Stall Time Fair Memory Access Scheduling for Chip Multiprocessors

Durga Kumari Pisipati, Naveen Gunasekaran, Aayushi Agarwal, Pavan Kumar Naik

Department of Electrical and Computer Engineering

University of Illinois at Chicago
Chicago, IL 60607, USA
{dpisip2, ngunas3, aagarw27, pnaik5}@uic.edu

Abstract – In a Chip Multiprocessor system (CMP), DRAM is the main resource which is shared among cores. Hence, there exists interference due to memory requests from different threads. Current Memory Access Scheduling techniques do not consider the interference among threads and only try to optimize the overall data throughput from the DRAM. Therefore, two threads running together on the same chip might experience extremely different performance of the memory system.

This paper proposes a memory access scheduler which provides fair quality of service to different threads especially when run together on the same chip sharing the DRAM memory system – Stall Time Fair Memory Access Scheduler. This scheduler mainly tries to consider the interference from other threads.

The results of STFM provide the best fairness, throughput and scalability. In this project, we make use of the STFM scheduler algorithm and analyze its results relative to the tradition FCFS algorithm.

Index Terms – DRAM, main memory, latency, DRAM controller, Thread Interference, DRAM controller policy, bank, row, USIMM.

I. INTRODUCTION

In a Chip Multiprocessor System (CMP), it is possible to run multi threads simultaneously on a single chip. Sharing parts of memory sub system in a CMP organization provides an advantage of power-efficiency, throughput and scalability when compared to that of a single-core system.
But this concept of shared hardware resources in a CMP system results in a problem of resource management. Due to this issue, some threads could be unfairly prioritized over others, and thus could be starved for long time and waiting to access shared resources.

The problems with unequal resource sharing in CMP systems are:
First, unequal resource sharing would cause discomfort to the end user who expects threads with higher priority to get higher share of the performance provided by the system.
It is favorable to the malicious programs which try to intentionally deny the service to the other threads. They try to devise the system by exploiting the unfairness in the resource sharing schemes. This results in the performance degradation and productivity loss in the system.
It becomes difficult to analyze and optimize the system performance.

FR-FCFS algorithm prioritizes memory requests based on the hit in the row buffers of DRAM banks. This memory access scheduling algorithm is thread unaware. Hence, when different threads that belong to the same chip run together: one of them might experience starvation while other is unfairly prioritized. Therefore, techniques to improve the Quality of service to threads sharing CMP resources are required.

In this paper, a new memory scheduling algorithm called the Stall – Time Fair Memory Scheduler (STFM) is proposed. The idea is to provide equality/ fairness to different threads sharing the DRAM Memory System.

Problem Statement:
Multiple threads share the DRAM controller
DRAM controllers are designed to maximize DRAM throughput
Current DRAM scheduling policies are thread unaware and unfair
Hence the Quality of Service (QOS) goal is to equalize the memory related priority each thread experiences due to interferences from other threads without hurting the overall performance.

Basic Idea: The scheduler estimates or predicts two values for each thread:
$T_{shared:}$ The Memory stall time which the thread experiences when running with others
T_{alone}: The Memory stall time the thread would have experienced when it runs alone.
Based on the estimates the memory slowdown S is calculated by the scheduler and defined as follows.

$$S = \frac{\text{T shared}}{\text{T alone}}$$

If the ratio of maximum slowdown and minimum slowdown in the system is higher than the threshold, then the scheduler prioritizes memory requests accordingly from the threads that are slowed down the most.

II. BACKGROUND

A. Memory Access Scheduling

In recent times, the performance of the memory system is highly dependent on the organization of the DRAMs, thereby allowing the memory to be exploited for efficient execution. The DRAMs as shown in Fig.1, is organized into a three-dimensional device with dimensions being banks, rows and columns. The structure of contemporary DRAMs has been modified such that the latency for accessing different rows within the same bank is much greater compared to the latency for accessing different banks within the same row. The property of non-uniform memory accesses to be supported by DRAM, enables memory devices to reorder memory operations. This is usually implemented in DRAMs through bank pre-charge, row activation, and column access, thereby reducing the average memory access latency and increase its bandwidth utilization.

