PAR-BS BASED PREDICTIVE ROW CLOSE MEMORY SCHEDULER FOR DRAM

A Project

Presented to the faculty of the Department of Computer Science

California State University, Sacramento

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

SOFTWARE ENGINEERING

by

Kautilya Aswal

SPRING
2013
PAR-BS BASED PREDICTIVE ROW CLOSE MEMORY SCHEDULER FOR DRAM

A Project

by

Kautilya Aswal

Approved by:

__________________________________, Committee Chair
Dr. Du Zhang

____________________________, Committee member
Prof. Bob Buckley

____________________________
Date
Student: Kautilya Aswal

I certify that this student has met the requirements for format contained in the University format manual, and that this project is suitable for shelving in the Library and credit is to be awarded for the project.

________________________________________, Graduate Coordinator

Dr. Behnam Arad

Date

Department of Computer Science
Abstract

of

PAR-BS BASED PREDICTIVE ROW CLOSE MEMORY SCHEDULER FOR DRAM

by

Kautilya Aswal

In a Chip Multiprocessor System (CMP) DRAM system is the one, which is shared by the multiple cores of the system. Each core has several threads running on them. Request from a thread, cannot only delay request from other threads but can also destroy the request from other threads as well. The primary reasons can be-

- Bank Conflicts
- Bus Conflicts
- Row Buffer Conflicts

As the gap between memory and processor performance continues to widen, it becomes imperative that new techniques, which improve memory performance, be proposed. This research describes the functioning of our high-performance Memory scheduler.

The scheduler implements a parallelism aware batch-scheduling algorithm (PAR-BS). This research takes ideas from existing scheduling heuristics and combines them with
new optimizations to obtain a scheduler with row based closure policy and smart read to write switching heuristics.

The performance of the scheduler will be measured on 3 bases: -

- Delay
- Energy-Delay Product (EDP)
- Performance-Fairness Product (PFP)

This research proposes a new approach in designing the Scheduler that makes sure about having the fairness and providing a good quality of service to the running threads while keeping system throughput in mind. We can divide the functioning of our shared DRAM memory scheduler in two ways. First, PAR-BS processes DRAM request in batches and it will not allow the other batch to go until all the threads in previous batch have been satisfied. Second, to optimize system throughput it employs a parallelism aware batch scheduling policy that aims to process requests from a thread in parallel in the DRAM banks, hence reducing memory related stall time experienced by the thread.
In addition to the already existing features of the scheduler, I tried to implement several new enhancement features that I think helped me achieve better memory scheduling algorithm than the already existing ones.

The evaluation of our DRAM memory scheduler will be based on comparing its performance to the already existing ones like First Ready-First Come First Serve Base (FR-FCFS) and Close Page Policy (CPP) Algorithm. This can be achieved by running all the Schedulers with a set of trace files on the USIMM V3 Simulator [5]. Trace files, in general, are usually logs produced by a program. A polite program will erase them when done with them; sometimes they stick around. Most often, these are ASCII text files which you can examine in a text editor. In our project we are using 10 trace files like comm2, comm1 comm1, fluid swapt comm2 comm2 , stream stream stream stream etc which are embedded inside the simulator just to perform the experiment and measure the performance of our scheduler with default one. USIMM [5] is a simulation infrastructure that models the memory system and interfaces it with a trace based processor model and a memory scheduling algorithm.

Our scheduler yields a performance improvement of 5.19% over a baseline of FCFS scheduler. In addition, our scheduler has an improvement of 10.00% in energy-delay product (EDP) and performance-fairness (PFP) improvement of 10.55% compared to the baseline FCFS scheduler. I have also compared the performance of our scheduler with
another popular memory scheduling algorithm known as Close Page Policy scheduler. The results I got after running our scheduler on USIMM simulator [5] are pretty close to the numbers I got from running CPP scheduler, but still sufficient enough to prove it superior than CPP scheduler. Our scheduler yields a performance improvement of 1.21% over CPP scheduler. It also improves energy delay product and performance fairness product by 2.34% and 1.25% respectively as compared to the CPP scheduler.

_______________________, Committee Chair
Dr. Du Zhang

_______________________
Date
ACKNOWLEDGEMENTS

While working on this project many people have helped me achieve my goal. I would like to thank Dr. Du Zhang, for investing his invaluable time reviewing this project. Without his support, it would have been difficult for me to complete this project.

I would also like to thank Professor Bob Buckley for giving me his time and input as second reader. He was a blessing in disguise for me.

I would like to thank my family and Friends, for their undying love and support. In addition, I would like to thank all my professors, who taught me. Without their assistance and guidance, my Master’s degree would not have been completed successfully.
# TABLE OF CONTENTS

| Acknowledgement | ................................................................................................................... ix |
| List of Tables  | .................................................................................................................... xii |
| List of Figures | ..................................................................................................................... xiii |
| List of Graphs  | .................................................................................................................... xiv |

## Chapter

1. INTRODUCTION ........................................................................................................ 1
   1.1 Problem Statement ............................................................................................. 2
   1.2 Solution .............................................................................................................. 3
   1.3 System Configuration ......................................................................................... 5
   1.4 Ultimate Objective Solution ............................................................................... 7
   1.4.1 Performance Metrics Solution ....................................................................... 8
   1.5 Report Organization ........................................................................................... 9

2. BACKGROUND OF THE STUDY ............................................................................... 10
   2.1 Related Work .................................................................................................... 10
   2.2 Some Possible Memory Scheduling Algorithm Heuristics ................................ 10
      2.2.1 FR-FCFS .................................................................................................... 10
      2.2.2 STFM ......................................................................................................... 11
      2.2.3 PAR-BS ..................................................................................................... 11
      2.2.4 CPP ............................................................................................................ 11
   2.3 Background of DRAM Memory Controller ....................................................... 12
3. DESIGN AND IMPLEMENTATION OF THE SIMULATOR .................................. 15
   3.1 Parallelism Aware Batch Scheduling .................................................. 15
   3.2 Open Page Policy with History- Based Row Closure .............................. 18
   3.3 Opportunistic Refresh of Ranks ......................................................... 20
   3.4 Heuristics for Switching Between Read and Write Modes ...................... 23
   3.5 Two-Level Watermark Switching Policy from read mode to write mode ....... 24
   3.6 Detailed Explanation of Code ............................................................ 25

4. PERFORMANCE CHARTS EVALUATION AND RESULTS .......................... 33
   4.1 Percentage Performance Improvement over FCFS ................................ 39
   4.2 Percentage Reduction in EDP over FCFS ........................................... 40
   4.3 Percentage reduction in Maximum Slowdown over FCFS ....................... 41
   4.4 Percentage Performance Improvement over CPP ................................... 42
   4.5 Percentage reduction in maximum Slowdown over CPP .......................... 43
   4.6 Percentage reduction in EDP over CPP .............................................. 44
   4.7 Result ................................................................................................. 45

