Thread-Fair Memory Request Reordering

Kun Fang, Nick Iliev, Ehsan Noohi, Suyu Zhang, and Zhichun Zhu

Dept. of ECE, University of Illinois at Chicago

{*kfang2,niliev4,enoohi2,szhang33,zzhu*}@*uic.edu*

Abstract

As an important part of modern computer system, the main memory is responsible of storing programs and data structures needed for executing the programs. Performance, power consumption and capacity are the three major factors of a memory system design. Among them, performance and power consumption can be improved by carefully reordering concurrent memory requests to reduce their average latency and wisely utilizing the memory power-saving modes to reduce power consumption.

We have implemented a "Thread-Fair" memory request scheduling policy on top of the USIMM [1] memory simulation infrastructure to enter the memory scheduling competition. The scheduler give the read requests blocking the ROBs' (reorder buffer) head high priority when choosing among multiple requests who want to perform row access. In this way, the most critical request is serviced faster and none memory intensive thread is less likely being interfered by memory intensive thread. Also, our scheduler uses a "Read Hit Queue" and "Write Hit Queue" per memory bank to group requests hitting on the same row buffer together. By giving the "hitting group" higher priority over the "page missing" requests, the scheduler can maximize the row buffer hit rate and hence reduce average memory access latency and power consumption. Using provided single program and multi-thread workloads, the simulation results indicate that our scheme performs 8.7% than the FCFS baseline. The fairness metric is improved by 9.1% and the energy delay product EDP is reduced by 17.3% on average.

1. Introduction

The DRAM sub-system in modern CMPs is a critical resource which has to be managed in real time as a shared resource between all active threads running on all cores. A conventional memory system usually consists of several memory channels. Each memory channel can process memory requests and transmit data independently. Within each channel, several memory devices form a memory rank to provide a 64-bit data path and serve requests as a whole. There are normally 4-8 banks per rank and the data can be pin-pointed by sending the row address and column address to the destination bank. To get the data, an activation command needs to be sent to the bank to bring a row containing the required data to the row buffer first. Then, a column access command is responsible to get the required data in/out of the bank. Following requests to the same row can directly issue column access commands to drive the data without another activation. However, if the subsequent request goes to a new row, a precharge command should be issued before activation to store the data in the row buffer back to the bank.

In the uniprocessor era, a single core running a single thread had un-constrained access to the entire DRAM sub-system. The goal of memory access scheduling was to maximize DRAM throughput, so that system throughput could be maximized as well. The FR-FCFS scheduling policy [9] achieved this by prioritizing row-hit requests over other requests. However this policy does not scale well when multiple threads are active and competing for the same DRAM resource [5], [8], [2].

Several recent schedulers attempt to improve the handling of memory requests from multiple competing threads, so that overall fairness and throughput (quality of service) are maintained to an acceptable level. For instance, the stall-time fair memory scheduler [6], estimates the slowdown of each thread, compared to when it is running alone in the system, and prioritizes the thread that has been slowed down the most. Parallelism-aware batch scheduling [7], forms a batch of requests over a fixed interval, and selects the thread with the fewest requests to different banks in each batch to improve system throughput. The ATLAS

scheduler [3] aims to maximize system throughput by prioritizing threads that have attained the least service from the memory sub-system.

The Thread Cluster Memory Scheduling algorithm [4], attempts to provide the best throughput and best fairness to all threads by distinguishing latency or bandwidth sensitive threads.

Our scheduler implements a "Thread-Fair" scheduling scheme that prioritizes reads which is blocking the ROB head over the ones queued in the middle of ROB when opening a row. Hence the pipeline doesn't get stuck because of the head request is delayed. This policy helps the thread fairness by giving each thread equal opportunity accessing memory banks. Besides of that, we also prioritize read requests over writes and tries to exploit row hits as much as possible using a "Read Hit Queue" and "Write Hit Queue" per memory bank to group requests hitting on the same row buffer together. Then the memory controller can choose to issue column command in a group so unnecessary bank precharges and activations can be avoided to reduce memory access latency. If there is no column access ready in current cycle, which means no row buffer hits are available and all ROB head are serviced, the memory controller will issue other commands of memory requests in the order of their arrival time.

We implement our scheduler on the memory simulator infrastructure USIMM [1] and use provided single program and multi-thread workloads to test our scheduler. The simulation results indicate that our scheme outperforms the FCFS baseline by 8.7%. The fairness metric is improved by 9.1% and the energy delay product EDP is reduced by 17.3% on average.

