Request Density Aware Fair Memory Scheduling

Takakazu Ikeda
Tokyo Institute of Technology
ikeda@arch.cs.titech.ac.jp

Shinya Takamaeda-Yamazaki
Tokyo Institute of Technology
JSPS Research Fellow
takamaeda@arch.cs.titech.ac.jp

Naoki Fujieda
Tokyo Institute of Technology
fujieda@arch.cs.titech.ac.jp

Shimpei Sato
Tokyo Institute of Technology
satos@arch.cs.titech.ac.jp

Kenji Kise
Tokyo Institute of Technology
kise@cs.titech.ac.jp

ABSTRACT

With the rapid advances in semiconductor process technology and microarchitecture, the speed gap between the clock cycle time of processor cores and that of memory systems has increased significantly. To solve this problem, memory system should be efficiently managed. In general, since a thread with fewer requests has a large capability to improve the total execution performance by being prioritized over the other threads with many requests, these lightweight threads should be more prioritized over the heavy threads. This paper describes our high-performance and fair memory scheduler employing 3 distinct prioritizing algorithms: (1) Thread Clustering, (2) Read Density Survey, and (3) PC-based Gain Estimation. We combined distinct prioritizing schemes to dynamically deal the multi-grain changes of program behaviors. The evaluation result shows that our memory scheduler improves the performance by 10.9% in average compared with First-Come First-Served (FCFS) memory scheduling.

1. INTRODUCTION

The performance of processor cores is rapidly improving with the technology scaling. However, long latency and inefficient bandwidth of memory system are critical bottlenecks in the performance[7][8]. Memory scheduling on DRAM is one of remarkable issues to affect the performance on modern processors. In modern chip-multiprocessors (CMP), not only the system throughput of memory system, but also fairness among programs sharing the resources should be seriously considered. To further improve the overall throughput and fairness in memory systems, we need a clever scheduler exploiting differences in application behaviors.

State-of-art memory scheduling algorithms improve the memory performance by increasing the bandwidth utilization and decreasing the latency. Conflict on row buffer seriously increases memory access latency[10][1]. Close-Page Policy[9] is an effective policy to reduce the latency on row buffer conflicts by aggressively precharging. Importance of each memory request is different depending the application characteristics. Criticality-aware reordering of requests is good to improve the total throughput. ATLAS[5] is a throughput-oriented scheduler that periodically reorders the priorities of each thread serving memory requests. Every thread has a priority based on the served amount of requests. Memory controller prioritizes a thread with the fewest service amounts over the others for each period. Thread Cluster Memory Scheduling (TCM)[6] is thread-behavior-aware scheduler to improve both the system throughput and the fairness. In TCM, threads are clustered into 2 groups: light thread cluster and heavy thread cluster. In order to improve the total system throughput, higher priorities are assigned to the light threads over the heavy threads. Priorities of heavy threads are shuffled for every interval period to improve manage the fairness among whole the threads.

This paper describes our high-performance and fair memory scheduling scheme employing 3 distinct algorithms: coarse-grain thread prioritization, fine-grain thread prioritization, and instruction-level prioritization. Combining multiple algorithms improves the memory performance and the fairness among multiple applications by dynamically detecting multi-grain behavior changes in programs.

For higher bandwidth utilization, we optimized the drain mode of write requests. In the default scheduler, write requests are issued in coarse-grain. A write request will be issued when the read queue on the channel is empty or the write queues is filled with much requests. In order to leverage the bank-level parallelism, we adopted fine-grain write-drain mode in our scheduler. A write request in the write queue is processed if the destination bank of the request is idle. However, a read request should be prioritized over a write request to progress the application execution[3]. Too aggressive processing of writes may occur row conflicts with the other read requests and will degrade the total performance. To avoid these read-write conflicts, the scheduler waits for a phase with no read requests to the write destination bank. To issue a write request destined to each bank,

the scheduler waits a certain period after the write request is ready to be issued and all the read requests to the write destination bank were issued.