5. CONCLUSION & FUTURE WORK .............................................................. 47
   5.1 Conclusion ............................................................................................ 47
   5.2 Future work .......................................................................................... 47

Appendix ........................................................................................................ 48
Bibliography .................................................................................................... 54
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Tables</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Parameters for Hardware Used</td>
<td>17</td>
</tr>
<tr>
<td>2. Performance &amp; Evaluation Metrics</td>
<td>34</td>
</tr>
<tr>
<td>Figures</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
</tr>
<tr>
<td>1. DRAM Bank Structure</td>
<td>4</td>
</tr>
<tr>
<td>2. Handling of Latencies over Single Core System Configuration</td>
<td>6</td>
</tr>
<tr>
<td>3. DRAM Devices with Aggressive Core</td>
<td>7</td>
</tr>
<tr>
<td>4. Flow Chart for Opportunistic Refresh in Ranks for DRAM</td>
<td>22</td>
</tr>
</tbody>
</table>
# LIST OF GRAPHS

<table>
<thead>
<tr>
<th>Graphs</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Percentage Performance Improvement over FCFS</td>
<td>39</td>
</tr>
<tr>
<td>Percentage Reduction in EDP over FCFS</td>
<td>40</td>
</tr>
<tr>
<td>Percentage Reduction in Maximum Slowdown over FCFS</td>
<td>41</td>
</tr>
<tr>
<td>Percentage Performance Improvement over CPP</td>
<td>43</td>
</tr>
<tr>
<td>Percentage Reduction in Maximum Slowdown over CPP</td>
<td>44</td>
</tr>
<tr>
<td>Percentage Reduction in EDP over CPP</td>
<td>4</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

We use memory-scheduling heuristics to improve access patterns based on past memory reference events. A good memory scheduler manages and schedules requests from a pool of waiting memory requests such that the high overall system throughput is achieved while ensuring fairness[1,3,4] and lower levels of power consumption subject to various timing and DRAM-based parameters.

We have started with a baseline of FR-FCFS (First ready-first come first served) scheduler and incrementally added series of optimizations based on pre-existing heuristics. We have used parallelism aware batch scheduler (PAR-BS) as the basic model for our scheduler. The PAR-BS schedules read requests in batches to ensure fairness and avoid starvation. Inside a batch, we use a thread-ranking scheme, which prioritizes non-memory intensive threads over memory intensive threads, and thus helps in improving throughput and overall utilization. Scheduling requests from a thread together helps avoid interference from other threads and preserves bank level parallelism. We define Bank-level parallelism (BLP) as the average number of requests being serviced in the DRAM banks when there is at least one request being serviced in the DRAM banks Row hits are still prioritized to minimize access latency. Writes use a simple FR-FCFS scheduler, which prioritizes row hits over row conflicts.
1.1 Problem Statement

The DRAM system is a major shared resource among multicore subsystem. When accessing the shared DRAM resource, different thread running on different cores can make the processing delay by data bank/row buffer conflicts. Bank conflicts occur when bank preparation fails to open the requested row in a bank. This occurs due to the timing constraints of the memory. Row-buffer conflicts occur when a sequence of requests on different rows goes to the same memory bank, causing much higher memory access latency than requests to the same row or to different banks. In addition inter thread interference can destroy bank level parallelism of individual threads, because due to interference, scheduler may schedule requests from same threads continuously without giving other threads opportunity to ask for the resources. This can happen, when a thread is producing requests, which are hitting in open row buffer.

Moreover, some less important threads can get an unfair priority over other important threads. These important threads can tend to starve for very long time. We need to come up with the some kind of memory scheduling algorithm that can ensure fairness and performance of the multiple thread running at the same time without ignoring system throughput.
1.2 Solution

This research will propose a new approach for providing fair and high performance DRAM scheduling. A good memory scheduler manages and schedules requests from a pool of waiting memory requests such that the high overall system throughput is achieved while ensuring fairness and lower levels of power consumption subject to various timing and DRAM-based parameters.

Open paged policy with predictive row closure has been used in our scheduler. Current implementations and proposals tend to group into two classes with respect to row buffer usage. Leaving a row buffer open after every access (Open-page policy) enables more efficient access to the open row, at the expense of increased access delay to other rows in the same DRAM array. Open-page policies attempt to gather multiple requests into each row buffer access by speculatively delaying the precharge operation in an effort to execute additional row reads. This enables latency, scheduling, and power improvements possible with page-mode accesses. However, as these policies are applied to the numerous request streams of a many-core system, priority is given to accesses to already opened pages; introducing memory request priority inversion and potential thread fairness/starvation problems. Closed-page policies avoid the complexities of row buffer management by issuing a single access for each row activation. This class of policies provides a consistent fair latency at the expense of potential page-mode gains. The
histories can include access numbers or access durations.

In order to select a victim for closure, we identify banks, which have dead rows or the rows, which have not been used for a certain period. You can get a general idea about the structure of DRAM bank from Fig.1 described below. For defining the closing period, the scheduler records the average access interval of bank and compares the time to the threshold time. If the interval exceeds the threshold, the memory controller issues a Precharge command. Precharge is to bring data into row buffer of a bank. This operation is done to prepare a bank for future accesses.

The decisions regarding when to switch between read and write modes play an important role in determining the overall performance of the memory-scheduling algorithm. There

Figure.1: DRAM Bank Structure
is a Write to Read time delay associated between the end of a write data burst and the start of a column-read command. A column read command moves data from the array of sense amplifiers of a given bank to the memory controller. In a typical, modern DRAM device, each row access command bring thousands of data bits. The frequent switching between read and write mode can cause major performance drawbacks. In our algorithm, we use average time spent in read and write mode and mode factor to find the optimum time for switching between modes. For each channel, the average time spent in the read and the write mode is monitored. The scheduler makes sure that the time before switching from read mode to write mode is always greater than the average time spent in the read mode divided by the mode factor.

1.3 System Configuration

This research will focus on two main system configurations. The first uses a smaller scale processor core and a single memory channel, while the second uses a more aggressive processor core and four memory channels.

First configuration requires the scheduling algorithm to take advantage of single command channel by maximizing the burst mode performance where each open page in row buffer can be used to service multiple requests. In a single core system, a thread has exclusive access to the DRAM banks, so its concurrent DRAM accesses are serviced in
parallel as long as they are not to the same bank. This is illustrated in the simple, conceptual example in Fig. 2

**Figure 2:** Handling of latencies over single core system configuration

In a single core system, a thread has exclusive access to the DRAM banks, so its concurrent DRAM accesses are serviced in parallel as long as they are not to the same bank. If we look at the figure Request1’s (Req1) latency is hidden by the latency of Request0 (Req0), effectively exposing only a single bank access latency to the thread is processing core. Once Req0 is serviced, the core can commit Load 0 and thus enable the decode/execution of future instructions. When Load 1 becomes the oldest instruction in the window, its miss has already been serviced and therefore the processor can continue computation without stalling.

