activated row. FR-FCFS reduces the average memory latency because row-hit memory access reduces the latency of the activations. However, FR-FCFS causes unfairness on multi-core processors because it always prioritizes row-hit memory access.

To improve the fairness and the system throughput, the memory controller should prioritize the requests from a specified thread. Generally, the memory controller gives high priority to the request from a thread in the compute-intensive phase, in which a thread issues few memory requests, because the progress per memory request from such thread is larger than that from the thread in the memory-intensive phase. When the memory controller prioritizes the requests belonging to the compute-intensive phase, the system throughput and the fairness are improved. However, it requires fine-grain detection of the execution phase transition to improve the performance because the phase frequently changes between the compute-intensive and memory-intensive. The memory controller cannot give appropriate priority without the timeliness, which even worsen the performance. Unfortunately, existing

Table 1: Refresh penalty for each DRAM capacity

	1G bit	2G bit	4G bit
Refresh cycle (tRFC)	110 ns	160 ns	260 ns
Refresh interval (tREFI)	7800 ns	7800 ns	7800 ns
Penalty (tRFC / tREFI)	1.4 %	2.1 %	3.3 %

thread-priority schemes cannot detect the change of the execution phase with fine granularity. For example, ATLAS [3] and Thread Cluster Memory Scheduling [4] determine the thread-priority on the boundary of each quantum, which typically takes millions of cycles.

2.2 Penalty of DRAM Refresh

As the density of the DRAM cells increases, the refresh cycle time (tRFC) also increases. Table 1 shows the relationship between the DRAM capacity and the refresh cycle time. On the latest DRAM device, the penalty of the refresh is 3.3% of the refresh interval time (tREFI). This means that the modern memory system wastes 3.3% of available memory bandwidth. Unfortunately, many studies of DRAM controllers have not considered the refresh penalty because its performance impact was not large. Therefore, most of the previous memory controllers simply issue the refresh command when the refresh interval timer has expired. However, this prevents memory controller from issuing the memory access during the DRAM refreshing.

Elastic Refresh [6] mitigates the refresh penalty by issuing the refresh command when the memory bus is idle. However, this scheme cannot mitigate the penalty while the memory bus is not idle. Therefore, such existing work cannot resolve the problem described in this section.

3. SCHEDULING ALGORITHMS

In this section, we propose two novel memory access scheduling algorithms: Compute-Phase Prediction and Writeback-Refresh Overlap.

3.1 Compute-Phase Prediction

Compute-Phase Prediction is a fine-grain thread-priority prediction technique. It improves the timeliness and the accuracy of the thread-priority control of Thread Cluster Memory Scheduling. It categorizes the execution phase of each thread into one of two types, the compute-intensive phase and the memory-intensive phase. While Thread Cluster Memory Scheduling categorizes the workload based on the memory traffic of L2 cache miss and the consumption of memory bandwidth, Compute-Phase Prediction uses the committed instruction count at the occurrence of a cache miss to estimate the execution phase.

Compute-Phase Prediction uses two saturation counters, the distance counter and the interval counter, to estimate the execution phase. The interval counter counts the number of executed instructions from previous cache misses; therefore this counter is incremented on the commit stage of the processor pipeline. When the interval counter reaches the maximum value ($Max_{interval}$), Compute-Phase Prediction assumes that the thread is in the compute-intensive phase. The distance counter measures the amount of the off-chip memory traffic. With heavy off-chip memory traffic, the corresponding thread is assumed to be in the memory-intensive phase. The distance counter is incremented when

