Priority Based Fair Scheduling: A Memory Scheduler Design for Chip-Multiprocessor Systems

Chongmin Li, Dongsheng Wang, Haixia Wang, Yibo Xue

Tsinghua National Laboratory for Information Science and Technology, Beijing 100084, China
{liismn, wds, hx-wang, yiboxue}@tsinghua.edu.cn

***Abstract*— Memory is commonly a shared resource for a modern chip-multiprocessor system. Concurrently running threads have different memory access behaviors and compete for memory resources. A memory scheduling algorithms should be designed to arbitrate memory requests from different threads, provide high system throughput as well as fairness.**

This work proposes a memory scheduling algorithm, Priority-Based Fair Scheduling (PBFS), which classifies threads memory access behavior by dynamically updated priorities. Latency-sensitive threads have top-priority to guarantee system throughput, and starvation of memory-sensitive threads can be avoided simultaneously. Simulation results show that compared with a FCFS scheduler, PBFS improves the system throughput and fairness metric by 7.4% and 7.7% respectively.

I. Introduction

The gap between the CPU speed and memory speed keeps growing as technique advancing. Modern high performance chip-multiprocessor systems can support tens or hundreds of threads running concurrently, the latency of off-chip memory accesses has long been a bottleneck of high performance memory subsystems. Shared memory is commonly adopted in modern chip-multiprocessors, a thread may contends with other threads when accessing memory. A straightforward solution is to increase the number of memory channels and/or memory banks in a channel. As DRAM has numerous strict timing constraints, it is critical for the memory controller to schedule concurrent memory requests to exploit memory level parallelism [1], [2], [3]. Earlier studies on memory controller primarily improve memory performance [1], [4], [5], [6]. As more threads are running concurrently in modern CMP systems, a memory scheduling algorithm should focus on memory contention as well as promote throughput.

According to threads' memory access behavior, the threads can be classified as latency-sensitive threads and bandwidth-sensitives threads [7]. System throughput benefits more from prioritizing latency-sensitive threads than memory-sensitive threads. Memory-sensitive threads are prone to starvation if less memory-sensitive threads are statically given a higher priority over them. A good memory scheduler algorithm should achieve both high throughput and good fairness. In this work, we propose a memory scheduling algorithm named *Priority Based Fair Scheduling* (PBFS) which can provide both fairness and high throughput. *PBFS* gives each thread a priority, which is updated when performing memory accesses or reaching a time threshold. On every memory cycle, the memory request which has the highest thread priority is selected to issue. *PBFS* achieves fairness by dynamically updating threads' priority, starvation of memory-sensitive threads can be avoided because the priority of a frequently issued thread decreases rapidly.

The paper makes the following contributions:

- We propose a priority based mechanism for each thread which helps the scheduler to make decision. Latency-sensitive threads and bandwidth-sensitive threads can be classified through different priorities.
- We illustrate that *PBFS* can guarantee system throughput as well as fairness. latency sensitive threads has top-priority which can be issued as soon as possible. Starvation of bandwidth-sensitive thread is avoid by dynamic priority updating.
- *PBFS* scheduler is easy to implement, and has low hardware overhead.
- Evaluation results under two memory configurations show that *PBFS* can improve system throughput, fairness metrics as well as energy-delay-product (EDP).

The rest of this paper is organized as follows. Section II gives a brief introduction of memory systems. Section III presents our *PBFS* design in detail. Section IV and V presents the experimental methodology and evaluation results. Section VI discusses the related work and Section VII summarizes the paper.

II. Background

A. Basic DRAM Architecture

Most of modern DRAM systems make use of dual in-line memory modules (DIMM). A basic DRAM consists of one or more DRAM channels, each channel has one or more memory modules. A modern DDR3 channel typically can support 1-2 DIMMs; each DIMM is typically consists of 1-4 ranks; each rank can be partitioned into multiple (4-16) banks. All banks in an active rank must sequentially share the data and command wires of the memory channel while different bank can process different memory requests in parallel. Figure 1 gives a basic DRAM system structure, with two DRAM channels. Each channel has one double ranked DIMM, with eight internal banks per rank.

