Pre-Read and Write-Leak Memory Scheduling Algorithm

Long Chen Yanan Cao Sarah Kabala Parijat Shukla
{longc,yanan,skabala,parijats}@iastate.edu
Iowa State University

Abstract—Main memory system latency is a major bottleneck for system performance and memory reads are on the critical instruction path. Typically, memory access schedulers prioritize read requests over write requests and eventually enter a drain-write mode to process a batch of waiting writes. Conventional memory scheduling algorithms strictly separate read mode from drain-write mode, and no parallelism is exploited between the two modes. However, parallelism is available when there are no bank conflicts between pending read and write requests and the command bus is idle. To reclaim this unexploited parallelism, we propose a pre-read write-leak scheduling scheme to reduce the frequency of entering drain-write mode. In order to mitigate the impact of bus turnaround time as reads and writes are interleaved, we further propose two methods to elaborately select the read and write requests to be issued during the other mode. Our simulation results show that with the proposed read-write interleaving schemes, EDP is reduced by 18.4% and 17.3% compared with typical FCFS scheduling algorithm. Additionally, our methods reduce PFP by up to 24.3% and 22.8% respectively.

1 Introduction

Contemporary multi-core processor systems demand high bandwidth and low latency for data transactions with main memory. Unfortunately, memory access latency and throughput have been major performance bottlenecks for decades [3]. Processor frequencies and core-counts continue to rise in the multi-core era; yet, the rate of improvement to key features of memory system performance lags behind. This widening gap exacerbates the memory bottleneck already hindering performance in many of today's systems. Although main memory technology has been improved, the memory access latency is still hundreds of processor cycles, which can stall the execution of a thread on the processor.

Memory read latency is on the critical path of system performance. Generally, a memory read access is fulfilled by a sequence of memory commands: precharge the addressed bank, activate the addressed row into that bank's row buffer, and read the addressed column from the row buffer back to the processor. If a read access hits the currently active row buffer due to its activation by an earlier request, then the read latency is significantly reduced to only t_{CAS} (the time to access a column) and the data transfer time. A bank can precharge and activate a row in parallel with other banks. However, simultaneous read accesses to different rows within the same bank create a bank conflict and must be serviced sequentially. A sequence of different-row memory read accesses to the same bank thus requires the additional time and energy costs of precharging the bank, which closes its currently open row, and then activating the new row into the bank's row buffer.

Servicing a memory write may stall a number of memory reads and significantly increase the memory read latency. Because memory writes are not on the critical path of system performance, they are often delayed to be processed when the queue of pending write requests reaches a certain size called the high-water mark. Generally, during the drain-write mode, memory write accesses are processed aggressively and memory reads are completely stalled. The drain-write mode is necessary because once the memory write queue is full, the entire processor pipeline is stalled, which harms system performance. This creates a dilemma when scheduling drain-write mode. On one hand, read requests are forbidden during drain-write mode whether the command bus is idle or not. On the other hand, write requests are not allowed when processing reads, which accumulates writes to the high-water mark frequently.

In order to break the performance cap created by the read-prohibiting use of drain-write mode, we propose *pre-read write-leak scheduling (PRWL)* to interleave memory read and write accesses and reduce the frequency of entering drain-write mode. We allow memory read commands during drain-write mode, but we elaborately select only those read accesses which could utilize the otherwise idle command bus without harming

the progress of write drainage. Similarly, we allow memory write commands during read mode. Although both exceptions introduce bus turnaround latency, this is mitigated by latency savings in row buffer hits that would miss in their native mode, yielding an overall performance improvement.

We propose two methods to limit the ratio of mode exceptions to balance their negative and positive impacts. Our simulation results show that energy-delay product (EDP) reduces by 18.4% and 17.3%, respectively, for our two schemes compared to typical FCFS scheduling. As PRWL is based on FCFS, the performance-fairness product (PFP) is also reduced by up to 24.3% and 22.8%. These improvements mainly come from the reduction of the frequency to enter drain write mode.

The rest of this paper is organized as follows. Section 2 introduces the main memory system and related work. Section 3 presents our memory scheduling algorithms and their implementation overhead. Section 4 briefly introduces the evaluation platform for DDR*x* memory. The simulation results for performance, EDP, and PFP are presented and analyzed in Section 5. Finally, Section 6 concludes the paper.