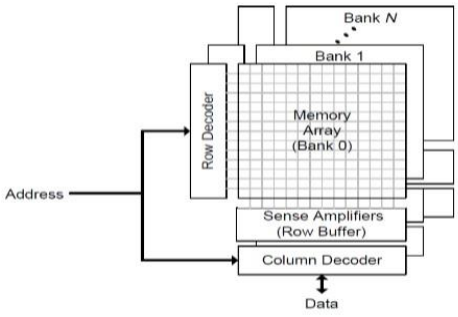

Figure 1. DRAM architecture

B. DRAM Organization and Access Latencies

A modern DRAM consists of one or more DIMMS known as Dual In line Memory Modules. A DIMM consists of multiple DRAM chips that are organized together which can be accessed in parallel. A single chip contains independent memory banks organized such that they also can be accessed in parallel. Each bank is again organized as a 2-D array which consists of multiple rows and columns.
Hence, a location in the DRAM can be accessed using a DRAM address which contains bank, row and column fields.
A thread's DRAM performance is dependent on its inherent Row-buffer locality and Bank parallelism.

A Memory Request to DRAM falls in one of the following categories:

Row hit: The request (read or write command) is accessing the row currently in a row buffer. It results in the lowest bank access latency (t_{CL}) as only a column access is required.

Row closed: When there is no row in the row buffer, an activated command needs to be issued first to open the row. It results in a bank latency of ($t_{RCD} + t_{CL}$) as both row and column access are required.

Row conflict: To access a row different from the one currently in the row buffer. This requires the contents of the row buffer to be written back to the memory. This results in highest bank latency ($t_{RP} + t_{RCD} + t_{CL}$)

C. *DRAM Controller Organization:*

DRAM controller is the communicator between the processors and DRAM. It is responsible to satisfy the memory requests and at the same time obey the timing and resource constraints of the DRAM banks, chips and address/ data buses.

DRAM Controller Consists of:

Request buffer: It holds the state associated with each memory request.

Read/ Write data buffers: Holds the data that is written to or read from DRAM.

DRAM access scheduler- It decides which DRAM command should be issued every clock cycle.

FR-FCFS is an algorithm designed to optimize the throughput obtained from the DRAM. Hence it prioritizes DRAM commands as:

Row-hit first: Unfairly prioritizes threads with high row buffer locality.

Oldest First: Unfairly prioritizes memory-intensive threads.

D. Simulator

To implement the STFM algorithm, Utah Simulated Memory Module (USIMM) simulator was used. It is a DRAM main memory system platform that allows implementation of memory access scheduling policies and track their performance and power. USIMM reads in application traces, models the progression of application instructions through the reorder buffers of a multi-core processor, and manages the memory controller read/write queues. Every memory cycle, USIMM checks various DRAM timing parameters to figure out the set of memory commands that can be issued in that cycle. It then hands control to a scheduler function that picks a command from this candidate set. An MSC contestant will only have to modify the scheduler function, i.e., restrict all the changes scheduler.c and scheduler.h. This clean interface makes it very easy to produce basic schedulers.

III. STALL-TIME FAIR MEMORY SCHEDULING:

Algorithm and Implementation

Although FCFS maximizes the throughput by exploiting parallelism it puts the other threads at a disadvantage that cannot reap row spatiality benefits and younger threads are serviced before increasing the stall time of older threads. However, in FR-FCFS plus the cap value could improve this. For instance, setting a value, let's say 3 would only quench the requests by three younger threads and then focus on older threads. But in our algorithm implementation we have come up with a fair and an efficient of equalizing these requests to evenly distribute the stall times which considers the memory ingrained properties of the said thread.