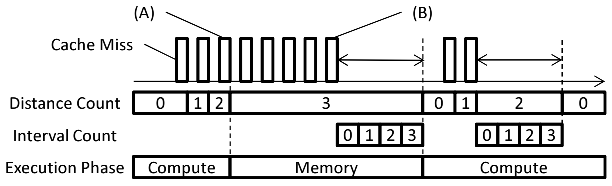

Figure 1: Overview of Compute-Phase Prediction. $Max_{distance} = Max_{interval} = 3$ **in this example.**

the cache miss is observed on the private L2 cache and cleared when the interval counter is saturated. When the distance counter reaches the maximum value ($Max_{distance}$), Compute-Phase Prediction assumes that the thread is in the memory-intensive phase. Each counter counts the executed instructions instead of the number of processor cycles that is easily affected on the execution condition. To calculate the interval counter and the distance counter on each memory controller, the processor cores attach the committed instruction count for the memory requests. This reduces the communication cost of the run-time information among all memory channels that is required in Thread Cluster Memory scheduling.

Figure 1 shows an example of the prediction. We assume both $Max_{interval}$ and $Max_{distance}$ are 3 in this example. After the third cache miss is observed in Figure 1(A), the corresponding thread is treated as the memory-intensive phase because the amount of the off-chip memory traffic (L2 cache misses) exceeds the threshold. After the series of the cache misses is finished in Figure 1(B), the interval counter starts to count. When the interval counter achieves $Max_{interval}$, the corresponding thread is treated as the compute-intensive phase because enough instructions are executed without heavy off-chip memory traffic.

We compare Compute-Phase Prediction with Thread Cluster Memory Scheduling, which is the latest memory access scheduling strategy. Figure 2 shows the memory traffic of Blackscholes. Each plot is the memory traffic per 1000 instructions. In Figure 2, the background color of the compute-intensive phase is gray and the background color of the memory-intensive phase is white. Thread Cluster Memory Scheduling cannot detect several memory-intensive phases, as shown in Figure 2(A), and several compute-intensive phases, as shown in Figure 2(B), because its prediction result is not updated with fine granularity. On the other hand, Compute-Phase Prediction categorizes those parts appropriately (the detailed parameters are shown in Section 5) because its algorithm uses the instruction count distance of the cache misses.

3.2 Writeback-Refresh Overlap

As described in previous section, all memory commands are stalled during the DRAM refresh, when the memory controller issues the refresh commands for all ranks simultaneously. Figure 3(A) shows a typical refresh situation; the memory controller issues the refresh command when the tREFI timer is expired.[1]

[1]We assume that the refresh commands for multiple ranks are issued simultaneously because this policy is simple and reasonable. The memory controllers are allowed to issue refresh commands rank by rank, but this often increases the stall time of the read requests to access the refreshing rank.

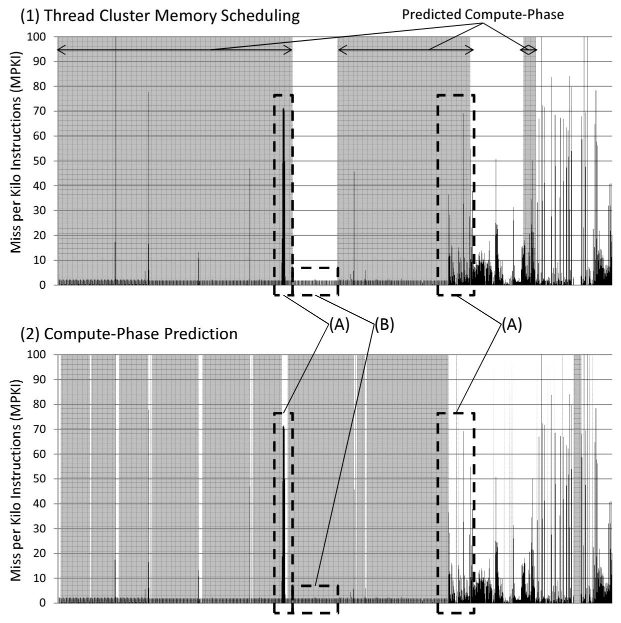

Figure 2: Phase prediction results for Blackscholes. The background of the compute-intensive phase is gray and the background of the memory-intensive phase is white.