B. DRAM Commands

The memory commands can be partitioned into two groups, commands that advance the execution of a pending memory request (read or write), or commands that manage general

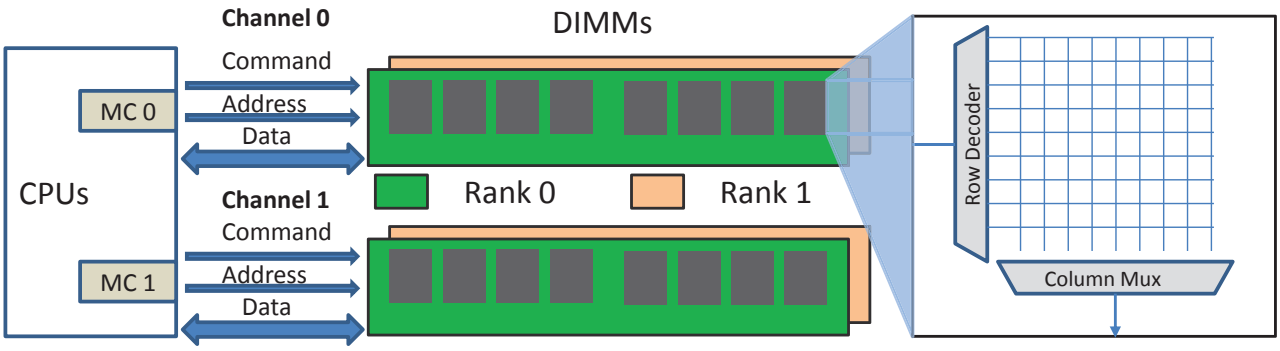


Fig. 1: Basic DRAM structure, with two independent channels , one double ranked DIMM per channel, and eight internal banks per rank.

DRAM state. There are four commands which advance the execution of a memory request:

- **PRE**: Precharge the bitline of a bank so a new row can be read out.
- **ACT**: Bring the contents of a bank's DRAM row into the bank's row buffer.
- **CLO-RD**: Bring a cache line from the row buffer to the processor.
- **CLO-WR**: Write a cache line from the processor to the row buffer.

DRAM state management commands include five memory commands, as follows:

- **PWR-DN-FAST**: Puts a rank into the low-power-mode with quick exit times.
- **PWR-DN-SLOW**: Puts a rank into the precharge-power-down (slow) mode with longer time to transition into the activate state.
- **PWR-UP**: Brings a rank out of low-power mode.
- **Refresh**: Forces a refresh to multiple rows in all banks in a rank.
- **PRE-ALL**: Forces a precharge to all banks in a rank.

When the memory system is not busy, PWD-DN-FAST and PWR-DN-SLOW commands can put memory ranks into low-power-mode to save power. PWR-UP command is needed to bring a rank out of low-power-mode. As DRAM is non-persistent, periodically refresh is needed to maintain the data integrity (Refresh). To design a memory scheduler, various timing constraints must be met. Details about these constrains can be found in [8].

C. Address Mapping

The address mapping policy determines the extent of parallelism that can be leveraged within the memory system. In this paper, a cache line is placed entirely in one bank, and two different processor-memory configuration are studied. The first configuration places consecutive cache lines in the same row and tries to maximize row buffer hits. The second configuration tries to maximize memory access parallelism by scattering consecutive blocks across channels, ranks, and banks. More system details can be found in Section IV.

There is a row buffer in each bank to store the last row accessed within the bank. When there is a read or write command, one row must be copied into the bank's row buffer (ACT), then complete the read or write operation. If the following requests to the same bank can be serviced by data already in the open row buffer, the request consumes less time and energy. For a following request to a different row, current contents in row buffer must be written back to the DRAM arrays (PRE). If a following memory command hits the row buffer, it take less latency and energy to complete the operation.

D. Typical Schedulers

Here we introduce two typical schedulers which results will be evaluated together with our design.