2 Background and Related Work

2.1 Memory System

The predominant memory system used in today's commodity computers is a one- to four-channel double data rate (DDR) 3 interface to one or two dual in-line memory modules (DIMM) per channel. The system may have one or more memory controllers which are either processor-integrated or found on a separate chip. A DDR3 memory bus connecting a controller and its DIMM's supports 64-bit data, 23-bit address and command, and additional clock and control signals [1].

Typically, a DIMM will have one, two or four *ranks* and they share the same data bus in one channel. A rank is further subdivided into *banks* and each bank is addressed as an array of *rows* and *columns*. A row is read in a bank by first precharging the bitlines and then activating the addressed row so that the data are pushed into the bank's *row buffer*. A following column access command can then access the row buffer to fetch or store data for a read or write memory request [5].

The memory commands—PRE (precharge), ACT (activate), and COL-RD/COL-WR (column read/write) — are issued by the memory controller (MC) as the result of a scheduling algorithm that is executed every memory cycle. The efficiency of the MC in servicing requests from read and write queues has a significant impact on memory latency and, thus, execution time for program threads that exhibit memory-bound behavior. If a scheduling algorithm favors memory-hungry threads imprudently, it may improve their execution time marginally at great cost to other threads that make very few requests. Such an algorithm is said to be *unfair*. Additional DIMM commands for power conservation and the periodic refresh of volatile memory contents are also administered by the MC's scheduling algorithm.

These conflicting performance, fairness, and power-saving concerns make memory scheduling a very difficult problem. Further complicating that problem are numerous timing constraints stemming from the physical construction of the DDR*x* memory systems [2].

2.2 Memory Scheduling Algorithms

The ideal performance improvement that can be achieved through memory scheduling will utilize the full bandwidth of its DDR3 memory channels when servicing read/write requests from the processor(s). Though full bandwidth utilization may not be achieved in practice, many algorithms have been proposed to improve the performance of the threads making memory requests by carefully ordering when those requests are serviced.

The pioneering work of Rixner et al. sought to reorder memory requests to exploit row locality for single-threaded systems [7]. Zhu and Zhang extended the considerations of memory access scheduling to address conflicts introduced by multi-threaded workloads [8]. Ipek et al. used reinforcement learning (RL) techniques to develop an adaptive scheduler that would optimize its scheduling policy configuration in response to recent workloads [4]. The application of RL to memory access scheduling shows promise as does the thread-clustering technique of Kim et al., which applies different scheduling policies to threads whose requests show more performance sensitivity to bandwidth vs. latency. The two clusters are constructed dynamically in response to the workload to produce an adaptive policy application depending on a thread's relative behavior [6].

3 Memory Scheduling Algorithm

3.1 Algorithm Overview

Given that memory reads are on the critical path and a row buffer hit has the lowest access latency, we schedule memory reads with higher priority than memory writes and row buffer hits with higher priority than other memory commands as our fundamental scheduling principle. Though read requests are strictly forbidden in conventional drain-write mode, we issue several beneficial read commands when write draining nears its end. Moreover, carefully selected write requests are allowed to execute during the processing of read requests. As the intrusion of memory writes may inflate memory read access times, two simple write-filtering schemes are proposed to limit the write ratio and balance its negative impacts with its benefits to the system.

3.2 Read-Write Interleave

The conventional FCFS memory scheduling technique strictly separates read mode from drain-write mode. As memory reads are prioritized over memory writes, the write request queue fills to a high-water mark. At that time, drain-write mode becomes active, and memory reads are deprioritized. Our technique services memory reads and minimizes the interference of memory writes. However, the two modes are no longer strictly designed so that no read or write can be issued during the other mode even if there are some idle resources. Our proposed read-write interleaving issues several memory read commands during drain-write mode when the command bus becomes idle. In addition, selected memory write requests are issued during read mode when the overall system would benefit from those exceptions.

A read-during-write-mode exception opens the row buffer for the following read-mode column access. This greatly reduces memory read latency and partially increases read-write parallelism. In addition, the leaked write requests reduce the frequency that full drain-write mode is entered and, thus, reduce the amount of time memory reads are stalled, which further improves system performance. Although this read-write interleaving will introduce some overhead in the form of bus turnaround delay, the effect is limited by two factors in our design. First, the read commands pre-issued during drain-write mode are only PRE and ACT commands, which do not incur data bus turnaround. Second, the leaked write commands during read mode are filtered to a low ratio so that their benefits remain greater than their negative impacts.