Second configuration tries to maximize memory access parallelism by scattering consecutive blocks across channels, ranks, and banks as shown in Fig.3.
1.4 Ultimate Objective

The goal for this project is to compare different memory scheduling algorithms in a common framework. Scheduling algorithm designed will be tested using DRAM main memory simulator. Algorithms will be evaluated on three tracks:

(1) Delay (or Performance)

(2) Energy-Delay Product (EDP)

(3) Performance-Fairness Product (PFP)
1.4.1 Performance Metrics Solution

(1) Delay (or Latency): Modern high-performance memory subsystems support a high degree of Concurrency. It is critical that the memory controller be able to produce a schedule that can leverage this potential concurrency to maximize performance.

(2) Energy-Delay Product (EDP): In current processors, DRAM accounts for a significant fraction of power Consumption. Therefore, apart from performance, power and energy are also becoming first order issues while designing memory schedulers for multicore systems.

(3) Performance-Fairness Product (PFP): DRAM memory bandwidth has also become a critical shared resource in a multi-core system, and it is important to efficiently share the memory bandwidth among multiple threads maintaining fairness.

The objective of this research is to find out the optimal scheduling algorithm that achieves the proper balance between the various criteria: performance, energy-delay product and performance-fairness product. By proper balance, It mean that our scheduler will maintain a balance in between all the defined matrices. It will not just focus on the performance aspect of the scheduler but will also make sure about sharing the resources equally among the threads and hence making it fair to use.
1.5 Report Organization

This project report is organized into 5 chapters. First chapter deals with the introduction of the project, the problem statement and its solution and the ultimate objective of the project. Second chapter will talk about the background overview and related work. Third chapter will deal with the design and implementation of our scheduler, where we will be talking more about the technical specification of our scheduler and will be presenting the pseudo code for a better understanding. Forth chapter will be regarding the performance chart evaluation and comparison, where we will present and discuss the graphs generated by the simulator after comparing the output generated by our scheduler with the output generated by other memory scheduling techniques. Finally, fifth chapter will talk about the conclusion of our project and some related future work.
Chapter 2

BACKGROUND OF THE STUDY

2.1 Related Work

Our aim in this research is to have such an optimal memory-scheduling algorithm for shared DRAM that keeps in mind both the performance and fairness of the thread without ignoring the overall throughput of the system.

To achieve this objective we will compare the performance of our scheduler with the already existing schedulers present today. Let us discuss some of the possible memory scheduler algorithm and there shortcomings.

2.2 Some Possible Memory Scheduling Algorithm Heuristics

1) First-ready first-come-first-serve (FR-FCFS), fails to fairly schedule memory requests of equal priority threads running on a multicore processor in an interleaved manner. It does not take into account interference between different threads when making scheduling decisions. Instead, it tries to maximize the data throughput obtained from the DRAM by prioritizing memory requests that hit in the row- buffers of DRAM banks over other requests, including older ones. This leads to starvation or poor memory
performance (increased stall time of memory requests) for some threads and increased performance for others.

2) STFM (Stall Time Fair Memory) scheduler tries to provide fairness to equal priority threads by minimizing the difference between memory related slowdown of different threads. STFM tracks the slowdown of threads and as the difference between minimum slowdown and maximum slowdown in the system exceeds a threshold (the threshold of maximum tolerable unfairness), the scheduler prioritizes memory requests from threads that slow down the most. Otherwise, the scheduler tries to maximize DRAM throughput by using the baseline FR-FCFS scheduling policy. STFM is unable to exploit bank level parallelism.

3) PAR-BS (Parallelism-Aware Batch Scheduling): DRAM requests are organize into batches based on their arrival time and requesting threads. The requests from the oldest batch are prioritized and therefore guaranteed to be serviced before other requests. Within a batch of requests, PAR-BS is parallelism-aware: it strives to preserve bank-level access parallelism of each thread in the presence of interference from other threads’ DRAM requests.

4) Close page policy: From a memory intensity perspective, we classify threads into one of two distinct groups: latency-sensitive or bandwidth-sensitive. Latency-sensitive threads spend most of their time at the processor and issue memory requests sparsely.
Even though the number of generated memory requests is low, the performance of latency-sensitive threads is very sensitive to the latency of the memory subsystem; every additional cycle spent waiting on memory is a wasted cycle that could have been used on computation. Bandwidth sensitive threads experience frequent cache misses and thus spend a large portion of their time waiting on pending memory requests. Therefore, their rate of progress is greatly affected by the throughput of the memory subsystem. Even if a memory request is quickly serviced, subsequent memory requests will once again stall execution. Threads are dynamically assigned into latency sensitive and bandwidth sensitive. By prioritizing latency sensitive threads over bandwidth sensitive threads, we can increase the throughput and additionally using a shuffling priority between bandwidth intensive threads to reduce slowness of interfering threads.

2.3 Background on DRAM Memory Controller

A modern SDRAM chip consists of multiple DRAM banks to allow multiple requests for memory accesses to occur in parallel if they require data from different bank. Each bank consists of 2D array, consisting of rows and columns. Rows typically store data in consecutive memory location, where each block is size 1-2 KB. The data in a bank can be accessed only from the row buffer, which can contain at most one row. Each row can be considered as a bank. Row here does not mean row of bank, it means row buffer of bank, which acts as a cache in front of bank. Each bank has single row buffer.
The amount of time it takes to service a request for DRAM depends on the status of the bank or row buffer. Status of DRAM can be categorized into 3 categories:

- **Row Hits**: if the request is made to the row, which is currently, opened in the row buffer. The controller only needs to issue the command read/write to the bank.

- **Row Closed**: if the request is made to the row, which is currently not active/closed. The controller need to issue an activate command first to open the required row and then issue the read/write command.

- **Row Conflicts**: if the request is made to the row, which is not the one present in the row buffer. The controller needs to first close the row by issuing precharge command and then open the required row, and then issue a read/write command.

A DRAM controller consists of a memory request buffer that buffers the memory request while they are waiting to be serviced. When selecting the next request for service, the scheduler first looks at the state of the DRAM banks and buses. The DRAM command for the request of the resource is acceptable when there is no conflict of resource, such a DRAM command is known as ready.