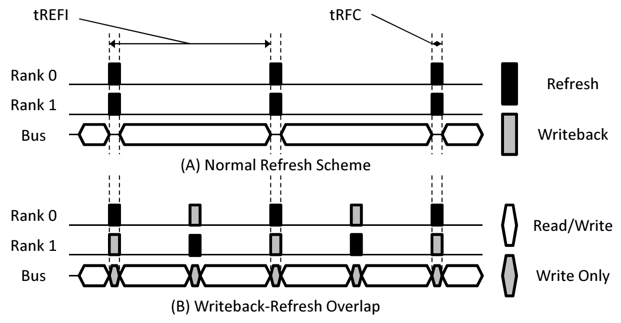

Figure 3: Overview of Writeback-Refresh Overlap

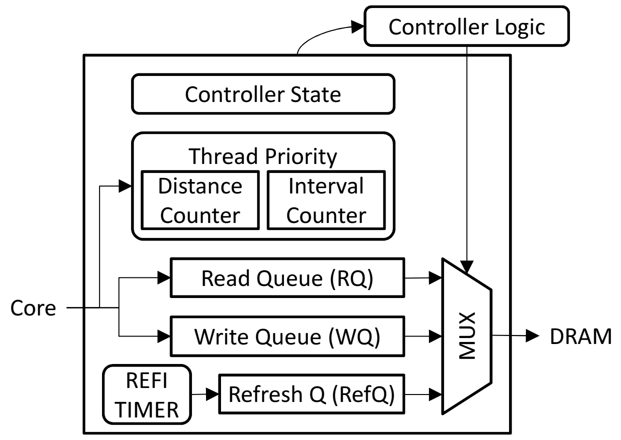

Figure 4: Overview of the optimized controller

To mitigate the refresh penalty of modern DRAM devices, we propose Writeback-Refresh Overlap. Figure 3(B) shows the refresh strategy of Writeback-Refresh Overlap; the memory controller schedules the refresh commands, overlapped with the write commands. To saturate the limited memory bus, Writeback-Refresh Overlap holds many write commands on the command queue in order to find row-buffer locality from pending memory requests for the rank that is not refreshing. To retain pending write commands effectively, the memory controller with Writeback-Refresh Overlap prioritizes the issuing of the write commands for the rank that is the next refresh target. On Writeback-Refresh Overlap, the refresh commands are overlapped with write commands instead of read commands. This is because the pending read commands cause the waiting processor to stall, whereas the write commands do not cause processor stall because the processor cores do not need to wait for the reply. For this characteristic, write requests can be delayed for the sake of committing read requests, and they are flushed at the next refresh period. The implementation of Writeback-Refresh Overlap uses the extended memory controller state for coarse-grain scheduling control and priority policy to control the pending write commands.

Writeback-Refresh Overlap can collaborate with the other techniques. When the memory controller uses much larger write buffer by using virtual write queue feature [7], the memory controller utilizes the limited memory bus much more efficiently because the memory controller finds the row-buffer locality from the much larger write queue. Even if there are not enough pending write commands, the prefetch requests can also be utilized to saturate the memory bus. As shown in the other study [2], stream prefetching is an efficient way to improve performance.

4. IMPLEMENTATION

We implement Compute-Phase Prediction and Writeback-Refresh Overlap on the optimized memory controller. Figure 4 shows a block diagram of the controller. The memory controller has several states to determine coarse-grain scheduling policy. The memory controller also employs several queues to store the memory requests from each processor core. The actual memory requests are pushed to the read command queue (RQ) and the write command queue (WQ). For Compute-Phase Prediction, the number of the committed instructions is attached to the memory requests. This additional information is stored in the thread-priority control block. The Refresh command queue (RefQ) holds the refresh commands that are scheduled in the next refresh timing window. The refresh commands are pushed to the refresh queue when the refresh interval timer has expired. The controller logic uses the information stored in the controller to determine the next issuing command.