Circling back to the stall times – they incur as threads running simultaneously are in conflict. Instead of dividing DRAM bandwidth equally like previously explained schedulers here we adjust the stall time in all the threads. For this STFM revolves around calculating a threshold value: α *(alpha)*

Unfairness Factor - A Water Level which determines which path the DRAM should follow. Its bifurcated as:

1) Implement STFM if > α
2) Implement FR-FCFS if < α

For this the STFM revolves around the principal of calculating Time factor first for a thread when it runs on a shared DRAM with other threads: $\mathbf{T_{shared}}$ and calculating Time for a thread when it runs alone $\mathbf{T_{alone}}$. Taking a ratio of the two will give Slowdown **S** which prioritizes requests from the threads.

Finally taking a ration of Max (S) and Min (S) will conclude with the threshold factor – Unfairness quotient α. Let us dive into the detailed working of our algorithm and to gauge this Timing element of the threads.

A. SCHEDULER POLICY

At the start of our algorithm an assumption is made that all the threads should be treated important. Values calculated each DRAM cycle. Estimating $\mathbf{T_{Shared}}$ is quite simple – a counter is maintained by the processor which is incremented whenever there is an L2 cache miss which is further conveyed to the memory scheduler. Estimating $\mathbf{T_{alone}}$ is rather challenging.

To counter this, we introduce another timing parameter which is $\mathbf{T_{intereference}}$ which is at first initialized to zero. $\mathbf{T_{intereference}}$ is ascertained as the stall time our target thread C experiences when DRAM services other requests by different threads ahead of C's Request R. Hence the value 0 is updated of the thread with a ready request. Had the thread run by itself, $\mathbf{T_{intereference}}$ would be negated. Finally, $\mathbf{T_{alone}}$ is calculated as the difference between $\mathbf{T_{shared}}$ and $\mathbf{T_{interference.}}$ The stall time inflicted on threads with a ready request is again categorized as interference in:

DRAM bus:

DRAM data bus is kept occupied by the read/write command of target thread C for a period of $\mathbf{t_{bus}}$ cycles. The other threads with ready requests although scheduled cannot access this bus hence their $\mathbf{T_{interference}}$ increments by the specified $\mathbf{t_{bus.}}$ This $\mathbf{t_{bus}}$ value is approximated based on burst length plus

command type. For instance, a read or write command could have $\mathbf{t_{bus}}$ = BL/2.

DRAM bank:

a] Focus turns to memory-parallelism and here we talk in context of bank access made by request of target thread C. Other threads with read Rs must wait if accessing the same banks. A layman's assumption would be that a C or any other thread would just have one R. Here speculation is made that C' would have two requests being made to two different banks and the stall time would increase accordingly. Taking summation of the stall times of these two Rs - R1 and R2 would be cynical, ignoring the parallelism. Instead what the underlined policy is dividing the said R - one memory request by X number of banks. X being the BankWaitingParallelism of Thread C'.

Thus, STFM operates as increasing stall time of every other thread C' that has a ready command R' waiting to be scheduled to Bank B if a DRAM already has a command R from C to that same Bank B. Considering service Latency of R we state:

$$T^{old}_{Interference}(C') + \frac{Latency(R)}{\gamma * BankWaitingParallelism(C')}$$

ϒ Factor is a scaling factor that tells how much the bank waiting parallelism should be considered in calculating the $T_{interference}$ value. This could be aggressively sized by the scheduler if requests made are not in a parallel fashion in the future. There are several side-notes on this estimation and whether the delayed request is on the critical path of execution and how it affects the thread's performance. Such subjection is difficult to implement and for now ϒ is assumed to be ½ as in the paper.