2. Scheduler Design

2.1. Scheduling Policy

The "Thread-Fair" memory request scheduling policy tries to let each thread has equal opportunity being serviced by memory system. This is achieved by giving the requests generated by the head entry of ROB higher priority when opening a row. The requests coming from the ROB head will be serviced faster and pipeline will less likely being blocked. The "Thread-Fair" policy ensures each thread is not heavily delayed by memory system if some memory intensive workload is executing.

The other factor we would like to optimize it row hit. Because row buffer hits have much shorter latency and consume less power than row buffer misses, our scheduler tries to exploit row buffer hits as much as possible. We build one "Row Hit Queue" for read requests and another one for write requests on memory bank to group requests to the same row together so that row buffer hits can be picked out and be prioritized. Another important issue that we want to optimize is to reduce the latency between row transitions. If a request misses on an open row, it will issue a precharge command followed by an activation to bring the required data into row buffer. In this case, the request's access latency will be long. On the other hand, simple close-page policy will eliminate any chance of row hits. Thus, we decide to use a clever page management scheme that works perfectly with the Row Hit Queue. It closes an open row after the last column hit to the row is issued. In this way, the following request does not need to do the precharge operation while at the same time, possible row hits are maximized.

In summery, our scheduler performs following rules in order:

1) Read is processed before writes (read first) unless the "write first" rule is triggered.
2) When the write queue is about to be full (with only four free entries) , process writes before reads until the write queue has at least fourteen free entries (write first).
3) When at "read first", try to issue row hits first; if there are no row hits, try to issue request generated by ROB head in round-robin.
4) If the requests from ROB is already in service, issue reads based on FCFS; and if no reads can be processed, issue write hits.
5) When at "write first", try to issue row hits first; if there are no row hits, issue writes based on FCFS; and if no writes can be processed, issue read hits.
6) Auto-precharge the open row together with the last column access if there are no requests pending on the open row.

2.2. Scheduling Implementation

Firstly, the memory controller will maintain two queues, one for reads and the other for writes to track all the requests received from the last level cache. We call them "Read Queue" (RQ) and "Write Queue" (WQ). Secondly, to find out and reorder the requests to the same row so that the row buffer hit rate is optimized, we build a "Read Row Hit Queue" (RRHQ) and a "Write Row Hit Queue" (WRHQ) for each bank. Also, a "Read Pending queue" (RPQ) and a "Write pending queue" (WPQ) are setup for every bank to store requests mapped to it. Every time when a request is inserted into the memory controller, its index of the Read/Write Queue in memory controller and

row address will also be inserted into the Read/Write Pending Queue of its mapped bank. So basically the requests are queued on its mapped bank. As shown in Figure 1, each request index is queued on the destination bank and linked to itself in the memory controller queue so that the full request information can be fetched.

In order to maximize the row hits, a scheduling policy is implemented for each bank based on its current status. Because read requests are more critical than write requests, the scheduler will always try to prioritize reads over writes unless the write queue in memory controller is about to be full. If the number of free entries in the write queue is below a pre-determined threshold, the scheduler will prioritize writes over reads to free some space for incoming writes and prevent processor stall due to write queue full. In our scheduler, we set the threshold for starting write over read to when the write queue has filled up 60 out of the total 64 entries. At this point, all write requests get higher priority than read requests. When the write queue has less than 50 entries filled, the scheduler will change back to read first mode.

At read first, each bank will check its own state. If it is closed, memory controller first check if any requests generated by the ROB head is still not serviced and matches the closed bank. If yes, they will be issued at the highest priority. This is where the "Thread-Fair" scheduling is enforced. The ROB head information can be get by passing from CPU to memory controller the request ID or physical address of ROB head each time the ROB retires a read instruction. To simplify the hardware, and preventing the CPU transmitting the ROB head information, a simplified scheme can assume the oldest request from each thread is coming from the ROB head. Modern CPU tries to get the instruction level parallelism by aggressively exploiting out-of-order and runahead execution. These techniques somewhat make the oldest request not be the ROB head. However, the oldest request is very likely to be close to the ROB head so the approximation is effective.

If no, it will pick the first request in RPQ as the best candidate to issue a activation command. If the memory controller picks the request to process, then for each memory cycle, requests in RPQ of this bank are scanned and the RQ indexes of any requests whose row addresses matches that in the row buffer will be inserted into RRHQ. In this way, all the read requests hitting on the open row will be grouped into the RRHQ. At the same time, requests in WPQ are scanned and all hitting WQ indexes will be inserted into the WRHQ. If the bank is activated, the best candidate request is picked from RRHQ. If RRHQ is empty, the best candidate request is picked from WRHQ. Scheduling using RRHQ and WRHQ is for the purpose of maximizing row hit and is called Hit-First.

The scheduling of Hit-First when WQ is about to be full is very similar to the policy at read first except that the candidate request to issue activation command is picked from WPQ and the candidate hit is first picked from WRHQ and then from RRHQ if WRHQ is empty.