4.1 Controller State

In Writeback-Refresh Overlap, the memory controller has to saturate the memory bus by writing data during the DRAM refresh. To saturate the memory bus, the memory controller has to hold enough pending write commands in the write command queue. To realize this feature, the memory controller employs four internal states: READ, WRITE, BEFORE_REFRESH, and REFRESH. The state diagram is shown in Figure 5. On READ, the controller schedules read requests to minimize the stall time of the processor cores.

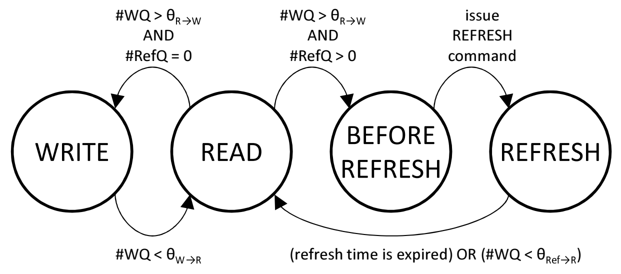
#WQ > $\theta_{R\to W}$ AND #RefQ = 0
#WQ > $\theta_{R\to W}$ AND #RefQ > 0
issue REFRESH command
WRITE
READ
BEFORE REFRESH
REFRESH
#WQ < $\theta_{W\to R}$
(refresh time is expired) OR (#WQ < $\theta_{Ref\to R}$)

Figure 5: State diagram of the memory controller

On WRITE, the controller schedules write requests so that they do not spill over to the write command queue. As described in the previous work [7], the controller divides the read phase and the write phase to minimize the turnaround time (tWTR penalty) of the bus. On BEFORE_REFRESH, the controller stops issuing memory requests for the rank, which is the next refresh target. On REFRESH, the controller issues the pending write commands to a rank that is not refreshing.

The state transition from READ to BEFORE_REFRESH or WRITE is performed when the number of pending write commands in the write command queue is over the specified threshold ($\theta_{R\to W}$). The state transits to BEFORE_REFRESH when the controller has at least one scheduled refresh, and when the number of pending write commands to the rank that is not the target of the next refresh occupies more than a half of the write command queue. Otherwise, the controller state simply transits to WRITE. Once the state transits to BEFORE_REFRESH, the controller stops all memory requests for the rank of the refresh target. When the refresh conditions for the rank of the refresh target are met, the controller issues a refresh command and transits to REFRESH. When the amount of the pending write commands becomes smaller than the specified threshold ($\theta_{W\to R}, \theta_{Ref\to R}$), REFRESH and WRITE are changed to READ. The state transition from REFRESH to READ also occurs when the refresh cycle time (tRFC) has expired.

4.2 Priority of Memory Requests

The controller categorizes the memory requests into two priority levels. The priority policies are different between the read requests and the write requests. We first describe priority strategy for read requests, and then describe that for write requests. To track the priority level, each read command has three additional flags: the timeout flag, the priority-level flag, and the row-buffer locality flag. These flags are attached to all read / write commands.

4.2.1 Read Requests

Our memory controller determines the priority of pending request commands based on the thread-priority by using Compute-Phase Prediction. Each processor core attaches the number of committed instructions in order to determine the execution phase (memory-intensive phase or compute-intensive phase) of the corresponding thread. When the read requests arrive at the memory controller, they compare their addresses with the addresses of the other pending commands and mark the row-buffer locality flags on the commands that are accessing the same row. This address check is mainly performed for the merging of read commands and for the forwarding of write data. The memory controller also updates the priority of the existing memory requests in the request queues.

We also add extra priority to the read commands that have low memory-level-parallelism (MLP). When the number of the read commands in one memory controller becomes less than the threshold (θ_{MLP}), the memory controller increases the priority of these memory read commands like Minimalist [2]. As well as the MLP, the pending requests are promoted to priority requests when the memory requests stay in the read command queue for more than $\theta_{priority}$ cycles. This is to avoid the starvation of the corresponding processor core. When a read request stays the command queue for a long time ($\theta_{timeout}$ cycles), the controller gives maximum priority to the requests.