FCFS: FCFS assumes that the read queue is ordered by request arrival time, the scheduler simply scans the read queue sequentially until it finds an instruction that can issue in the current cycle. A separate write queue is maintained, when the write queue size exceeds a high water mark, writes are drained similarly until a low water mark is reached. The scheduler switches back to handling reads at that time. Write queue will be drained if there is no pending reads.

Close: This policy is an approximation of a true close-page policy. In every idle cycle, the scheduler issues precharge operations to banks that last serviced a column read/write. Unlike a true close-page policy, the precharge is not issued immediately after the column read/write and we dont look for potential row buffer hits before closing the row.

III. Memory Scheduler Design

A. Ranking Requests with Threads' Priorities

Kim et al. [7] classified threads into latency-sensitive cluster and bandwidth-sensitive cluster. Latency-sensitive threads are computationally intensive which get low number of memory requests. The performance of latency-sensitive threads is sensitive to memory request latency. Bandwidth-sensitive threads

have more memory requests and spend a large portion of their time for waiting the memory responses.

In *Priority-Based Fair Scheduler* (PBFS), a thread's priority is an integer ranges from -1 to a positive number (n). The priority of each thread can be divided into four categories.

- Top-priority (n): Read requests from threads with top-priority need to be served as soon as possible.
- Bottom-priority (0): Threads have bottom-priority are memory-sensitive and a number of requests have been served recently.
- Medium-priority (1 to n-1): Threads have medium-priority may be either latency-sensitive or memory-sensitive. One or several read requests from the corresponding tread have been served recently.
- Idle-priority (-1): When a thread doesn't issue memory request during last time interval, it's priority is set to idle-priority. The thread may either be busy in computing or be idle.

On every memory cycle, the scheduler traverses the read request queue to find if there is a request with top priority can be served. If there is no request with top-priority or the request has top-priority can not be issued immediately, the scheduler chooses an issuable request from the read queue in order. The thread priorities are updated either on issuing a request or on arriving a periodically time threshold. Detailed updating mechanism is given in Section III-B.

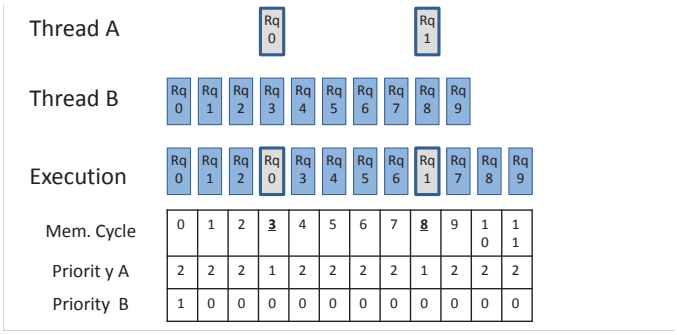
In *PBFS*, a thread which has top-priority is treated as latency-sensitive thread. Memory scheduler tries to serve it's requests as soon as possible. Bottom-priority means a threads maybe memory sensitive, it's requests are not served until all requests with top-priority are issued (or can't issue in current cycle). A medium-priority thread can either be upgraded to a top-priority thread or be downgraded to a bottom priority thread, which is depend on whether its read request is served by the scheduler or not. An idle-priority thread is either a latency sensitive thread which doesn't have memory request for some time or an inactive thread.

B. Priority Updating Rules

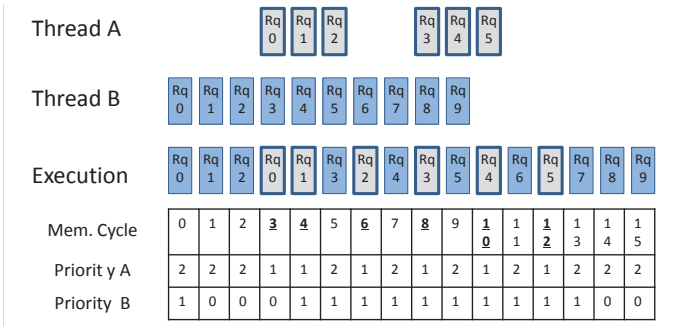
The threads' priorities updating rules in *PBFS* are as follows:

- If there is a read request with Idle priority, which means the first read request from corresponding thread after a time interval, it's priority is change to top priority directly.
- When there is no request in read queue has the top priority, the priorities of all threads with medium or bottom priority are increased by one, so at least one core's priority reaches top priority.
- When a read request is issued, the corresponding thread priority minus by one.
- When there is no request from a thread in the last time interval, the thread priority is set to idle-priority.

When idle threads are identified, the value of top-priority is adjusted in accordance to fit the number of active threads. At the same time, the value of top-priority is also adjusted



(a) Case A: Thread A has two memory request sequence, each sequence contains one memory request.



(b) Case B: Thread A has two memory request sequence, each sequence contains three memory requests.

Fig. 2: Examples of fairness, the next memory cycle of underlined cycle increase both threads' priorities first.

by *PBFS* according to the threads' memory behavior. When different threads have unbalance memory requests, top-priority is increased to give more chance to the latency-sensitive threads and vice versa.

C. Fairness Concern

In *PBFS*, a memory requests from a thread which is latency-sensitive are prioritized over a request from bandwidth-sensitive core. Previous studies show that prioritizing latency sensitive threads which access memory infrequently can increase the overall system throughput [9], [7]. Assuming an example system with one latency-sensitive/less bandwidth-sensitive thread (A) and a bandwidth-sensitive thread (B), both threads' priorities are set to top-priority (2), as shown in Figure 2. In Case A, thread A is a latency-sensitive thread. When thread A issued a request and lose its top-priority, according to the priority updating rules, its priority will soon

	1channel.cfg	**4channel.cfg**
Priority monitor	4	4
Request counter	4	4
Open row monitor	2*8	2*8
Priority upper/lower bound	2	2
Top priority	1	1
Total	27 (108Bytes)	27 (108Bytes)

TABLE I: Per channel storage overhead of PBFS

TABLE II: Memory system configurations

Parameter	1channel.cfg	4channel.cfg
Processor clock speed	3.2GHz	3.2GHz
Processor ROB size	128	160
Processor retire width	2	4
Processor fetch width	4	4
Processor pipeline depth	10	10
Memory bus speed	800 MHz (plus DDR)	800 MHz (plus DDR)
DDR3 Memory channels	1	4
Ranks per channel	2	2
Banks per Rank	8	8
Rows per bank	32768 × NUMCORES	32768 × NUMCORES
Columns (cache lines) per row	128	128
Cache line size	64 Bytes	64 Bytes
Address bits	32 + log(NUMCORES)	34+log(NUMCORES)
Write queue capacity	64	96
Address mapping	row:rank:bank:chnl:col:blkoff	row:col:rank:bank:chnl:blkoff
Write queue bypass latency	10 cpu cycles	10 cpu cycles

be upgraded to top-priority because of following requests from thread B. *PBFS* avoids starvation of bandwidth-sensitive threads by limiting the number of requests belong to latency-sensitive thread. As show in Figure 2b, thread A is marked as latency-sensitive thread at the beginning. When thread A consecutively issue two request, the priority of thread B is the same as thread A. A request from either thread A or thread B can be issued at the next memory cycle. If the next request is still from thread A, thread B will get top-priority, otherwise a request from thread B will be issued (the case of this example). Anyway, thread B avoids starvation.

As mentioned in Section II, if a memory request hits the row buffer in a bank, both time and power are saved. However, if a row buffer is unlikely to yield future hit, closing the row and precharging the bitlines is beneficial as the bank can quickly serve a new memory request. In this work, we close an open row when there is no request needs to issue.