Each bank will check the timing constraint of the selected request and inform the memory channel when the request is ready to issue. For each channel, it will look through the candidate requests picked by each bank and set their priorities based on the same Read-First, Hit-First policy. If more than one banks pick requests at the same priority level, the highest priority goes to the one with the oldest request (FCFS). The memory controller will process the request with the highest priority in each channel and let the bank issue command accordingly.

2.3. Page Management

There are two basic policies for row buffer management: open page and close page. If the row buffer is left open after a column access, the following requests to the same row can fetch/store data to the row buffer without issuing precharge or activation command. Thus, the access latency can be reduced. However, requests to the same bank but different rows will take longer to finish than under the close page policy, which closes the row buffer (precharges) right after a column access. We decide to use a greedy algorithm that working together with our scheduler to group memory requests to the same row together. If there is pending requests to the open row, the page is left open to maximize row hits. If there is no pending request to the latest accessed row, an auto-precharge command is issued together with a column read/write command to close the page so that requests to the same bank but different rows will not to penalized by the precharge command. Because each bank has RRHQ and WRHQ, the policy can be simplified to: if the current column access is the last request of RRHQ and WRHQ, an auto-precharge command is issued together with this column access.

2.4. Hardware Cost

We estimate the hardware cost based on a 4-channel memory system with 2 ranks per channel and 8 banks per rank. So there are totally 64 banks in the memory system. The RPQ and WPQ are 32 entries for each

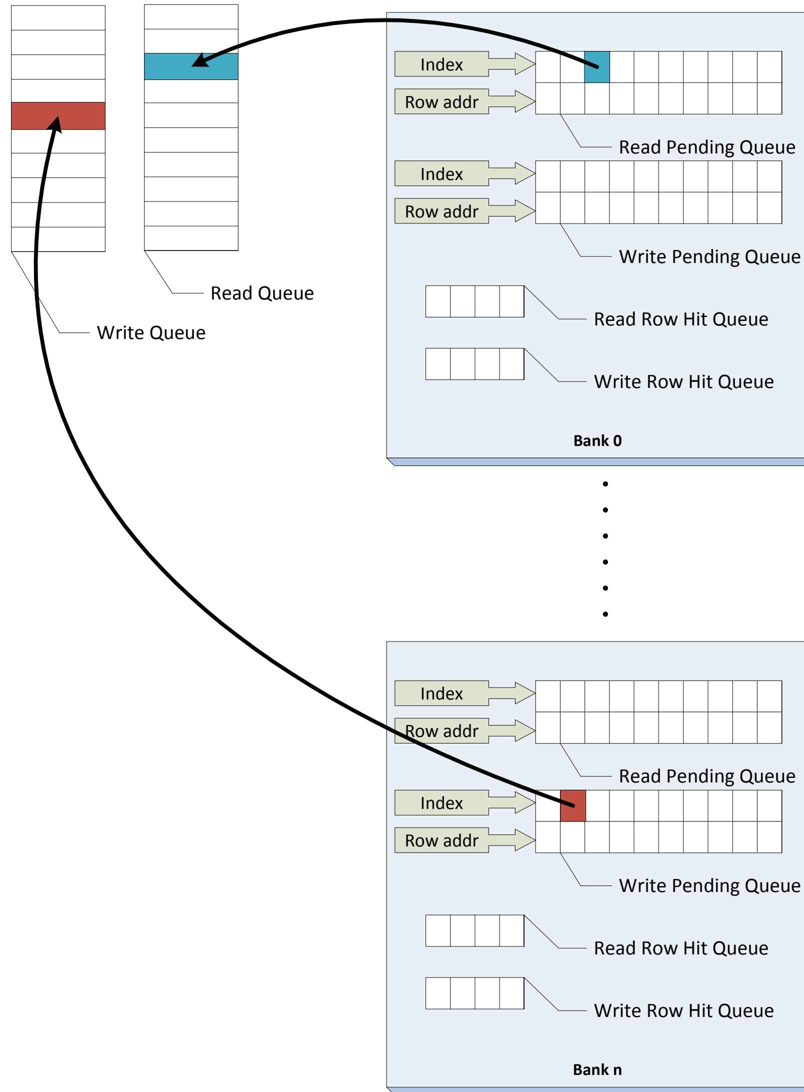

Figure 1: Overview of scheduler implementation.

bank and each entry will contain 6-bit RQ/WQ (64-entry) index and 16-bit row address (65536 rows). Thus, the total storage overhead for RPQ and WPQ is 11KB. The RRHQ and WRHQ only stores RQ/WQ index, which is 6-bit. If they are 32-entry queues, the total overhead is 3KB. The total storage overhead of our scheduler is 14KB.