When the read command that belongs to the compute-intensive phase is pushed to the read command queue, all read commands that come from the same thread are promoted to priority requests, in order to minimize memory access latency. Otherwise, the priority read commands are not updated.

4.2.2 Write Requests

The priority of the write command is different from that of the read commands, because the latency of the write requests does not affect the execution time. We try to control the row-buffer locality through prioritizing the write commands for the rank that is the next refresh target. This scheduling policy increases the pending write commands for the other rank that is not the next refresh target. This helps to increase the row-hit write commands during the next DRAM refresh. Moreover, the memory controller modifies the priority by using the controller state. On READ, the controller gives higher priority to the write commands whose row-buffer locality flag is not set, because it is trying to schedule read access. Thus, the controller saves the row-hit commands that occupy the memory bus for a long time (typically tens of cycles).

This helps to increase the number of the pending write commands to the rank that is not the target of the next refresh. During REFRESH, the available memory bandwidth is mainly restricted by the tFAW and tRRD. Therefore, the controller has to exploit row-buffer locality to improve the available memory bandwidth. On WRITE, the controller gives higher priority to the write commands with a row-buffer locality flag, in order to maximize the write traffic. This priority is determined by the number of requests that access the same row. The threshold of the priority is determined by the number of activations that are issued during the current tFAW timing window.

4.3 Request Scheduling

We describe the scheduling policy of the memory requests. Our memory controller is optimized to schedule read / write requests from each core. To support optimized memory scheduling, the memory controller also issues other commands (refresh, precharge, and autoprecharge).

4.3.1 Read / Write Commands

The memory controller determines which command to issue, based on the memory controller state and the request priority. The overall rule for priority control is shown in Table 2. On READ, the read commands are first priori-

Table 2: Scheduling priority policy. The old requests are prioritized within the same priority level.

	(A) READ STATE		(B) OTHER STATE		
	Priority Req.	Normal Req.	Priority Req.	Normal Req.	Comments
Timeout Read Request	1 (Highest)	1 (Highest)	2	2	Request lifetime > $\theta_{timeout}$
Low-MLP Read Request	2	2	3	3	#RQ of the thread < θ_{MLP}
Read Data (Row-Hit)	3	6	4	5	Row-hit read requests
Read Activate / Precharge	4	7	6	Not Issue	
Write Data (Row-Hit)	5	5	1 (Highest)	1 (Highest)	Row-hit write requests
Write Activate / Precharge	8	9 (Lowest)	7	8 (Lowest)	

Table 3: Parameters of the optimized controller

$\theta_{R \to W}$	(WQ size * 3/4)
$\theta_{W \to R}$	(WQ size * 1/2) − 6
$\theta_{Ref \to R}$	(WQ size * 1/4) + 2
$Max_{distance}$	13 memory requests
$Max_{interval}$ (compute-intensive)	220 instructions
$Max_{interval}$ (memory-intensive)	970 instructions
θ_{MLP}	2 requests
$\theta_{priority}$	100,000 cpu cycles
$\theta_{timeout}$	1,000,000 cpu cycles

tized. In the other states (WRITE, BEFORE_REFRESH, and REFRESH), the write commands are first prioritized. "Not Issue" indicates that the corresponding requests are never issued in the corresponding controller state. Activate / precharge commands with normal priority for the next refresh target rank are prioritized over the requests for the other ranks.

4.3.2 Refresh Commands

Refresh commands are issued in the following two cases: the state is BEFORE_REFRESH, and the read command queue is empty. On BEFORE_REFRESH, a refresh command is immediately issued if all conditions to issue the refresh command are met. When the read command queue is empty, the memory controller schedules the refresh commands, such as Elastic Refresh [6]. The refresh command is issued after the idle delay time, which is proportional to the number of the scheduled refresh commands. During the refresh, the other rank drains the pending write commands. These scheduling priorities are higher than the normal read / write command. After the refresh command is finished, the next refresh is scheduled for the other rank. When the refresh commands have not been issued before the deadline, the memory controller stops all operations and issues refresh commands (force refresh mode).