b] The stall time of the target thread C could also increase because of its own subsequent requests R1 and R2. It might happen that the two requests are either being made to the same bank or there could be other requests served by other threads C' between the said R1 and R2. In such a case the last row accessed is saved in row buffer to provide a chance of row hit. Stall time would further increase in case of a row conflict. Had the thread C run alone, it would not have been penalized by the much higher delay of (trp + trcd + tcl). In this event the STFM operates using BankAccessParallelism which is the number of banks accessed by the n number of requests of the target thread C.

$$T^{new}_{Interference}(C) = T^{old}_{Interference}(C) + \frac{ExtraLatency}{BankAccessParallelism(C)}$$

Based on the three operations of STFM $\mathbf{T_{interference}}$ is calculated which then helps in estimation of $\mathbf{T_{alone}}$=$\mathbf{T_{shared}}$-$\mathbf{T_{interference}}$.

Slowdown indicator is now calculated as:

$$S = \frac{\text{T shared}}{\text{T alone}}$$

As said earlier the alpha factor is calculated by

$$Max\ (S) \Big/ Min\ (S)$$

If the level of unfairness is in acceptable range then the STFM regulates default FR-FCFS policy. If unfairness crosses the set threshold, STFM's own policy kicks in by:

1) Tmax first – Ready command issued by thread with largest Slowdown value estimated.

2) Column-First: Requests served from threads where the row is active to get more hits.

3) Oldest-first: Requests served from old threads over younger threads.

Consequently, it could be said that the STFM uses either FR-FCFS policy or modified FR-FCFS with priority given to the thread which is slowed down the most.

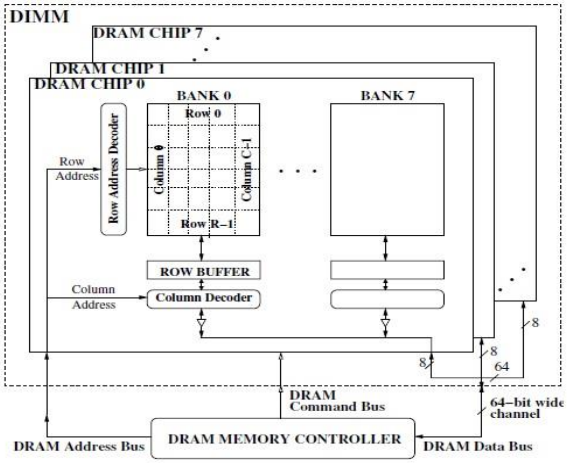

Figure 2: High Level Organization of DRAM system [1]

The figure 2 depicts the hardware representation of DRAM bank storing Last row accessed in Row Buffer to provide more row hits. Memory controller will decide upon the row active, close or conflict using this last row. [1]

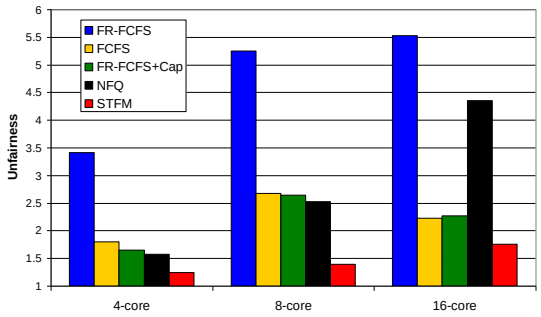

Figure 3: Plot of unfairness [1]

The figure 3 shows the unfairness for algorithms- FR-FCFS, FC-FCFS, FR-FCFS+Cap, NFQ and STFM. We can observe that systems with STFM provide the best optimization. [1]

C. OPTIMIZATION

We now try to tweak the algorithm implemented on certain areas to improve the three main factors of memory scheduler that is:
a. Total Execution Time
b. Energy-Delay Product
c. Processor Core Power.

Our STFM policy at the start states the assumption that all the threads are to be treated equally. But what if the system is not in favor of such enforcement. It may happen certain threads are less important than others but are prioritized first. Software could use another thread-based fairness mechanism.