Pseudocode for the main part of our algorithm is given in Algorithm 1. The selection of pre-issued read commands and leaked write commands are further described in the following sections.

Algorithm 1 Memory Scheduling Algorithm

```
if In drain-write mode then
    (1) memory write hit
    (2) memory write PRE or ACT
    if near the end of drain-write mode then
        (3) issue doable read PRE or ACT
    end if
else
    (1) memory read hit
    (2) memory write hit
    (3) memory read PRE or ACT
    if doable write then
        issue memory write requests
    end if
end if
```

3.3 Pre-Issue Read Command

In conventional drain-write mode no read access commands are issued. This restricts the maximum parallelism possible for read and write commands. However, the memory command bus is idle if write commands are not issued during every scheduling cycle of write draining. If feasible read commands are waiting in the read queue, the system would benefit from higher utilization of the command bus. We therefore break from a strict drain-write mode design and issue several selected read commands during drain-write mode.

The pre-issued read requests and their issue time are carefully selected to avoid any negative impact to write requests so that write draining will proceed smoothly and the drain-write period will not be prolonged. First, the read commands are only issued when the write queue is almost drained to the low-water mark. Second, the chosen memory read commands are limited to PRE and ACT commands while COL-RD (column read) remains forbidden. This prohibition avoids data bus conflict and prevents data bus turnaround delay overheads during drain-write mode. In addition, the banks accessed by the chosen reads are those which have no bank conflict with remaining memory writes. Any memory read whose PRE and ACT commands are pre-issued during drain-write mode can finish immediately with their COL-RD command when the scheduler returns to read mode. In the best case, pre-issued memory read requests have a read access latency that is only t_{CAS}.

3.4 Limit Write Rate

Allowing write requests to execute during read mode can potentially impact read access latency negatively. One example is a read-write bank conflict where a write request opens a row buffer in a bank yet the next read request closes that row before the write's column access command is issued because the write column access is delayed by the data bus conflict. This introduces an extra read PRE command and the write's ACT is wasted.

In order to avoid this overhead, we limit the write rate during read mode and eliminate this detriment to system performance. We use two selection schemes to refine Algorithm 1. *Bus Prediction* anticipates future bus utilization for a number of cycles and looks for idle gaps where a write request could be serviced. *Random* avoids the complexity of modeling bus utilization and simply allows a pending write to be serviced with some probability. These schemes are described in further detail below.

3.4.1 Write-Leak Selection With Bus Prediction

Our intuitive solution is to select writes to only those banks which will not conflict with issuable reads. Memory request addresses are mapped by a fixed scheme to a

physical memory bank and row. When a column access command is issued, a bus reservation vector is updated to mark the data bus as occupied. If an idle cycle for the command bus is encountered, Bus Prediction scans the read/write queues looking for a write request that does not cause a bank conflict. If one is found, the reservation vector is checked for a free slot to service the write request. If a slot is available, the write is selected to be serviced during read mode. Otherwise, no write is leaked, leaving the command bus idle during that cycle.

This technique efficiently reduces the frequency of entering drain-write mode, and limits the negative impacts of write-leaking during the processing of read requests. One shortcoming of this method is that upcoming requests are not taken into consideration when identifying conflict-free banks and bus availability. Predicting upcoming read requests is difficult and is an area for future work.

3.4.2 Random Write-Leak Selection

As an alternative to Bus Prediction, we adopt a simple heuristic—a pseudo-random probability limits the write-rate to a reasonable level. If an idle command bus cycle is encountered, a modulo of the system clock by a leak-rate parameter determines whether or not a write will be selected for service during read mode. If the result is zero, a scan of the addresses in the read and write queues is performed as described for the Bus Prediction scheme. And the first non-conflicting write request is leaked. Otherwise, the command bus is left idle.

3.5 Design Overhead

Our proposed memory scheduling algorithm is practical and introduces virtually no overhead to the memory system. The only computational overhead is in selecting pre-issuable read commands and leakable write commands. Our selection technique is a simple comparison of read queue and write queue addresses, and those comparisons occur only when the command bus would otherwise be idle. The pseudo-random function used in our heuristic can be approximated efficiently by a modulo 2^n operation, which can be implemented with a simple shift operation. Thus, our scheduling algorithm is implementable and introduces almost no overhead in practice.