4.3.3 Precharge Commands

The memory controller also issues precharge requests to the already opened row. The aggressive precharge command is issued when the corresponding row meets the following conditions: there are no pending read / write commands to the row and the corresponding rank allows at least one activation command in the current tFAW timing window. The memory controller issues autoprecharge commands with read / write commands when the scheduled read / write commands do not have other read / write commands that can access the same row. To detect such requests, the row-buffer locality flag is used. The flag is set when the successive read / write commands have arrived at the memory controller. This is feasible because the incoming command checks the addresses of all read / write commands that have already been stored in the read / write queue to support read-merging and write-data forwarding.

Table 4: Hardware budget count. Total hardware budget count is 2469B for 16-core, 4-channel system. We assume 160 read requests per core and 96 write requests per channel for our evaluation.

(A) Per-Request

Resource	Budget
Row-hit Status	1-bit
Priority Level	1-bit
Timeout Status	1-bit
Per-Request	3-bit
Total Budget	1104B

(B) Per-Channel

Resource	Budget
Controller State	2-bit
Last access bank	3-bit
Last access rank	1-bit
Refresh Control	18B
tFAW Tracking	64.5B
Core Tracking	258B
Per-Channel	341.25B
Total Budget	1365B

5. PERFORMANCE EVALUATION

5.1 Controller Configuration

We implement our optimized controller on the memory scheduling championship framework [1]. This controller uses both Compute-Phase Prediction and Writeback-Refresh Overlap. The evaluation environment provides two configurations, which differ in the number of their memory channels. In addition, each configuration uses different address mapping, write queue size, and core configuration. We change several parameters in order to adapt to each memory configuration. Table 3 shows the basic parameters of the memory controller. Each parameter is described in the previous section. We use different $Max_{interval}$ values for the compute-intensive phase and the memory-intensive phase.

Table 4 shows the total hardware cost of the memory controller. The optimized controller attaches additional storage for each memory request and for each memory controller. The table shows the budget amount for the 4-channel configurations. The maximum number of read commands is estimated through the number of the entries in the reorder buffer. In the 4-channel configuration, we assume 160 read requests, which is the number of reorder buffer entries, and is also the maximum number of in-flight requests. The total budget is 2469B which is much less than 68KB.

5.2 Evaluation Result

We evaluate the optimized memory controller using the memory scheduling championship framework. The evalu-

Table 5: Comparison of key metrics on baseline and proposed memory controllers.

Workload	Config	Sum of exec times (10 M cyc)			Max slowdown			EDP (J.s)		
		FCFS	Close	Proposed	FCFS	Close	Proposed	FCFS	Close	Proposed
MT-canneal	1 chan	418	404	367	NA	NA	NA	4.23	3.98	3.76
MT-canneal	4 chan	179	167	156	NA	NA	NA	1.78	1.56	1.35
bl-bl-fr-fr	1 chan	149	147	137	1.20	1.18	1.10	0.50	0.48	0.42
bl-bl-fr-fr	4 chan	80	76	74	1.11	1.05	1.02	0.36	0.32	0.30
c1-c1	1 chan	83	83	78	1.12	1.11	1.05	0.41	0.40	0.36
c1-c1	4 chan	51	46	46	1.05	0.95	0.94	0.44	0.36	0.36
c1-c1-c2-c2	1 chan	242	236	213	1.48	1.46	1.33	1.52	1.44	1.23
c1-c1-c2-c2	4 chan	127	118	113	1.18	1.10	1.06	1.00	0.85	0.78
c2	1 chan	44	43	42	NA	NA	NA	0.38	0.37	0.34
c2	4 chan	30	27	27	NA	NA	NA	0.50	0.39	0.39
fa-fa-fe-fe	1 chan	228	224	200	1.52	1.48	1.33	1.19	1.14	0.92
fa-fa-fe-fe	4 chan	106	99	92	1.22	1.15	1.06	0.64	0.56	0.49
fl-fl-sw-sw-c2-c2-fe-fe	4 chan	295	279	257	1.40	1.31	1.21	2.14	1.88	1.58
fl-fl-sw-sw-c2-c2-fe-fe--bl-bl-fr-fr-c1-c1-st-st	4 chan	651	620	579	1.90	1.80	1.64	5.31	4.76	4.06
fl-sw-c2-c2	1 chan	249	244	220	1.48	1.43	1.26	1.52	1.44	1.14
fl-sw-c2-c2	4 chan	130	121	117	1.13	1.06	1.02	0.99	0.83	0.77
st-st-st-st	1 chan	162	159	147	1.28	1.25	1.16	0.58	0.56	0.48
st-st-st-st	4 chan	86	81	78	1.14	1.08	1.04	0.39	0.35	0.33
Overall		3312	3173	2941	1.30 PFP: 3438	1.24 PFP: 3149	1.16 PFP: 2721	23.88	21.70	19.06