In such an event we could use weights to determine the priority of threads in a queue. A less important thread could have less weight and vice versa. Hence a thread with higher weight would have more slowdown therefore be prioritized ahead. This weighted slowdown in terms of an equation is given as:

$$S = 1 + (S - 1) * \text{Weight}.$$

Equal weighted threads will be scaled and treated equally by the scheduler.

Secondly, the paper presents a debate on increasing the alpha value instead. Making the unfairness large would prevent any DRAM based hardware enforcement and the system could circumvent that. In this the optimization would be instead of a user setting the value in the software, the system could dynamically modify the alpha value at the runtime based on the workloads that are to be checked – be it either memory intensive, non-memory intensive or a mix of both. Although one should note, setting alpha value too large or too less could render the STFM ineffective and may not perform as desired.

Third Factor would be modifying Bank Waiting Parallelism Factor gamma. In the paper, it is simply assumed to be 0.5. Although extracting accurate information of this scaled parameter could be difficult, the nature of requests generated and which banks they are mapped to another equation could be derived rather than simply taking a summation or an average of banks accessed.

Modifying STFM's policy when it's in range of Unfairness threshold. As we saw that it implements FR-FCFS policy, a cap value could again be empirically estimated so as not to penalize any threads with less row hits. For instance, let's say that the cap value is set to 3, so 3 younger threads which are accessing the same row could be serviced first depending if that row is active i.e. latest row address stored in row buffer. As soon as the cap value is crossed, then the older requests are serviced regardless of a row hit or a row conflict.

All the parameters are calculated and set at every DRAM cycle that is the Unfairness quotient, threads with maximum and minimum slowdown and time of threads in contention. These parameters could probably be set earlier in a cycle to save on latency.

The weighted speedup provided is 1% where as Hmean speedup is 6%. We could look further how the throughput of the system could be maximized by preventing the starvation of non-intensive memory threads that tend to slow down too much.

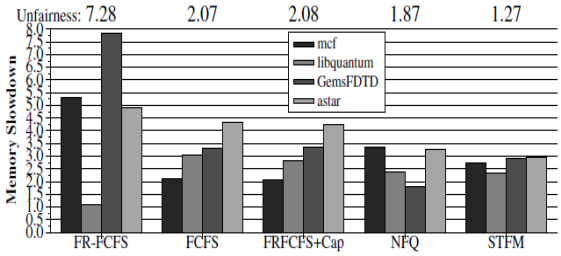

Figure 4: Memory Slowdown and Unfairness on a 4-core workload [1]

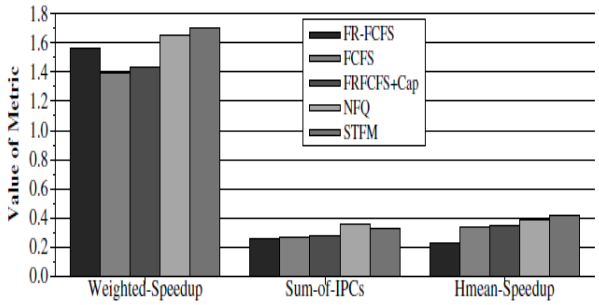

Figure 5: Memory Slowdown and Throughput on a 4-core workload [1]

IV. PERFORMANCE RESULT AND EVALUATION

The performance evaluation of the Stall time fair memory scheduling algorithm and the optimized algorithm has been compared to the traditional First Come First Serve (FCFS) Algorithm with the help of USIMM workloads. For each workload an input trace with different configuration is given to the USIMM. First, the default FCFS algorithm is run.