This fine-grain write draining is very effective to improve the total performance in bandwidth sensitive situations. In addition to these optimizations for bandwidth, row buffer management is very important to improve performance[4]. Activate commands are often issued to prepare the data on row buffer to increase the row buffer hit rate, and precharge commands are aggressively issued to reduce row buffer conflicts.

2. IMPLEMENTATION

2.1 Thread Priority

Our scheduler is made up from 3 distinct algorithms to treat differences of thread behaviors and instruction behaviors: (1) Thread Cluster, (2) Read Density Survey and (3) PC-based Gain Estimation. A priority score to determine the each priority is assigned to each thread for every DRAM cycle by these schemes. A read request from a thread with higher priority score is processed earlier than a read request from a thread with lower priority. Assigning the different priorities to each thread according to its behavior type improves the chip-level fairness and the program performance.

Figure 1 shows the thread prioritizing mechanism of our scheduler. Our scheduler has 2 scheduling stages of the read requests. In beginning of scheduling, the scheduler selects a thread by PC-based gain estimation to pursue instruction-level differences of program behavior. If any gainful requests with large capability to progress the program are found, the scheduler prioritizes these requests than the others. If gainful requests were not found by PC-based gain estimation, the scheduler then selects a candidate request by thread-level prioritization. In this paper, to optimize the gainfulness of scheduler, we decided to take advantage of this metric in the top two threads. To pursue thread-level differences of program behavior, we use two schemes to determine the priorities. The total priority score to determine each thread priority is calculated as sum of two scores by these schemes. The priority score to pursue the coarse-grain the changes of thread behaviors are assigned by thread clustering. The priority score to pursue the fine-grain changes of thread behaviors are assigned by read density survey.

The following sections describe key schemes of our scheduler to assign the thread priorities.

2.1.1 Thread Clustering

Thread Cluster Memory (TCM) Scheduling is a high performance memory scheduler with good fairness by employing different scheduling policies for each thread behavior. We used this scheduling algorithm to improve the total performance of all threads with keeping the fairness. A key insight described in the paper of TCM is that prioritizing the latency-sensitive threads over the bandwidth sensitive threads achieves higher performance than prioritizing the bandwidth sensitive threads over the latency sensitive threads. Latency sensitive thread is defined as a thread causing infrequent cache-misses, and, on the other hand, bandwidth sensitive thread is defined as a thread causing frequent cache misses. In TCM, all the running threads are clustered into 2 groups, a latency sensitive cluster and a bandwidth sensitive cluster, based on the number of requests from each thread for every last quantum time (quantum is a predetermined certain time interval period. about 1M cycles in our scheduler). For higher throughput of the system, memory requests of latency sensitive threads are prioritized over the other requests of bandwidth sensitive threads. Each priority of bandwidth sensitive threads is shuffled for each short interval time (about 1K cycles in our scheduler).

We use a modified version of original TCM algorithm to optimize for this contest. The number of latency sensitive threads is dynamically determined in our scheduler. Latency sensitive threads are determined according to the amount of read requests in the last quantum and the amount of read requests from the beginning of the program from each thread.

2.1.2 Request Density Survey

TCM cannot follow small changes of a program 's phase because it determines the thread priorities by using the sum of requests in the past interval period. We employ a fine-grain prioritization mechanism to follow small changes of the thread behavior. In order to determine the priority score of each thread, the memory controller counts the number of read requests in each read queue for every DRAM cycle. Lower priority score is assigned to a thread containing many read requests, and higher priority score is assigned to a thread containing fewer requests.

2.1.3 PC-based Gain Estimation

Each cache-miss instruction has different capability to progress the program. A thread will significantly progress, when some particular cache-miss instructions are processed in not only latency sensitive situations but also bandwidth situations. We referred to these particular cache-miss instructions with large capability to progress a program as "Gainful Instructions". To improve the total throughput, gainful instructions should be prioritized over the other cache-miss instructions.