When multiple threads want to gain access to the shared resources, the FR-FCFS scheduling policy tends to be unfair towards prioritizing the threads. It prioritize high
row buffer locality (i.e. row buffer hit rate) over those with low row buffer locality due to row hit first prioritization rule [4]. It also prioritizes memory intensive thread to non-memory intensive threads due to the oldest first prioritization rule [4]. As a result even though FR-FCFS scheduler achieves high performance for DRAM data throughput but overall the performance degrades as it keeps on starving more important threads for a long period of time, causing unfairness among threads and low overall system throughput.
Chapter 3

DESIGN AND IMPLEMENTATION OF THE SIMULATOR

3.1 Parallelism aware batch scheduling

Our scheduler uses the existing parallelism aware batch-scheduling algorithm (PAR-BS) [1] as the implementation model. It schedules read requests in batches to ensure fairness and avoid starvation. A new batch will be formed only when the requests from an older batch is done. Mark cap is define as the maximum number of requests going to a single bank for a single thread and is used to define the boundaries of a batch.

Inside a batch, we use a thread-ranking scheme, which prioritizes non-memory intensive threads over memory intensive threads and thus helps in improving throughput and overall utilization. Memory intensive threads are the threads, which want access towards the memory aspect of the DRAM. It is completely opposite is non-memory intensive thread. These “light” threads, only use a small fraction of the total available memory bandwidth. On the other hand, shuffling the priority order of memory-intensive threads improves fairness because it ensures no thread is disproportionately slowed down or starved. Scheduling requests from a particular thread together help avoid interference from other threads and preserves bank-level parallelism. Batch requests are prioritized over non-batch requests. The batched requests follow a first-ready first come first served
policy (FR-FCFS) for prioritizing row hits. For ranking threads, the thread having the maximum requests to any given bank is given the lowest rank and vice-versa. This is done by using a counter for every thread. For breaking ties, the thread having greater total number of requests over all the banks in the queue is given lower priority. Writes use a simple FR-FCFS scheduler, which prioritizes row hits over row conflict.

Ensuring that the requests in the batch are prioritized and thus completed, PAR-BS provides strong guarantee against starvation and thus ensures fairness. To achieve this goal, Parallelism-Aware Batch Scheduling (PAR-BS) consists of a component called request batching (BS), or simply batching, component that groups a number of outstanding DRAM requests into a batch and ensures that all requests belonging to the current batch are serviced before the next batch is formed. Batching not only ensures fairness but also provides a convenient granularity (i.e., a batch) within which possibly thread-unfair but high-performance DRAM command scheduling optimizations can be performed. In addition, by isolating requests from a given thread the scheduler is able to exploit bank-level parallelism and hence increases the overall utilization. Further, the PAR-BS avoids penalizing non-memory intensive or computationally intensive threads, which are more prone to DRAM related slowdown by prioritizing them over memory intensive threads.
The table below gives the amount of hardware you will need if you want to implement this algorithm on chip. For example, Read Queue Size 64 means; we need 64-bit register to implement this queue in hardware, same goes for all the other parameters as well.

Our implementation is based on Parallelism aware batch scheduling algorithm. A marked bit augments each request field, rank priority bits which

Table 1: Parameters for hardware used

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read Queue Size</td>
<td>64</td>
</tr>
<tr>
<td>Maximum threads</td>
<td>4</td>
</tr>
<tr>
<td>Number of Channels</td>
<td>4</td>
</tr>
<tr>
<td>Number of ranks/channel.</td>
<td>2</td>
</tr>
<tr>
<td>Number of banks/rank</td>
<td>8</td>
</tr>
</tbody>
</table>

require 2 bits for 4 threads. Hence, the total number of bits added to the read request queue is size of read request queue* 3(192 bits)

The ranking scheme requires two kinds of counters. A set of counters are required to measure the total number of requests for a thread in all the banks. This requires a storage of:-
(Maximum number of request bits) \times (Maximum number of threads) = 24 bits

Maximum no. of request per thread in a batch of PAR BS can be $2^6 = 64$, so 6 bits are required. The second set of per-thread counter measures the number of requests sent by the thread to any given bank. This requires storage of-

$$(\text{Maximum number of threads}) \times (\text{number of channels}) \times (\text{Number of ranks}) \times (\text{Number of banks}) \times (\text{Maximum Request bits}) / \text{bank} = 1536 \text{ bits}$$

Additionally scheduler would require a configurable mark_cap register of 5 bits.

Hence the additional storage required for implementing PAR-BS is:-

$$192 + 24 + 1536 + 5 = 1757 \text{ bits} \approx 220 \text{ bytes}$$

3.2 Open page policy with history-based row closure

In order to close/precharge a bank the scheduler employs a method, which detects a dead row that has not satisfied any requests for a period called the threshold. This is done so that the rows, which are not satisfying any incoming requests, do not have to pay the precharge penalty when a new request arrives. The Scheduler records run time average of access interval for each of the banks in memory. This average value is compared with the
time that has been passed after last access to the bank. If this interval exceeds a threshold, a row close (precharge) command is issued. A per bank average monitoring allows algorithm to predict row closing based on bank specific access patterns.

Before closing a row, i.e. writing it back to the bank, we crosscheck our speculation with the read and write queue in order to ensure accuracy in the above scheme. We check both the write queue and read queue for any row request that can be satisfied by the row that we were closing speculatively. If we find any such request, we do not close that row.

This method requires registers to store last access time and last access interval for each of the banks in the memory. The scheduler needs to compare the last access interval of the row buffer with the interval since the last access for the buffer to make decision whether to close the row or not. This decision-making logic is very simple to implement in the hardware requiring comparators and few logic gates.

Row Close Hardware: \[(\text{Number of channels}) \times (\text{Number of Ranks}) \times (\text{Number of Banks}) \times (\text{Number of time specific registers=4}) \times (64 \text{ bits/register}) = 2\text{Kbytes}\]

In addition, algorithm requires a 64-bit comparator per banks to compare the value of time from the last access to a threshold value.

Threshold value = close factor \times \text{average bank access interval}
It also requires a 64bitsX3bit multiplier, which can be optimized since the value of close factor, can be hardwired.

3.3 Opportunistic Refresh of Ranks

Based on how frequently a rank is being referenced, it can be observed to be operating in two phases. The busy phase in which rank is being accessed very frequently and idle or standby phase in which relative frequency of accesses to the rank is low or accesses are relatively separated in time. If a refresh command is issued to a rank while in busy phase its leads to additional latencies in read/write requests to that ranks as a minimum of 128 cycles (Refresh Cycle Time or tRFC) are required before an activation command can be issued after refresh. Such refreshes were determined to be resulting due to forced refresh of rank by memory controller.

Our scheduler employs a method, which rather than depending solely on forced refreshes, dynamically issues refresh to a rank, which has entered idle/standby phase. Our algorithm dynamically determines which rank is operating in idle phase and issues refresh to it. This effectively avoids refreshes from occurring in busy phase of rank.