The screenshot of the simulation output for the FCFS algorithm is shown in Figure 5. The screenshots of the simulation output for STFM algorithm is shown in figure 6. The screenshots of the simulation output for the optimized STFM algorithm is shown in Figure 7.

```
-------- Channel 0 Stats----------
Total Reads Serviced :          4579961
Total Writes Serviced :         2627265
Average Read Latency :          183.54511
Average Read Queue Latency :    123.54511
Average Write Latency :         1248.72276
Average Write Queue Latency :   1184.72276
Read Page Hit Rate :            0.57492
Write Page Hit Rate :           -0.28504
-----------------------------------

#---------------------------Simulated Cycles Break-Up-----------------------------------------
Note:  1.(Read Cycles + Write Cycles + Read Other + Write Other) should add up to % cycles during which
         the channel is busy. This should be the same for all Ranks on a Channel
       2.(PRE_PDN_FAST + PRE_PDN_SLOW + ACT_PDN + ACT_STBY + PRE_STBY) should add up to 100%
       3.Power Down means Clock Enable, CKE = 0. In Standby mode, CKE = 1
#-------------------------------------------------------------------------------------------
Total Simulation Cycles                 442948245
-------------------------------------------------------------
```

Figure 5: FCFS Algorithm

```
-------- Channel 0 Stats----------
Total Reads Serviced :          4579961
Total Writes Serviced :         2627265
Average Read Latency :          176.36180
Average Read Queue Latency :    119.59166
Average Write Latency :         1223.7483
Average Write Queue Latency :   1162.2130
Read Page Hit Rate :            0.6380
Write Page Hit Rate :           -0.26711
-----------------------------------

#---------------------------Simulated Cycles Break-Up-----------------------------------------
Note:  1.(Read Cycles + Write Cycles + Read Other + Write Other) should add up to % cycles during which
         the channel is busy. This should be the same for all Ranks on a Channel
       2.(PRE_PDN_FAST + PRE_PDN_SLOW + ACT_PDN + ACT_STBY + PRE_STBY) should add up to 100%
       3.Power Down means Clock Enable, CKE = 0. In Standby mode, CKE = 1
#-------------------------------------------------------------------------------------------
Total Simulation Cycles                 430545694
-------------------------------------------------------------
```

Figure 6: STFM Algorithm

```
-------- Channel 0 Stats----------
Total Reads Serviced :          4579961
Total Writes Serviced :         2627265
Average Read Latency :          175.18180
Average Read Queue Latency :    118.97432
Average Write Latency :         1223.9478
Average Write Queue Latency :   1163.1098
Read Page Hit Rate :            0.6378
Write Page Hit Rate :           -0.26902
-----------------------------------

#---------------------------Simulated Cycles Break-Up-----------------------------------------
Note:  1.(Read Cycles + Write Cycles + Read Other + Write Other) should add up to % cycles during which
         the channel is busy. This should be the same for all Ranks on a Channel
       2.(PRE_PDN_FAST + PRE_PDN_SLOW + ACT_PDN + ACT_STBY + PRE_STBY) should add up to 100%
       3.Power Down means Clock Enable, CKE = 0. In Standby mode, CKE = 1
#-------------------------------------------------------------------------------------------
Total Simulation Cycles                 426240237
-------------------------------------------------------------
```

Figure 7: Optimized STFM Algorithm

Below table summarizes the results for FCFS, STFM and optimized STFM algorithm. The alpha value is changed from 1.2 to 0.97 in the optimized STFM algorithm. The table shows the variation of simulation cycles for different workloads.

Total Simulation Cycles			
Workload	**FCFS**	**STFM**	**Optimized STFM**
c2-1	442948245	430545694	426240237
c2-4	303069945	293977846	288392266
c1-c1-1	417482312	408589938	408589979
c1-c1-4	257303732	252157656	250316805
c1-c1-c2-c2-1	656987765	667302472	651014956
c1-c1-c2-c2-4	347019637	350489833	350419735
MTc-1	1050414637	1071317888	1050309595
MTc-4	450486649	454856369	454174084
fl-sw-c2-c2-1	655301377	648813893	648554367
fl-sw-c2-c2-4	343904097	343525802	341567704
fa-fa-fe-fe-1	576222849	581520356	595651300
fa-fa-fe-fe-4	1058850358	1058638587	1063402460