In order to deal with these instruction-level differences, we propose Program-Counter-based (PC-based) gain estimation mechanism that uses the cache-miss history to estimate the gain of each cache-miss instruction when it is processed. Figure 2 shows the architecture of Gain History Table (GHT) to track the gains of past cache-miss load instructions. GHT is a full-way set-associative CAM structure of valid bit, a program counter tag and a saturated reference counter for each entry. A GHT is prepared for each thread (or core). When a new cache-miss of load occurs, the memory controller refers the GHT by using the PC of the instruction in order to estimate the gain value of a cache-miss instruction. If there is an entry of it, GHT returns the value of its reference counter. The memory controller uses this value to calculate the priority score at instruction-level.

Figure 3 shows how to update the GHT. The last cache-miss instruction of load is recorded in the Last Read Request (LRR) table that consists of PC of last cache-miss load instruction and its gain count. Gain count in LRR is incremented for every DRAM cycle if no following cache-miss load

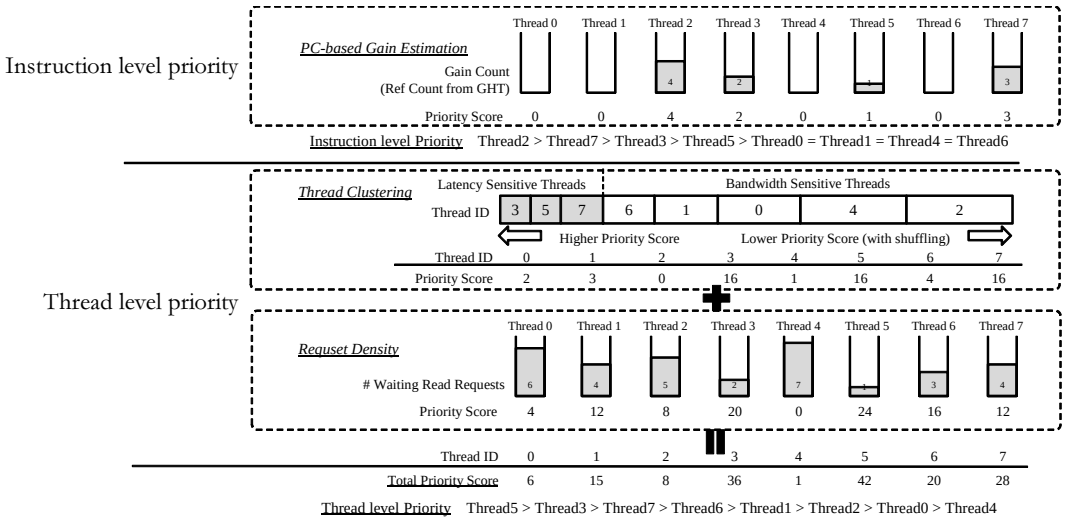

Figure 1: Priority score calculation in our scheduler. 3 distinct algorithms are used to determine the priority. At the beginning of scheduling, PC- based gain estimation is used to select a prioritized thread. Then, the thread with higher priority score is the more prioritized than the lower ones. Priority score is calculated by thread clustering and read density survey.