For determining the operating phase of a rank, our algorithm monitors average access time and last access time to a rank, where average access time is dynamic average of time between two consecutive accesses to it. Both of these quantities are updated at every
access to the rank. If a significant amount of time has passed after last access to a rank in comparison to average access time, the rank is assumed to enter idle phase and a refresh can be safely issued. See fig 4 for reference.

The rank refresh logic reuses the per bank average access time and last access time registers for calculating average values for rank. For keeping the track of refresh deadline for a rank and number of refreshes that has been issued to it in last $8 \times t_{REFI}$ time. The algorithm uses refresh_issue_deadline and num_issued_refreshes registers provided by memory controller. Thus, refresh logic does not add any significant hardware. It only needs an adder and two 64-bit registers average_rank_access and last_rank_access.

Hence, the number of refresh logic storage bits required is 128 bits. Allowing refresh improves performance by increasing the availability of ranks towards future accesses.
Figure 4: Flow for Opportunistic Refresh of rank in DRAM
3.4 Heuristics for Switching between read and write modes

The decisions regarding when to switch between read and write modes plays an important role in determining the overall performance of the memory scheduling algorithm. There is a Write to Read time delay associated between the end of a write data burst and the start of a column-read command. The frequent switching between read and write mode can cause major performance drawbacks. In our algorithm, we use average time spent in read and write mode and mode factor to find the optimum time for switching between modes. For each channel, the average time spent in the read and the write mode is monitored. The scheduler makes sure that the time before switching from read mode to write mode is always greater than the average time spent in the read mode divided by the mode factor.

The implementation of switching heuristics based on average time spent in read burst requires following registers:

- Register to store number of switching from read to write mode,
- Register to store average time spent in one read mode burst and a configurable mode factor register.
The implementation of logic requires comparator, adder and few logic gates. The total amount of storage required to implement the switching algorithm is minimal. The total storage required is three registers each of 32 bits. Storage for switching modes: 96 bits.

3.5 Two-Level Watermark Switching Policy from read mode to write mode

In our algorithm, we use two levels of watermark to find the right time to switch from the read to write mode. Once the length of write queue crosses the first watermark value, the scheduler tries to issue all the row hits in the read queue and then switch to write mode when no row hit is found. However, when the write queue length crosses the second watermark value the scheduler immediately switches to the write mode.

Read Mode Issue Priority

1. Marked Requests : Row Hits
2. Marked Requests : High Ranked First
3. Unmarked Requests : Row Hits
4. Unused Row Closure
5. Unmarked Requests: FCFS
6. Refresh Command
The two level watermark switching technique can also be easily implemented. It requires two configurable registers to store the values of watermark (10 bits total) and a comparator to compare the value with the write queue length. A simple logic circuit is required which switches the scheduler from write to read mode immediately when the write queue length reaches the level2 watermark value. It delays the switching if the write queue length is more than level1 watermark but less than level2 watermark and a row-hit request is present in the read queue.

Total Storage required: 192+24+1536+1757+16,384+128+96+10 = 20,127 bits (2.45KB)

3.6 Detailed Explanation of Code

Here, I will try to present the pseudo code for better understanding of our DRAM memory scheduler.

**Code snippet for defining the Macro’s**

// write queue high water mark; begin draining writes if write queue exceeds this value
#define HI_WM 40

// end write queue drain once write queue has this many writes in it
#define LO_WM 32
// when switching to write drain mode, write at least this many times before switching
back to read mode

#define MIN_WRITES_ONCE_WRITING_HAS_BEGUN 1

// 1 means we are in write-drain mode for that channel
int drain_writes[MAX_NUM_CHANNELS];

// how many writes have been performed since beginning current write drain
int writes_done_this_drain[MAX_NUM_CHANNELS];

// flag saying that we're only draining the write queue because there are no reads to
schedule
int draining_writes_due_to_rq_empty[MAX_NUM_CHANNELS];

**Code Snippet for Defining Dynamic Allocation of Access Time**

// Dynamic allocation of access time
channels_t = (channel_stats_t*)malloc(NUM_CHANNELS*sizeof(channel_stats_t));
for(i=0; i<NUM_CHANNELS; i++) {
    channels_t[i].channel_stats = (bank_stats_t**)malloc(NUM_RANKS*sizeof(bank_stats_t*));
    for(j=0; j<NUM_RANKS; j++) {
channels_t[i].channel_stats[j] = (bank_stats_t*)malloc(NUM_BANKS*sizeof(bank_stats_t));
}

for(i=0; i<NUM_CHANNELS; i++) {
    for(j=0; j<NUM_RANKS; j++)
        for(k=0; k<NUM_BANKS; k++) {
            channels_t[i].channel_stats[j][k].last_access_time = -1;
            channels_t[i].channel_stats[j][k].access_interval = -1;
        }
}

Code Snippet for Write Drain if We are Above Water Level Mark

// begin write drain if we're above the high water mark
if((write_queue_length[channel] > HI_WM) && (!drain_writes[channel]))
{
    drain_writes[channel] = 1;
    writes_done_this_drain[channel] = 0;
}

Code Snippet for Beginning Write Drain if Read Queue Is Empty

```c
if((read_queue_length[channel] < 1) && (write_queue_length[channel] > 0) && (!drain_writes[channel]))
{

drain_writes[channel] = 1;

writes_done_this_drain[channel] = 0;

draining_writes_due_to_rq_empty[channel] = 1;
}
```

Code Snippet for Ending Write Drain if We are Below Low Water Level Mark

```c
if((drain_writes[channel]) && (write_queue_length[channel] <= LO_WM) && (!draining_writes_due_to_rq_empty[channel]))
{

drain_writes[channel] = 0;
}
```
Code Snippet for Ending Write Drain that Was Due to Read_Queue Emptiness

Only if at least One Write has Completed

if((drain_writes[channel]) && (read_queue_length[channel] > 0) &&
(dRAINING_WRITES_DUE_TO_RQ_EMPTY[channel]) && (writes_done_this_drain[channel] >
MIN_WRITES_ONCE_WRITING_HAS_BEGUN))
{

drain_writes[channel] = 0;

draining_writes_due_to_rq_empty[channel] = 0;
}

Code Snippet for the Situation if COL_WRITE_CMD is the Next Command, then
that Means the Appropriate Row Must Already be Open

if(wr_ptr->command_issuable && (wr_ptr->next_command == COL_WRITE_CMD))
{

writes_done_this_drain[channel]++;
issue_request_command(wr_ptr);

return;

}

Code Snippet for Reading FR

if((wr_ptr->command_issuable && (wr_ptr->next_command == COL_WRITE_CMD))
{
writes_done_this_drain[channel]++;

issue_request_command(wr_ptr);

return;