4 Experimental Methodologies

For our entry into the three competitive tracks of the 2012 JWAC Memory Scheduling Competition, we use *USIMM* [2] as our simulation platform. In this simulator, device level memory commands are issued by the memory controller based on the current channel, rank and bank status. Cache-line-interleaving address mapping is used in the four-channel configuration to increase the available channel parallelism and page-interleaving address mapping is used in the one-channel configuration to increase the available row locality. Basic power modelling is provided in the simulator following Micron's power calculation methodology. Details of the power simulation's parameters are given in [2].

Workloads		Workloads	
blblfrfr1	4C_1Ch1	blblfrfr4	4C_4Ch1
c1c11	2C_1Ch1	c1c14	2C_4Ch1
c1c1c2c21	4C_1Ch2	c1c1c2c24	4C_4Ch2
fafafefe1	4C_1Ch3	fafafefe4	4C_4Ch3
flswc2c21	4C_1Ch4	flswc2c24	4C_4Ch4
stststst1	4C_1Ch5	stststst4	4C_4Ch5
flflswswc2c2fefe4			8C_4Ch1
flflswswc2c2fefeblblfrfrc1c1stst4			16C_4Ch1

TABLE 1: Simulated Workloads

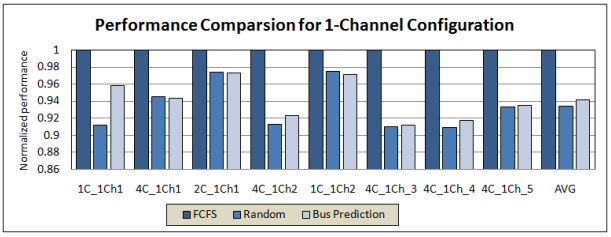


Fig. 1: Performance comparison of PRWL scheduling with FCFS in one channel memory configuration.

In our experiments, we simulate one, two, four, eight, and sixteen cores running multi-threaded or multi-programmed workloads from PARSEC and commercial transaction processing workloads. The detailed workloads are listed in Table 1. For convenience, we use nC_mChk to represent an n-core, m-channel simulation running workload k. The typical FCFS scheduling algorithm is used as a baseline, and performance (accumulated execution cycles), energy-delay product (EDP), and performance-fairness product (PFP) are measured and evaluated. We consider two variations of our algorithm:

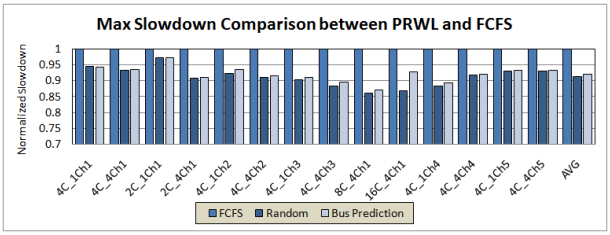
- Random—which uses a randomized write leak policy during read mode.
- Bus Prediction—which ensures that leaked writes will not conflict with any read for the data bus.

5 Experimental Results

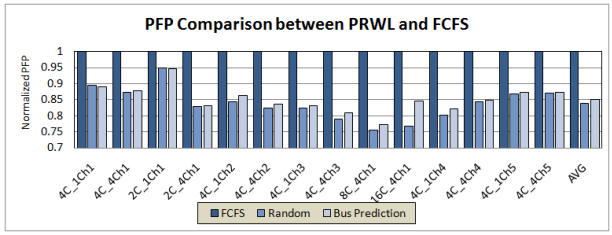
In this section, we use the metrics for the three competitive tracks provided by JWAC MSC to evaluate our proposed scheduling algorithms. Table 2 summarizes the results of our algorithms on the competition workloads.