D. Implementation and Hardware Overhead

PBFS needs hardware support to: 1) record the priority of each thread, 2) monitor the threads' behavior (read counts within a time interval) and 3) maintain the flags that whether a row buffer can close or not. Table I gives the major storage overhead for the two main memory configurations with 4 threads in our evaluation. If we use 4-bytes for each counter, the total storage overhead of each channel is 108-bytes, which is relatively small. *PBFS* needs additional logic to rank and update threads' priority and other monitors. More counters are needed when the number of threads increases, but the storage overhead is still small compared with the memory capacity.

IV. Experimental Setup

We use the Usimm [8] simulation infrastructure to build our memory system scheduler. Table II gives the system configurations used in our evaluation.

The traces used in our simulation are gathered from 10 different benchmarks listed in table III through simics [10]. The 10 traces are used to form 10 different workloads. All 10 workloads are run with 4channel.cfg, and the first 8 workloads are also run under 1channel.cfg. The memory workloads and corresponding traces used are given in table IV.

TABLE IV: Workload description

name	description
mix-1	comm2
mix-2	comm1 comm1
mix-3	comm1 comm1 comm2 comm2
mix-4	MT0-canneal MT1-canneal MT2-canneal MT3-canneal
mix-5	fluid swapt comm2 comm2
mix-6	face face ferret ferret
mix-7	black black freq freq
mix-8	stream stream stream stream
mix-9	fluid fluid swapt swapt comm2 comm2 ferret ferret
mix-10	fluid fluid swapt swapt comm2 comm2 ferret ferret black black freq freq comm1 comm1 stream stream

V. Evaluation

We compare *PBFS* with two typical implementations, *FCFS* and *Close*. Three metrics are taken in the evaluation: the sum of threads execution time, the threads' max slowdown and the energy-delay-product (EDP). All results all given in Table V.

A. Execution Time

The proposed *PBFS* scheduler gets 7.4% and 3.3% execution time reduction over the baseline *FCFS* design and *Close*

TABLE III: Benchmark trace description

Trace	Description
black	A single-thread run from PARSEC's blackholes
face	A single-thread run from PARSEC's facesim
ferret	A single-thread run from PARSEC's ferret
fluid	A single-thread run from PARSEC's fluidanimate
freq	A single-thread run from PARSEC's freqmine
stream	A single-thread run from PARSEC's streamcluster
swapt	A single-thread run from PARSEC's swaption
comm1	A trace from a server-class transaction-processing workload
comm2	A trace from a server-class transaction-processing workload
MT*-canneal	A for-thread run from PARSEC's cannel, organized in for files, MT0-cannel to MT3-cannel

TABLE V: Comparison of key metrics on baseline and proposed schedulers. c1 and c2 represent commercial transaction-processing workloads, MT-canneal is a 4-threaded version of canneal, and the rest are single-threaded PARSEC programs. "Close" represents an opportunistic close-page policy that precharges inactive banks during idle cycles.