}

Code snippet for Reading FCFS

LL_FOREACH(read_queue_head[channel],rd_ptr)
{
if(rd_ptr->command_issuable) {

if(rd_ptr->next_command == PRE_CMD)

reset_time(channel, rd_ptr->dram_addr.rank, rd_ptr->dram_addr.bank);

issue_request_command(rd_ptr);

return;

}

}


\textbf{Code Snippet for Closing Row}

for(i=0;i<NUM_RANKS;i++) {

for(j=0;j<NUM_BANKS;j++) {

if((CLOSE_FACTOR*channels_t[channel].channel_stats[i][j].access_interval <

(CYCLE_VAL - channels_t[channel].channel_stats[i][j].last_access_time))

&&

(channels_t[channel].channel_stats[i][j].last_access_time != -1)

&&

(channels_t[channel].channel_stats[i][j].access_interval != -1)) {

if(is_precharge_allowed(channel,i,j)) {

reset_time(channel,i,j);

issue_precharge_command(channel,i,j);

return;
Chapter 4

PERFORMANCE CHARTS EVALUATION AND RESULTS

There are different workload benchmarks provided by USIMM [5]. For each workload a trace file is given as input. Different workloads have different amount of data and thread level parallelism and number of threads.

Below described table will show all the result after running all the trace files individually without shortening there length. This research tried running all the trace files under different system configuration, like for single memory channel and then for more aggressive memory channel example for 4 channels. All the trace files are mentioned under workload column.

To substantiate that our scheduler is better than other schedulers, This research have used FRFCFS and CPP scheduler for comparison. All the objectives of this research can be obtained by running all the workloads under three schedulers, namely FR-FCFS, CPP and finally our proposed scheduler. The matrices used are based on sum of execution times, maximum slowdown and energy delay product.
Table 2: Performance & Evaluation metrics

<table>
<thead>
<tr>
<th>Workload</th>
<th>Config</th>
<th>Sum of exec times (10 Meyc)</th>
<th>Max slowdown</th>
<th>EDP (J.s) (3 significant digits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>FCFS</td>
<td>Close</td>
<td>Proposed</td>
</tr>
<tr>
<td>MT-eanneal</td>
<td>4 chan</td>
<td>1</td>
<td>418</td>
<td>404</td>
</tr>
<tr>
<td>MT-eanneal</td>
<td>1 chan</td>
<td>1</td>
<td>179</td>
<td>167</td>
</tr>
<tr>
<td>bl-bl-fr-fr</td>
<td>4 chan</td>
<td>1</td>
<td>149</td>
<td>147</td>
</tr>
<tr>
<td>c1-c1</td>
<td>1 chan</td>
<td>1</td>
<td>83</td>
<td>83</td>
</tr>
<tr>
<td>c1-c1</td>
<td>4 chan</td>
<td>1</td>
<td>51</td>
<td>46</td>
</tr>
<tr>
<td>c1-c1-c2-c2</td>
<td>1 chan</td>
<td>1</td>
<td>242</td>
<td>236</td>
</tr>
<tr>
<td>c1-c1-c2-c2</td>
<td>4 chan</td>
<td>1</td>
<td>127</td>
<td>118</td>
</tr>
<tr>
<td>c2</td>
<td>1 chan</td>
<td>1</td>
<td>44</td>
<td>43</td>
</tr>
<tr>
<td>c2</td>
<td>4 chan</td>
<td>1</td>
<td>30</td>
<td>27</td>
</tr>
<tr>
<td>fa-fa-fe-fe</td>
<td>1 chan</td>
<td>1</td>
<td>228</td>
<td>224</td>
</tr>
<tr>
<td>fa-fa-fe-fe</td>
<td>4 chan</td>
<td>1</td>
<td>106</td>
<td>99</td>
</tr>
<tr>
<td>fl-fl-sw-sw-c2-c2-fe-fe</td>
<td>4 chan</td>
<td>1</td>
<td>295</td>
<td>279</td>
</tr>
<tr>
<td>fl-fl-sw-sw-c2-c2-fe-bl-bl-fr-fr-c1-c1-st-st</td>
<td>4 chan</td>
<td>1</td>
<td>651</td>
<td>620</td>
</tr>
<tr>
<td>fl-fl-sw-c2-c2</td>
<td>1 chan</td>
<td>1</td>
<td>249</td>
<td>244</td>
</tr>
<tr>
<td>fl-fl-sw-c2-c2</td>
<td>4 chan</td>
<td>1</td>
<td>130</td>
<td>121</td>
</tr>
<tr>
<td>st-st-st-st</td>
<td>1 chan</td>
<td>1</td>
<td>162</td>
<td>159</td>
</tr>
<tr>
<td>st-st-st-st</td>
<td>4 chan</td>
<td>1</td>
<td>86</td>
<td>81</td>
</tr>
<tr>
<td>Overall (Without Rounding)</td>
<td></td>
<td>3312</td>
<td>3173</td>
<td>3140</td>
</tr>
</tbody>
</table>

These are the numbers, which we got after running all the trace files on the USIMM simulator. The numbers generated clearly shows the overall performance of all the possible approaches used today.
After submission of the entire individual runs this research, concludes that our proposed scheduler is far better than the other schedulers. This can be clearly seen in the table that the overall number of our proposed scheduler is far more convincing in all the three performance matrices used, proving it to be better than FR-FCFS and CPP memory scheduling algorithm.

The simulation takes a long time to complete a whole trace file so to do some experiments you can truncate the trace file and can see the desired output. These numbers generate by running the entire trace files on the simulator.

The Input directory inside the simulator contains the system and DRAM chip configuration files that are read by USIMM. The input given to the simulator is a set of 13 different trace files for 10 different benchmarks.

The following ten benchmark traces are included in the distribution (13 files): [5]

1. **black**: A single-thread run from PARSEC’s blackscholes.
2. **face**: A single-thread run from PARSEC’s facesim.
3. **ferret**: A single-thread run from PARSEC’s ferret.
4. **fluid**: A single-thread run from PARSEC’s fluidanimate.
5. **freq**: A single-thread run from PARSEC’s freqmine.
6. **stream**: A single-thread run from PARSEC’s streamcluster.
7. **swapt**: A single-thread run from PARSEC’s swaptions.
8. **comm1**: A trace from a server-class transaction-processing workload.
9. **comm2**: A trace from a server-class transaction-processing workload.

10. **MT*-canneal**: A four-thread run from PARSEC’s canneal, organized in four files, MT0-canneal to MT3-canneal.

- Benchmarks black, face, ferret, freq, stream have about 500 million instructions. These instructions were selected from 5 billion instruction traces with a methodology similar to that of Simpoint. SimPoint is used to generate traces which are representative of the benchmarks. SimPoint uses Basic Block Vectors (BBVs) to recognize intervals of the execution, which can be used to replicate the behavior of the benchmark. It assigns weights to each interval, which can be applied to the results obtained from each interval.