Workload	Config	Sum of exec times (10 M cyc)			Max slowdown			EDP (J.s)		
		FCFS	Random	Bus Prediction	FCFS	Random	Bus Prediction	FCFS	Random	Bus Prediction
MT-canneal	1 chan	418	381	400	NA	NA	NA	4.23	3.60	3.93
MT-canneal	4 chan	179	160	161	NA	NA	NA	1.78	1.42	1.44
bl-bl-fr-fr	1 chan	149	141	141	1.20	1.14	1.13	0.50	0.45	0.45
bl-bl-fr-fr	4 chan	80	75	75	1.11	1.03	1.04	0.36	0.31	0.31
c1-c1	1 chan	83	81	81	1.12	1.09	1.09	0.41	0.39	0.39
c1-c1	4 chan	51	47	47	1.05	0.96	0.96	0.44	0.37	0.37
c1-c1-c2-c2	1 chan	242	221	223	1.48	1.37	1.39	1.52	1.28	1.30
c1-c1-c2-c2	4 chan	127	115	116	1.18	1.18	1.09	1.00	0.81	0.82
c2	1 chan	44	43	43	NA	NA	NA	0.38	0.37	0.36
c2	4 chan	30	27	27	NA	NA	NA	0.50	0.41	0.41
fa-fa-fe-fe	1 chan	228	208	208	1.52	1.37	1.39	1.19	0.99	1.00
fa-fa-fe-fe	4 chan	106	95	96	1.22	1.08	1.09	0.64	0.51	0.52
fl-fl-sw-sw-c2-c2-fe-fe	4 chan	295	259	261	1.40	1.20	1.22	2.14	1.62	1.65
fl-fl-sw-sw-c2-c2-fe-fe--bl-bl-fr-fr-c1-c1-st-st	4 chan	651	576	594	1.90	1.65	1.77	5.31	4.13	4.37
fl-sw-c2-c2	1 chan	249	227	229	1.48	1.31	1.32	1.52	1.23	1.25
fl-sw-c2-c2	4 chan	130	119	120	1.13	1.04	1.04	0.99	0.81	0.81
st-st-st-st	1 chan	162	151	152	1.28	1.19	1.19	0.58	0.51	0.51
st-st-st-st	4 chan	86	80	80	1.14	1.07	1.07	0.39	0.34	0.34
Overall		3312	3005	3054	1.30 PFP: 3438	1.18 PFP: 2835	1.20 PFP: 2903	23.88	19.54	20.25

TABLE 2: Key metrics for the FCFS baseline and our proposed schedulers Random and Bus Prediction. c1 and c2 represent commercial transaction-processing workloads, MT-canneal is a 4-threaded version of canneal, and the other workload abbreviations are single-threaded PARSEC programs.



(a) Maximum Slowdown Comparison.



(b) PFP Comparison.

Fig. 3: Comparison of the maximum slowdown and PFP between PRWL and FCFS baseline.

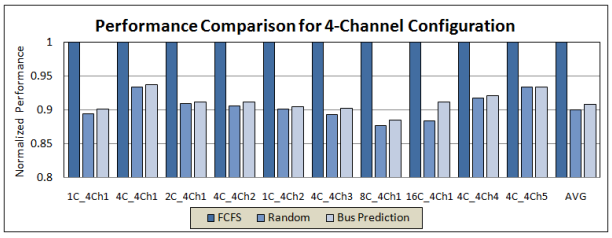


Fig. 2: Performance comparison of PRWL scheduling with FCFS in four channel memory configuration.

5.1 Performance

Figure 1 and Figure 2 summarize the performance results of simulation runs for the baseline FCFS scheduler and our two scheduler variations on the one- and four-channel memory configurations, respectively. In the one-channel configuration, Random consistently outperforms Bus Prediction except for some variations introduced by the randomness. Random shows 6.6% improvement on average over FCFS and 9.1% in the best case. This is due to Random's higher overall write leakage rate compared to Bus Prediction, which more strictly limits the PRE's issued for writes, and timely precharge is generally beneficial to system performance.

In the four-channel configuration, Random outperforms Bus Prediction with much less variation compared to the one-channel configuration. The average performance improves by 10.0% and 9.2% for Random and Bus Prediction, respectively. The best-case performance improves by 12.3% and 11.4% for our two scheduling policies. Because cache-line-interleaving address mapping is applied to the four-channel configuration, program row locality is lower in terms of DRAM device addresses. Random has a higher write leak rate, so more PRE's are issued, which favors the cache-

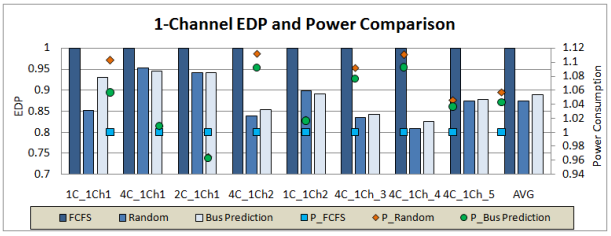


Fig. 4: EDP and power comparison for one-channel configuration.