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	397	NA	NA	NA	4.23	3.98	3.94
MT-canneal	4 chan	179	167	164	NA	NA	NA	1.78	1.56	1.50
bl-bl-fr-fr	1 chan	149	147	142	1.20	1.18	1.14	0.50	0.48	0.46
bl-bl-fr-fr	4 chan	80	76	76	1.11	1.05	1.05	0.36	0.32	0.32
c1-c1	1 chan	83	83	82	1.12	1.11	1.10	0.41	0.40	0.40
c1-c1	4 chan	51	46	47	1.05	0.95	0.97	0.44	0.36	0.38
c1-c1-c2-c2	1 chan	242	236	227	1.48	1.46	1.43	1.52	1.44	1.34
c1-c1-c2-c2	4 chan	127	118	117	1.18	1.10	1.10	1.00	0.85	0.85
c2	1 chan	44	43	43	NA	NA	NA	0.38	0.37	0.36
c2	4 chan	30	27	28	NA	NA	NA	0.50	0.39	0.42
fa-fa-fe-fe	1 chan	228	224	211	1.52	1.48	1.39	1.19	1.14	1.02
fa-fa-fe-fe	4 chan	106	99	97	1.22	1.15	1.11	0.64	0.56	0.54
fl-fl-sw-sw-c2-c2-fe-fe	4 chan	295	279	264	1.40	1.31	1.23	2.14	1.88	1.70
fl-fl-sw-sw-c2-c2-fe-fe--bl-bl-fr-fr-c1-c1-st-st	4 chan	651	620	586	1.90	1.80	1.69	5.31	4.76	4.34
fl-sw-c2-c2	1 chan	249	244	229	1.48	1.43	1.33	1.52	1.44	1.28
fl-sw-c2-c2	4 chan	130	121	121	1.13	1.06	1.06	0.99	0.83	0.85
st-st-st-st	1 chan	162	159	154	1.28	1.25	1.20	0.58	0.56	0.53
st-st-st-st	4 chan	86	81	81	1.14	1.08	1.08	0.39	0.35	0.35
	1 chan	1577	1538	1483	1.35 PFP: 1501	1.32 PFP: 1438	1.26 PFP:1320	10.34	9.82	9.33
	4 cha:n	1735	1634	1582	1.27 PFP: 1936	1.19 PFP: 1711	1.16 PFP: 1615	13.53	11.87	11.25
Overall		3312	3173	3065	1.30 PFP: 3438	1.24 PFP: 3149	1.21 PFP: 2934	23.88	21.70	20.58

design respectively. The top two reductions are 10.5% from Mix-9 (8 threads) and 9.8% from Mix-10 (16 threads), both running at 4-channels configuration. For the execution time of other workloads on 4-channels, *PBFS* gets results close to *Close* design. Both *PBFS* and *Close* outperform the *FCFS* design on all workloads in both configurations.

B. Fairness metric

In this evaluation, the fairness metric is defined as the maximum slowdown for any thread in the workload, relative to a single-program execution of that thread with an *FCFS* scheduler (a high number is bad). The average max slowdown are 1.30, 1.24 and 1.21 respectively for *FCFS*, *Close* and *PBFS* designs. Workloads on 1-channel configuration has higher max slowdown than 4-channel configuration as there are more contentions in 1-channel configuration. The maximum slowdown for 1-channel configuration are 1.52, 1.48 and 1.39 in Mix-6. Mix-10 has maximum threads concurrently running, thus it gets max slowdown in all three designs.

C. Energy-delay-product

Compared with *FCFS*, our proposed *PBFS* scheduler reduces EDP by 13.8%, while *Close* gets 9.1% EDP reduction against *FCFS*. The maximum reduction of *PBFS* is 20.5% which comes from Mix-9, and the next is 18.3% from Mix-10. Workloads under 1-channel configuration has less EDP reduc-

tion than workloads under 4channel.cfg where the maximum EDP reduction is 15.8% on Mix-5.

VI. Related Work

We discuss some of the most relevant prior work in the following.

Memory scheduler designs that do not distinguish requests from different threads [11], [12]. These works were mainly tested for single threaded, vector or streaming architectures, and aimed to maximize DRAM throughput. For multi-threaded context, thread-unaware scheduling policies were not as efficient as before and prone to starvation [13], [7].

Thread aware memory schedulers were designed in recent studies which improved fairness as well as QoS. Fair queueing memory schedulers[14], [13] used variants of the fair queueing algorithm to construct a memory scheduler which provides QoS to each thread. Stall-time fair memory scheduler (STFM) [5] estimates the slowdown of each thread when running alone and prioritizes the thread which has most slowdown. The main propose of these works are to maximize fairness.

Parallelism-aware batch scheduling (PAR-BS)[6] groups memory requests into batches and older batches gets higher priority over younger batches. PAR-BS some times implicitly leads to unfairness when memory-intensive threads insert too many requests into a bach.

ATLAS [9] aims to get maximum system throughput. threads attained least memory service get the highest priority. Thread Cluster Memory Scheduling (TCM) [7] divides threads into two separate clusters, latency-sensitive and bandwidth sensitive cluster, and employs different memory scheduling policies in each cluster and periodically shuffles the priority ordering among the threads in the bandwidth sensitive cluster.