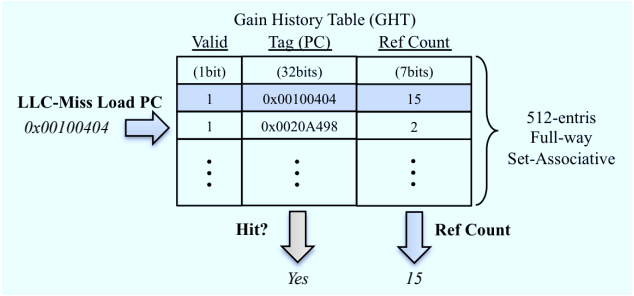

Figure 2: GHT: Gain History Table

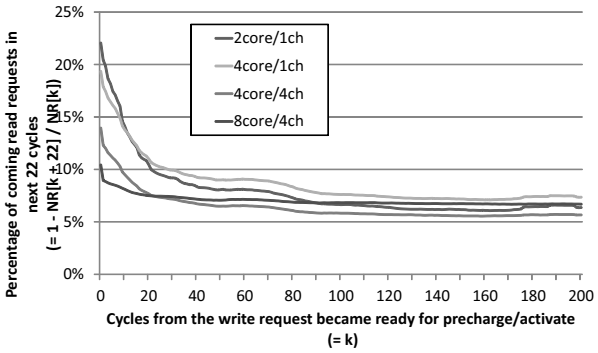

Figure 4: Wait Cycles for Write

instruction comes. When a following cache-miss load occurs, instruction in LRR is candidate to commit to the GHT. If the gain count in LRR is greater than the threshold value (TH_GAIN in the figure), the instruction is committed. If there is not a corresponding entry for the instruction in the GHT, a new entry is allocated. If there is a corresponding entry, the value of reference counter is incremented. Otherwise, if the gain count in LRR is less than the threshold value and there is a corresponding entry in the GHT, the value of reference counter is decremented[1].

2.2 Write Optimization

When the write queue is not so crowded, the default close policy tries to issue write requests only if the read queue is empty, otherwise it deals with read requests first. However, this policy often bumps into apparently inefficient cases that the controller keeps waiting for issuing read requests while issuable write requests exist in unrelated banks.

So we first divide the condition for starting to write. This means that our scheduler tries to drain write requests in banks where no read requests exist, though the read queue is not empty. It enables to increase chances to drain write requests and improves the performance especially with bandwidth-sensitive workloads.

On the other hand, aggressive write drain causes another problem particularly with latency-sensitive workloads. When a read request arrives right after issuing a precharge or activate command to the same bank for writing, the request will be delayed. And to make matters worse, if the interruption occurs before issuing the actual write command, the request will be cancelled and cannot be removed from the write queue. Therefore a single write request can be in-

[1]In some cases, not decrementing the value of reference count achieves higher performance than the decrementing case. In our scheduler, the decrement value is variable, 0 or larger.

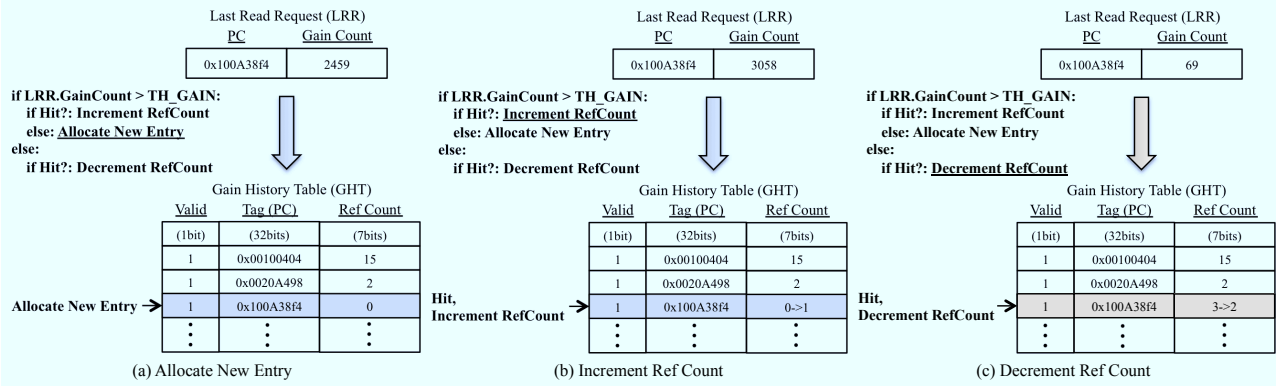

Figure 3: Update GHT

terrupted multiple times, and it causes a terrible effect on latencies.