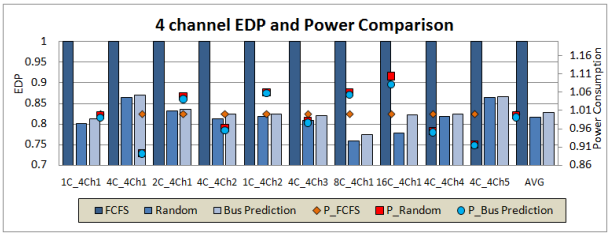


Fig. 5: EDP and power comparison for four-channel configuration.

line-interleaving address mapping. Therefore, the performance improvement of Random in the four-channel configuration is much higher than that in the one-channel configuration, and Random continues to perform better than Bus Prediction. Across all experiments, the overall performance improvement is 9.26% for Random and 7.78% for Bus Prediction.

5.2 Fairness Evaluation

Figure 3 presents the maximum slowdown and PFP for our PRWL scheduling algorithms compared to FCFS. All results are normalized to the FCFS baseline. As shown in Figure 3a, the maximum slowdown seen with PRWL is better than FCFS, which means that our PRWL scheduling algorithms improve the scheduling of all threads, not just certain ones. Figure 3b illustrates the overall PFP on all workloads except for the one-core configurations. On average, PRWL outperforms FCFS by 16.2% and 14.9% for Random and Bus Prediction, respectively. This rises to 24.3% and 22.8% in the eight-core configuration. PFP improvements arise mainly from improved performance since our fundamental scheduling principle is still first-come, first-served in PRWL.

5.3 Energy Consumption

Figure 4 and Figure 5 present the power consumption and EDP comparison between PRWL and FCFS for one-channel and four-channel workloads. The dots in the figures represent power consumption for each algorithm and the bars show EDP reduction. As stated earlier, our PRWL algorithms improve system performance over FCFS for all workloads. Power consumption is slightly increased but not consistently. Our proposed PRWL algorithms reduce average EDP by 12.5% and 11.4% in the one-channel configurations for Random and Bus Prediction, respectively, and by 18.4% and 17.3% in the four-channel configuration. The best case savings is 24.1% for Random on the eight-core, four-channel workload. These improvements stems from a significant increase in the row buffer hit rate for writes. For example, the write hit rate is increased from -46.8% in FCFS to -2.6% in Random for the *4C_4Ch2* workload. Other workloads show similar improvement. This saves power while at the same time increasing system performance.

6 Conclusion and Future Work

Memory access latency is a major bottleneck in system performance today. We propose optimizations to FCFS which interleave read and write requests and carefully select which read and write requests to issue in the other mode. This reduces the frequency that drain-write mode is entered and also partially increases read-write parallelism, which improves overall system performance, EDP, and PFP for all the MSC simulation configurations. Our interleaving algorithm introduces almost no overhead to the memory controller, which makes it practical for real world implementation.

Our current PRWL algorithm is designed for inter-thread scheduling. However, we expect that PRWL would improve intra-thread scheduling as well. We have done preliminary investigations of phase-prediction optimizations and will continue to pursue this technique for future work.

References

[1] Design guide for two ddr3-1066 udimm systems. Technical report, Micron Technology Inc., 2009.

[2] 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.

[3] J. Huh, D. Burger, and S. W. Keckler. Exploring the design space of future cmps. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, PACT '01, 2001.

[4] E. Ipek, O. Mutlu, J. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*, pages 39 –50, June 2008.

[5] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems - Cache, DRAM, Disk*. 2008.

[6] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of the International Symposium on Microarchitecture (MICRO)*, pages 65 –76, Dec. 2010.

[7] S. Rixner, W. Dally, U. Kapasi, P. Mattson, and J. Owens. Memory access scheduling. In *Proceedings of the 27th International Symposium on Computer Architecture (ISCA)*, pages 128 –138, June 2000.

[8] Z. Zhu and Z. Zhang. A performance comparison of dram memory system optimizations for smt processors. In *Proceedings of the International Symposium on High-Performance Computer Architecture (HPCA)*, pages 213 – 224, Feb. 2005.