Ipek et al. [15] propose self-optimizing memory controller based on reinforcement learning (RL) to pick the actions that maximize the desired long-term objective function. MORSE [16] improves Ipek et al.'s work, the new designed scheduler has the capability of targeting arbitrary figures of merit.

VII. Conclusions

We have presented *Priority-Based Fair scheduler* (PBFS), a memory controller scheduling technique that elegantly leverage memory throughput and fairness through a concise priority-based scheduler design. Through dynamically priority updating mechanism, *PBFS* prioritizes latency-sensitive threads over bandwidth-sensitive threads. Starvation of memory-sensitive threads is avoided as there is no latency-sensitive thread can consecutively issue a number of memory request. The implementation of *PBFS* is easy and the hardware overhead is small.

Our simulation results on both 1 channel and 4 channels memory configurations show that *PBFS* can achieve 7.4% execution time reduction over the baseline *FCFS* design. *PBFS* also provides around 7.7% promotion on fairness metric. And the EDP metric is reduced by 13.8%. Overall, we believe this research is a viable approach to scale future memory system.

Acknowledgment

The authors acknowledge the support of the Nature Science Foundation of China under Grant No. 60833004, 60970002, and the National 863 High-Tech Programs of China(No.2012AA010905, 2012AA012609).

References

[1] V. Cuppu and B. Jacob, "Concurrency, latency, or system overhead: which has the largest impact on uniprocessor dram-system performance?" in *Proceedings of the 28th annual international symposium on Computer architecture*, ser. ISCA '01, 2001, pp. 62–71.

[2] V. Cuppu, B. Jacob, B. Davis, and T. Mudge, "A performance comparison of contemporary dram architectures," in *Proceedings of the 26th annual international symposium on Computer architecture*, ser. ISCA '99, 1999, pp. 222–233.

[3] Z. Zhu and Z. Zhang, "A performance comparison of dram memory system optimizations for smt processors," in *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, feb. 2005, pp. 213 – 224.

[4] I. Hur and C. Lin, "Adaptive history-based memory schedulers," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37, 2004.

[5] O. Mutlu and T. Moscibroda, "Stall-time fair memory access scheduling for chip multiprocessors," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 40, 2007, pp. 146–160.

[6] M. Onur and M. Thomas, "Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared dram systems," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08, 2008, pp. 63–74.

[7] 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*, ser. MICRO '43, 2010, pp. 65–76.

[8] N. Chatterjee, R. Balasubramonian, M. Shevgoor, S. H. Pugsley, A. N. Udipi, A. Shafiee, K. Sudan, M. Awasthi, and Z. Chishti, "Usimm: the utah simulated memory module." [Online]. Available: http://www.cs.utah.edu/ rajeev/pubs/usimm.pdf

[9] Y. Kim, D. Han, O. Mutlu, and M. Harchol-Balter, "Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers," in *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*, jan. 2010, pp. 1 –12.

[10] *Wind River Simics Full System Simulator. http://www.windriver.com /products/simics/.*

[11] 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*, ser. ISCA '00, 2000, pp. 128–138.

[12] J. Shao and B. Davis, "A burst scheduling access reordering mechanism," in *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on*, feb. 2007, pp. 285 –294.

[13] K. Nesbit, N. Aggarwal, J. Laudon, and J. Smith, "Fair queuing memory systems," in *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, dec. 2006, pp. 208 –222.

[14] N. Rafique, W.-T. Lim, and M. Thottethodi, "Effective management of dram bandwidth in multicore processors," in *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, sept. 2007, pp. 245 –258.

[15] E. Ipek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ser. ISCA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 39–50. [Online]. Available: http://dx.doi.org/10.1109/ISCA.2008.21

[16] J. Mukundan and J. Martinez, "Morse: Multi-objective reconfigurable self-optimizing memory scheduler," in *High Performance Computer Architecture (HPCA), 2012 IEEE 18th International Symposium on*, feb. 2012, pp. 1 –12.