ation result is shown in Table 5. As shown in the table, our optimized memory controller consistently outperforms the other scheduling algorithms. It reduces the system execution time by 7.3% over closed page policy, which shows the best performance in the baseline. Our controller also improves the performance-fairness product (PFP) by 13.6% and the energy-delay product (EDP) by 12.2%.

6. CONCLUDING REMARKS

We have proposed two new memory scheduling algorithms to improve system throughput. Proposed methods are Compute-Phase Prediction and Writeback-Refresh Overlap. Compute-Phase Prediction utilizes thread behavior to prioritize requests that belong to the compute-intensive phase. As shown in Figure 2, Compute-Phase Prediction estimates the execution phase of the thread more accurately than existing prediction algorithms. The proposed prediction algorithm adapts to the execution phase of the thread with fine granularity, and the memory controller prioritizes the appropriate access to the memory. We also propose Writeback-Refresh Overlap. We schedule DRAM refresh commands that can be overlapped with pending write commands. This improves the efficiency of the bandwidth on the memory bus. Writeback-Refresh Overlap reduces the penalty of the refresh in high density DRAM devices. These two methods are implemented in our optimized memory controller.

We evaluated the optimized controller through simulations using the memory scheduling championship framework. Experimental results show that our optimized controller improves the execution time by 7.3%, the energy-delay product by 13.6%, and the performance-fairness product by 12.2%. Compute-Phase Prediction and Writeback-Refresh Overlap can be also applied to the other scheduling policies. Exploring more efficient scheduling algorithms is our future work.

7. REFERENCES

[1] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. Pugsley, A. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti. USIMM: the Utah SImulated Memory Module. Technical report, University of Utah, 2012. UUCS-12-002.

[2] D. Kaseridis, J. Stuecheli, and L. K. John. Minimalist open-page: a dram page-mode scheduling policy for the many-core era. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44 '11, pages 24–35, New York, NY, USA, 2011. ACM.

[3] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *HPCA'10*, pages 1–12, 2010.

[4] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 65–76, Washington, DC, USA, 2010. IEEE Computer Society.

[5] S. Rixner, W. J. Dally, U. J. Kapasi, P. Mattson, and J. D. Owens. Memory access scheduling. In *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, pages 128–138, New York, NY, USA, 2000. ACM.

[6] J. Stuecheli, D. Kaseridis, H. C.Hunter, and L. K. John. Elastic refresh: Techniques to mitigate refresh penalties in high density memory. In *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, pages 375–384, Washington, DC, USA, 2010. IEEE Computer Society.

[7] J. Stuecheli, D. Kaseridis, D. Daly, H. C. Hunter, and L. K. John. The virtual write queue: coordinating dram and last-level cache policies. In *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, pages 72–82, New York, NY, USA, 2010. ACM.