To avoid the interruptions, it is desirable for the scheduler to drain write requests when a read request to the same bank is less likely to come. Our idea for achieving this is to wait some cycles to begin to issue a precharge or active command for writing.

We made an experiment to see if our idea is useful. We successively record the cycles between the time the first precharge or activate command for writing becomes ready after the completion of read requests in a bank and the time the next read request to the same bank arrives at the read queue. Then we define and enumerate NR[i] as the number of times the next read request does not come until i or more cycles. We also define $R_{k,l}$ as the proportion of arrival of the request in the next l cycles when we wait k cycles for issuance of a precharge or activate for writing. Using the array NR, We can calculate $R_{k,l}$ according to the following equation: $R_{k,l}$ = 1 - NR[k + l] / NR[k] Since it takes 22 (tRP + tRCD) cycles for a DRAM to complete a precharge and activation, we can estimate the probability of interruption at $R_{k,22}$. Thus, we enumerate NR with various settings and inspect relationships between k and $R_{k,22}$.

Figure 4 shows relationships with four settings: 2 threads / 1 channel, 4 threads / 1 channel, 4 threads / 4 channels, and 8 threads / 4 channels. The x-axis stands for k, the number of waiting cycles, and the y-axis stands for $R_{k,22}$ or the estimated probability of interruption. Workloads are several mixes of body, black, fluid, and canneal. The left edge of the graphs show that 10%-22% of the write requests are likely to be interrupted by a new read request if we issue precharge or activate commands for writing right after they are ready.

However, if we wait for only 40 cycles to issue them, the probability of interruption decreases sharply. Also, it continues to decrease almost monotonically when the number of waiting cycles increases up to about 100 cycles. So it is possible to avoid the interruption by waiting some cycles for issuance of precharge or activate commands for writing.

2.3 Row Buffer Management

Efficient row buffer management reduces the memory access latency. Increasing row buffer hit rate and decreasing row buffer conflicts achieves smaller memory access latency. A general memory access needs a row activate command and a column read/write command. If the current row address in a bank is different from the row address of following request, a precharge command is issued before the following row is activated. The overhead of a precharge takes long period, and it suffers the performance.

Close-Page Policy is an efficient policy to reduce conflicts on row buffers by aggressive precharges. When a column read/write request is issued, the corresponding row buffer is precharged to avoid row buffer conflicts. Using aggressive activate command is another way to reduce memory access latency. If the row address of a new request is the same as the current opened row address already, the memory access spends just short time for column read/write. Decreasing of latency activating a row buffer improves the memory performance. In order to improve the row buffer hit rate for the higher performance, in our scheduler, if any issuable read requests do not exist in the read queue of each channel, the memory controller activates corresponding row of the next candidate read request from the most prioritized thread.

3. EVALUATION

Table 1 shows the evaluation result of our scheduler in performance, PFP and EDP.

3.1 Performance

We evaluated the performance of our scheduler by using usimm version 1.3 of DRAM timing model simulator, in 2 configurations of 1channel 4GB DDR3 memory and 4channel 4GB DDR3 memory. We used several mixed workloads for the evaluations: 1,2,4 thread(s) workload in 1channel model and 1,2,4,8,16 thread(s) workload in 4channel model.