In our model, each benchmark is simulated for 5 billion instructions with interval sizes of 5 million instructions. A number of metrics are collected from each interval, including number of LLC misses, number of reads, number of writes, number of floating-point, integer, and branch instructions. The collection of these metrics forms the BBV that is used with SimPoint. The BBVs are classified into 10 clusters using SimPoint. The number of intervals selected from each cluster depends on the weight of each cluster. The final trace of 500M instructions is the combination of 100 intervals, taken from the large trace.

Benchmarks fluid and swapt are also defined with the Simpoint-like methodology
described for the other PARSEC benchmarks. The only difference is that the traces
include 750 million instructions so they have execution times similar to the other
benchmarks.

- Benchmarks fluid and swapt are also defined with the Simpoint-like methodology
described for the other PARSEC benchmarks. The only difference is that the traces
include 750 million instructions so they have execution times similar to the other
benchmarks.

- Benchmarks comm1 and comm2 are roughly 500 million instruction windows
that are representative of commercial transaction-processing workloads.

- The 4-thread canneal traces represent the first 500 million instructions executed
by each thread once the region of interest is started. While all of the other traces were
collected with 512 KB private LLCs for each single-thread program, the 4-thread canneal
traces assumed a 2 MB shared LLC for the four cores. A write in a trace file represents
the dirty block evicted by a block that is being fetched by that thread.

- All 10 workloads are run with 4channel.cfg, and the first 8 are also run with
1channel.cfg. The numbers from these 18 simulations will be used to compute the
metrics that must be reported in the papers being submitted to the competition. The
runsim file in the usimm directory lists all 18 simulations.
The 10 workloads are:

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

- Fairness is being 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 final PFP metric will multiply the {average of maximum slowdowns across all experiments} and the {sum of execution times of all programs in those experiments}. For the PFP metric, only 14 of the 18 experiments will be used (the single-program comm2 workload and the multi-threaded canneal workload will not be used to evaluate fairness).
4.1 Percentage Performance Improvement over FCFS

This is the graph, which I got after comparing our proposed scheduler performance with the FCFS. It clearly shows the percentage improvement for all trace files when given input to the simulator. Here, mean performance improvement is 6.1 %

Graph.1: Percentage Performance Improvement over FCFS
4.2 Percentage Reduction in Energy Delay Product over FCFS

I compared difference between percentage of energy delay product between FCFS and our proposed scheduler. The graph clearly indicates that our proposed scheduler gives better result in terms of reducing energy delay product. Here, mean reduction in EDP is 12.21%
4.3 Percentage Reduction in Maximum Slowdown Over FCFS

While comparing the Maximum Slowdown of the running threads, our proposed scheduler once again proved to be much more efficient than the FCFS scheduler. We can think maximum slowdown as fairness among threads. Maximum slowdown of a multithreaded workload is the program that suffers the maximum slowdown relative to its single threaded execution here, mean reduction in slowdown is calculated to be 6.31%

Graph.3: Percentage Reduction in Maximum Slowdown over FCFS
4.4 Percentage Performance Improvement Over Close Page Policy

Comparing the results got by our proposed scheduler with the already existing one i.e Close Page Policy. The important thing here to note is the negative graph for C2 and fl-fl-sw-sw-c2-c2-fe-fe-bl-bl-fr-fr-c1-c1-st-st. The performance of these workload files is slightly better when ran on CPP scheduler rather than our scheduler.

C2 has very high bank-parallelism when run alone. Our Scheduler scheduling policy is to balance the requests from different threads in each bank, without any coordination among banks. As the other threads have busy access patterns in some banks, our scheduler prioritizes their requests over C2’s request in those banks during bursts (this is in part due to the idleness problem inherent in our scheduler design). Therefore, our scheduler destroys C2’s bank-parallelism: in some bank, C2’s requests are unhindered by requests from other threads, while in other banks; request from the thread is prioritized. C2’s requests in these banks are delayed, although they could have been serviced in parallel with its other requests. I found that C2’s performance improves a little better when ran on CPP scheduler. The same reason is valid for the other trace file as well.

Here the mean improvement is 1.51% over the CPP.
Comparing percentage reduction in maximum slowdown for our proposed scheduler with the close page policy gave us the following graph and reduced the overall mean reduction to 1.67%. The reason for a single negative graph is the highly random access pattern for the workload. Lack of correlation between accesses will make row buffer useless (as percentage hit will be low).
4.6 Percentage reduction in EDP over Close Page Policy

The final comparison is made for calculating the percentage reduction in EDP between our proposed scheduler and Close Page policy and the graph shows that the overall percentage reduction is less in our scheduler as compared to the already existing one. The mean reduction in EDP is 2.67%.
Graph.6: Percentage Reduction in EDP over Close Page Policy

4.7 Result

The proposed scheduler took 31400 Million cycles to complete all the sample trace. The baseline schedulers, FCFS and Close scheduler takes 33120 Million cycles and 31730 Million cycles to complete the same set of traces respectively. Performance improvement of 5.19% and 1.04% occurs as compared to FCFS and Close respectively. The total Energy Delay Product (EDP) improvement achieved by our scheduler was 10.00% and
0.96% over the FCFS and Close baseline scheduler respectively. Performance Fairness Product (PFP) improvement achieved by our scheduler was 10.55% and 2.34% over the FCFS and Close baseline scheduler respectively. Using an open row policy with row closure performs better with address mapping scheme in which cache lines are placed in the same row of a bank and thus have better locality.
5.1 Conclusion

Our research proposed a memory scheduling algorithm which uses PAR-BS as the basic model and combines it with some other techniques like predictive row closure, smart mode-switching techniques etc. We have measured the performance of our algorithm and have compared it with FCFS and Close-based scheduling algorithm. We have shown that significant performance improvement can be achieved by this algorithm over the baseline schedulers. The algorithm used is easily implementable in hardware and the hardware budget required to implement it is estimated to be very less than 68Kb as per the competition rules.