3. Experimental Results

We test our proposed scheduler on a memory simulation infrastructure: USIMM [1]. The simulator reading traces generated by CPU functional simulation and model DDR SDRAM memory system in detail. The timing model and power model are both implemented.

Running the competition trace and configuration provided by USIMM developer the simulation results for 1 channel and 4 channel are shown in table 1. The recommended metrics (Sum of execution time; Max slowdown and EDP) were used to compare the scheduler with the baseline.

The performance of our proposed scheduler outperforms the FCFS from 2.5% to 13.3% (8.7% on average). The overall execution time reduces by 9.7%. This is the result of grouping row hits and letting reads bypass writes. The read page hit rate ranges from 0.2% (c2-4channel) to 64% (fa-fa-fe-fe-1channel), which shows our scheduler did a good job of grouping row hits for 1 channel configuration. The low hit rate on 4channel is because the memory mapping is changed

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	377	NA	NA	NA	4.23	3.98	3.48
MT-canneal	4 chan	179	167	155	NA	NA	NA	1.78	1.56	1.34
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	81	1.12	1.11	1.09	0.41	0.40	0.39
c1-c1	4 chan	51	46	47	1.05	0.95	0.95	0.44	0.36	0.36
c1-c1-c2-c2	1 chan	242	236	219	1.48	1.46	1.37	1.52	1.44	1.24
c1-c1-c2-c2	4 chan	127	118	114	1.18	1.10	1.06	1.00	0.85	0.79
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	0.39	0.40
fa-fa-fe-fe	1 chan	228	224	206	1.52	1.48	1.37	1.19	1.14	0.98
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	260	1.40	1.31	1.21	2.14	1.88	1.63
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	1.66	5.31	4.76	4.14
fl-sw-c2-c2	1 chan	249	244	225	1.48	1.43	1.29	1.52	1.44	1.19
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	151	1.28	1.25	1.19	0.58	0.56	0.51
st-st-st-st	4 chan	86	81	79	1.14	1.08	1.05	0.39	0.35	0.33
Overall		3312	3173	2990	1.30	1.24	1.18	23.88	21.70	19.17
PFP					3438	3149	2816			

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.

to exploit MLP instead of row hitting rate.

Our "Thread-Fair" scheduler also does well on the fairness metric. It improves the max slowdown of threads for 9.1% (from 3.1% to 13.6%). Our "Thread-Fair" scheduler give concurrent running threads equal opportunities to retire the ROB head entry so the following "instructions" is less blocked. All workloads produces the squared deviation of slowdown for all threads for less than 2%. The only exception is the 16-thread workload which has a squared deviation of 9.8%. The result show that our proposed scheduler can well manage concurrent threads' slowdown and avoiding some of them running too fast or too slow.

Our scheduler can also improve the energy delay product. Compared to FCFS, the EDP is improved from 5.2% to 24.6% (17.3% on average). It is because the smart auto-precharge can maximize the bank precharge time and grouping hits can also reduce operation power.

4. Conclusion

In this paper, we proposed a memory request reordering scheme to enter the memory scheduling contest. The scheduler can improve thread fairness and system performance by monitoring and prioritizing the requests from ROB head. By grouping requests hitting the same row buffer, our scheduler can efficiently reduce memory access latency. The result show our scheme performs much better than the competition baseline.

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] E. Ipek, O. Mutlu, J. F. Martinez, and R. Caruana. Self-optimizing memory controllers: A reinforcement learning approach. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 39–50, 2008.

[3] Y. Kim, D. Han, O. Mutlu, and M. Harchol-balter. Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers. In *Proceedings of the 16th International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–12, 2010.

[4] Y. Kim, M. Papamichael, O. Mutlu, and M. H. Balter. Thread cluster memory scheduling: Exploiting differences in memory access behavior. In *Proceedings of*

the 43nd Annual IEEE/ACM International Symposium on Microarchitecture, pages 65–76, 2010.

[5] T. Moscibroda and O. Mutlu. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors. In *Proceedings of the 40th International Symposium on Microarchitecture*, Dec. 2007.

[6] O. Mutlu and T. Moscibroda. Stall-time fair memory access scheduling for chip multiprocessors. In *Proceedings of the 40th International Symposium on Microarchitecture*, pages 146–160, December 2007.

[7] O. Mutlu and T. Moscibroda. Parallelism-aware batch scheduling: Enhancing both performance and fairness of shared DRAM systems. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, pages 63–74, 2008.

[8] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair Queuing CMP Memory Systems. In *Proceedings of the 39th International Symposium on Microarchitecture*, pages 208–222, Dec. 2006.

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