Figure 5 shows the performance improvement from FCFS scheduler and Close scheduler, which are included in usimm simulator kit[2]. Performance is defined as total execution time to finish whole the applications in each configuration. Our scheduler improves the performance 14.2% in maximum and 10.9% in average from FCFS scheduler. It improves the

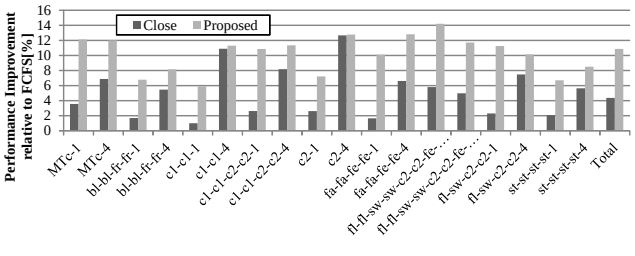

Figure 5: Performance Improvement

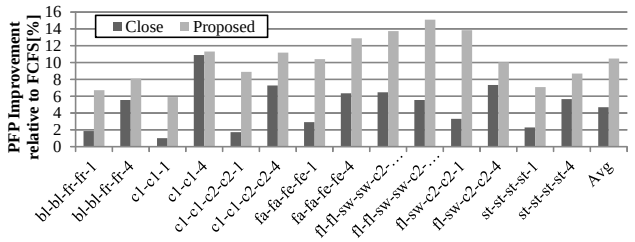

Figure 6: PFP

performance 8.8% in maximum and 6.2% in average from Close scheduler. This result shows that our scheduling policy is good in 1 channel memory and in 4 channels memory with same or more than 4 threads. So our scheduler is very effective especially in bandwidth sensitive situations.

3.2 PFP

Figure 6 shows the improvement of PFP (Performance-Fairness Product) from FCFS scheduler and Close scheduler. Our scheduler improves the PFP 13.8% in maximum and 10.5% in average from FCFS scheduler. This result shows that our scheduling policy is also very good in PFP metric due to its high performance in bandwidth sensitive situations.

3.3 EDP

Figure 7 shows the improvement of EDP (Energy-Delay Product) from FCFS scheduler and Close scheduler. Our scheduler improves the EDP 30.7% in maximum and 18.9% in average from FCFS scheduler. Though our scheduler requires the extra hardware budget for implementation, it can reduce the total EDP, which derives from the performance improvement.

3.4 Hardware Cost

We evaluated the hardware cost for the implementation of our scheduler in terms of storage budgets.

TCM requires a 20-bits read counter per thread, a 5-bits priority counter per thread, a 64-bits total read counter per thread and a 64-bits latest read time register per thread. Therefore the total storage budget required for TCM is 2448 bits (306 bytes).

GHT and LRR require extra storage budgets to track the history of past cache-misses. In our scheduler, the number

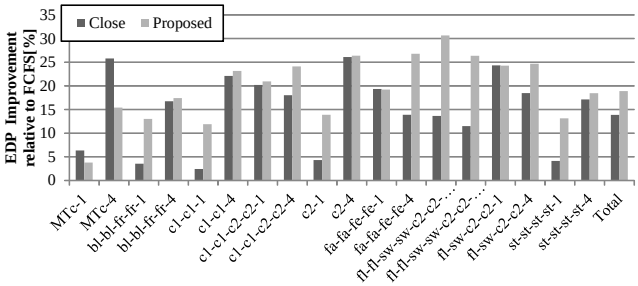

Figure 7: EDP

of GHT entries is 512 per thread. The size of a entry is 40-bits (1 bit for valid bit, 32 bits for PC, 7 bits for saturated reference counter). Therefore, total storage budget of GHTs for 16-threads is totally 327,680 bits (40 bits x 512 entries x 16 threads, 40K bytes). LRR is a small register table of up to 64 bits (32 bits for PC and 32 bits gain count).

OTHER components require a 8-bits counter of wait cycles for drain writes per bank (512 bits, 64 bytes).

TOTAL BUDGET is up to 41K bytes in the 16 threads setting. It satisfies the competition rule, up to 68K bytes.

4. CONCLUSION

This paper described our memory scheduler combining the 3 distinct algorithms. Our scheduler aims to improve the performance, the good fairness and the good energy efficiency by employing multi-grain prioritization mechanisms. The evaluation result showed that our scheme improves the performance 10.9% in average from FCFS scheduler. The result showed that our scheme improves the PFP value 10.5% in average from FSFS scheduler in the same configuration. The result also showed that our scheme improves the EDP value 18.9% in average from FCFS scheduler.