5.2 Future work

For now, this scheduler seems to be perfect but there can be a time when these numbers would not be just enough. This research tried 2-system configuration for this project to make it successful, one is for single channel and other one is for 4 channel. Similarly, we can do the same things for 8 channels and may be 16 channels to get more accurate numbers and better findings.
APPENDIX

Source Code

[a] Scheduler.c

#include <stdio.h>
#include "utlist.h"
#include "utils.h"
#include <stdlib.h>
#include <assert.h>
#include "params.h"
#include "scheduler.h"
#include "memory_controller.h"

#define CLOSE_FACTOR 25

extern long long int CYCLE_VAL;

extern int NUMCORES;

channel_stats_t* channels_t;

// write queue high water mark; begin draining writes if write queue exceeds this value
#define HI_WM 40

// end write queue drain once write queue has this many writes in it
#define LO_WM 32

// when switching to write drain mode, write at least this many times before switching
// back to read mode
#define MIN_WRITES_ONCE_WRITING_HAS_BEGUN 1

// 1 means we are in write-drain mode for that channel
int drain_writes[MAX_NUM_CHANNELS];

// how many writes have been performed since beginning current write drain
int writes_done_this_drain[MAX_NUM_CHANNELS];
// flag saying that we're only draining the write queue because there are no reads to
schedule
int draining_writes_due_to_rq_empty[MAX_NUM_CHANNELS];

void init_scheduler_vars() {
    int i, j, k;
    // Dynamic allocation of access time
    channels_t = (channel_stats_t*)malloc(NUM_CHANNELS*sizeof(channel_stats_t));

    for(i=0; i<NUM_CHANNELS; i++) {
        channels_t[i].channel_stats =
        (bank_stats_t**)malloc(NUM_RANKS*sizeof(bank_stats_t*));
        for(j=0; j<NUM_RANKS; j++){
            channels_t[i].channel_stats[j] =
            (bank_stats_t*)malloc(NUM_BANKS*sizeof(bank_stats_t));
        }    
        for(j=0; j<NUM_RANKS; j++){
            for(k=0; k<NUM_BANKS;
                k++) {
                channels_t[i].channel_stats[j][k].last_access_time = -1;
                channels_t[i].channel_stats[j][k].access_interval = -1;
            }    
        }    
    return;
}

void schedule(int channel)
{
    int i, j;
    request_t * rd_ptr = NULL;
    request_t * wr_ptr = NULL;
    // begin write drain if we're above the high water mark
    if((write_queue_length[channel] > HI_WM) && (!drain_writes[channel])){
        drain_writes[channel] = 1;
        writes_done_this_drain[channel] = 0;
    }

    // also begin write drain if read queue is empty

if((read_queue_length[channel] < 1) && (write_queue_length[channel] > 0) && (!drain_writes[channel]))
{
    drain_writes[channel] = 1;
    writes_done_this_drain[channel] = 0;
    draining_writes_due_to_rq_empty[channel] = 1;
}

// end write drain if we're below the low water mark
if((drain_writes[channel]) && (write_queue_length[channel] <= LO_WM) && (!draining_writes_due_to_rq_empty[channel]))
{
    drain_writes[channel] = 0;
}

// end write drain that was due to read_queue emptiness only if at least one write has completed
if((drain_writes[channel]) && (read_queue_length[channel] > 0) &&
(draining_writes_due_to_rq_empty[channel]) && (writes_done_this_drain[channel] > MIN_WRITES_ONCE_WRITING_HAS_BEGUN))
{
    drain_writes[channel] = 0;
    draining_writes_due_to_rq_empty[channel] = 0;
}

// make sure we don't try to drain writes if there aren't any
if(write_queue_length[channel] == 0)
{
    drain_writes[channel] = 0;
}

// drain from write queue now
if(drain_writes[channel])
{
    // prioritize open row hits
    LL_FOREACH(write_queue_head[channel], wr_ptr)
    {
        // if COL_WRITE_CMD is the next command, then that means the appropriate row must already be open
        (wr_ptr->command_issuable && (wr_ptr->next_command == COL_WRITE_CMD))
        {
            writes_done_this_drain[channel]++;
        }
issue_request_command(wr_ptr);
return;
}
// if no open rows, just issue any other available commands
LL_FOREACH(write_queue_head[channel], wr_ptr)
{
    if(wr_ptr->command_issuable)
    {
        issue_request_command(wr_ptr);
        return;
    }
}

// nothing issuable this cycle
    return;
}

// DO READ -FR
LL_FOREACH(read_queue_head[channel],rd_ptr) {
    if((rd_ptr->command_issuable) &&
       (rd_ptr->next_command == COL_READ_CMD)){
        update_time(channel, rd_ptr->dram_addr.rank, rd_ptr->dram_addr.bank);
        issue_request_command( rd_ptr);

        return;
    }
}

// FCFS READ
LL_FOREACH(read_queue_head[channel],rd_ptr){
    if(rd_ptr->command_issuable) {
        if(rd_ptr->next_command == PRE_CMD)
            reset_time(channel, rd_ptr->dram_addr.rank, rd_ptr->dram_addr.bank);
        issue_request_command( rd_ptr);
// Do ROW CLOSE
for(i=0;i<NUM_RANKS;i++) {
    for(j=0;j<NUM_BANKS;j++) {
        if((CLOSE_FACTOR*channels_t[channel].channel_stats[i][j].access_interval <
            (CYCLE_VAL - channels_t[channel].channel_stats[i][j].last_access_time))
            &&
            (channels_t[channel].channel_stats[i][j].last_access_time != -1)
            &&
            (channels_t[channel].channel_stats[i][j].access_interval != -1)){
            if(is_precharge_allowed(channel,i,j)) {
                reset_time(channel,i,j);
                issue_precharge_command(channel,i,j);
                return;
            }
        }
    }
}

void update_time(int channel, int rank, int bank) {
    if(channels_t[channel].channel_stats[rank][bank].last_access_time != -1)
        channels_t[channel].channel_stats[rank][bank].access_interval = CYCLE_VAL -
        channels_t[channel].channel_stats[rank][bank].last_access_time;
    channels_t[channel].channel_stats[rank][bank].last_access_time = CYCLE_VAL;
    return;
}

void reset_time(int channel, int rank, int bank) {
    channels_t[channel].channel_stats[rank][bank].access_interval = -1;
    channels_t[channel].channel_stats[rank][bank].last_access_time = -1;
    return;
}

void scheduler_stats() {
    return;
}
typedef struct bank_stats_t {
    long long int last_access_time;
    long long int access_interval;
} bank_stats_t;

typedef struct channel_stats_t {
    bank_stats_t** channel_stats;
} channel_stats_t;

void reset_time(int channel, int rank, int bank);
void update_time(int channel, int rank, int bank);

void init_scheduler_vars(); //called from main
void scheduler_stats(); //called from main
void schedule(int); // scheduler function called every cycle
BIBLIOGRAPHY

[1] Onur Mutlu and Thomas Moscibroda Parallelesim- Aware Batch Scheduling:
    Enhancing both performance and fairness of Shared DRAM systems. ISCA-2008


    Proceedings of MICRO 41 Proceedings of the 41st annual IEEE/ACM International
    Symposium on Microarchitecture.

[5] Niladrish Chatterjee, Rajeev Balasubramonian, Manjunath Shevgoor, Seth H.
    USIMM: the Utah SImulated Memory ModuleA Simulation Infrastructure for
    the JWAC Memory Scheduling Championship

    DRAM Page-mode Scheduling