Acknowledgment

The present study was supported in part by Core Research for Evolutional Science and Technology (CREST), JST.

5. REFERENCES

[1] M. Awasthi, D. W. Nellans, R. Balasubramonian, and A. Davis. Prediction based dram row-buffer management in the many-core era. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 183–184, Washington, DC, USA, 2011. IEEE Computer Society.

[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] N. Chatterjee, N. Muralimanohar, R. Balasubramonian, A. Davis, and N. Jouppi. Staged reads: Mitigating the impact of dram writes on dram reads. In *High Performance Computer Architecture*

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	373	NA	NA	NA	4.23	3.98	4.08
MT-canneal	4 chan	179	167	160	NA	NA	NA	1.78	1.56	1.54
bl-bl-fr-fr	1 chan	149	147	140	1.20	1.18	1.13	0.50	0.48	0.44
bl-bl-fr-fr	4 chan	80	76	74	1.11	1.05	1.03	0.36	0.32	0.31
c1-c1	1 chan	83	83	79	1.12	1.11	1.06	0.41	0.40	0.37
c1-c1	4 chan	51	46	46	1.05	0.95	0.95	0.44	0.36	0.36
c1-c1-c2-c2	1 chan	242	236	218	1.48	1.46	1.36	1.52	1.44	1.26
c1-c1-c2-c2	4 chan	127	118	114	1.18	1.10	1.06	1.00	0.85	0,80
c2	1 chan	44	43	41	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	207	1.52	1.48	1.38	1.19	1.14	1.00
fa-fa-fe-fe	4 chan	106	99	94	1.22	1.15	1.08	0.64	0.56	0.50
fl-fl-sw-sw-c2-c2-fe-fe	4 chan	295	279	258	1.40	1.31	1.23	2.14	1.88	1.64
fl-fl-sw-sw-c2-c2-fe-fe--bl-bl-fr-fr-c1-c1-st-st	4 chan	651	620	583	1.90	1.80	1.65	5.31	4.76	4.20
fl-sw-c2-c2	1 chan	249	244	224	1.48	1.43	1.30	1.52	1.44	1.22
fl-sw-c2-c2	4 chan	130	121	118	1.13	1.06	1.03	0.99	0.83	0.79
st-st-st-st	1 chan	162	159	152	1.28	1.25	1.19	0.58	0.56	0.52
st-st-st-st	4 chan	86	81	790	1.14	1.08	1.05	0.39	0.35	0.33
Overall		3312	3173	2987	1.30 PFP: 3438	1.24 PFP: 3149	1.18 PFP: 2812	23.88	21.70	20.08

Table 1: 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.

(HPCA), 2012 IEEE 18th International Symposium on, pages 1 –12, feb. 2012.

[4] 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.

[5] 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*, pages 1 –12, jan. 2010.

[6] Y. Kim, M. Papamichael, O. Mutlu, and M. Harchol-Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, pages 65 –76, dec. 2010.

[7] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Microarchitecture, 2007. MICRO 2007. 40th Annual IEEE/ACM International Symposium on*, pages 146 –160, dec. 2007.

[8] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enabling high-performance and fair shared memory controllers. *Micro, IEEE*, 29(1):22 –32, jan.-feb. 2009.

[9] V. Stankovic and N. Milenkovic. Dram controller with a close-page predictor. In *Computer as a Tool, 2005. EUROCON 2005.The International Conference on*, volume 1, pages 693 –696, nov. 2005.

[10] Z. Zhang, Z. Zhu, and X. Zhang. A permutation-based page interleaving scheme to reduce row-buffer conflicts and exploit data locality. In *Proceedings of the 33rd annual ACM/IEEE international symposium on Microarchitecture*, MICRO 33, pages 32–41, New York, NY, USA, 2000. ACM.