bl-bl-fr-fr-1	382876020	378664423	379118820
bl-bl-fr-fr-4	208459212	208021447	207998564
st-st-st-st-1	409103832	398671684	399508894
st-st-st-st-4	215059516	215038010	212866126
fl-fl-sw-sw-c2-c2-fe-fe-4	421758733	432117127	433197419
fl-fl-sw-sw-c2-c2-fe-fe-bl-bl-fr-fr- c1-c1-st-st-4	530213121	540764363	540656210

The graphical representation of the above results can be found in Figure 8.

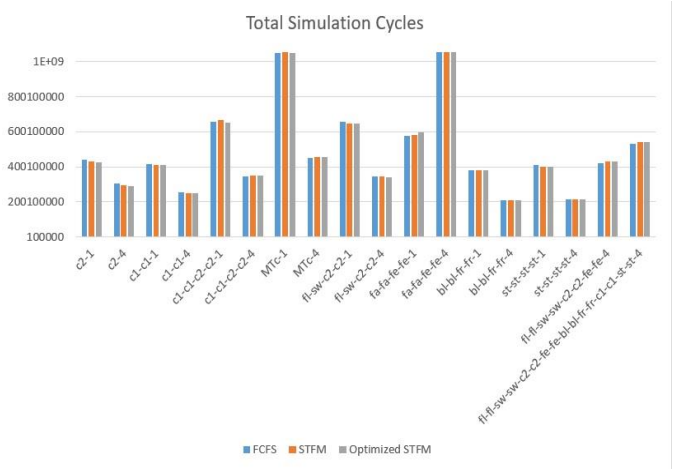

Figure 8: Total simulation cycles for different workloads

From the above results, we see that the number of simulation cycles for FCFS, STFM and the optimized STFM varies with intensity of the workload. For few workloads STFM shows good improvement while for others the traditional FCFS shows better performance results.

V. RELATED WORK

Nesbit et al. [9] addresses fairness issues at the DRAM controller level. Natarajan et al. [10] examine the effect of different memory controller policies on the performance of multiprocessor server systems. McKee et al. [11] proposed access reordering to optimize bandwidth in streamed accesses. These techniques primarily optimize throughput for single-threaded systems; they do not try to provide fairness to accesses from different threads. Providing fair access to threads sharing CMP caches has recently received increasing attention.

VI. CONCLUSION

In this project, we implemented a memory scheduling algorithm "STFM: Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors". The main theme of the paper is to implement fairness when equal priority threads are executed together. We implemented the STFM scheduler code. We also optimized the algorithm to improve the total execution time by changing the alpha value. This varying alpha value changes the amounts of fairness that is required when running for different workloads. The result shows that the optimized algorithm performs better for most of the workloads as there is considerable reduction in the total execution time. However, the optimized code does not perform better for certain workloads because the alpha value is already considered to be optimal.

VII. FUTURE WORK

The interaction of STFM with other fairness mechanisms at the shared caches is one important ground for future work. It is also observed that changing the alpha values were beneficial for certain workloads while it was not beneficial for certain others. Thus, a dynamic way could be implemented which would decide the alpha value based on the workload.

VIII.REFERENCES

[1] Onur Mutlu, Thomas Moscibroda. Stall-Time Fair Memory Access Scheduling for Chip Multiprocessors

[2] S. Bhansali et al. Framework for instruction level tracing and analysis of programs. In VEE 2006.

[3] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture optimizations for exploiting memory – level parallelism IN ISCA-31, 2004

[4] V. Cuppu, B. Jacob, B.T. Davis, and T. Mudge. A performance comparison of contemporary DRAM architectures. In *ISCA-26*, 1999.

[5] B. T. Davis. *Modern DRAM Architectures*. PhD thesis, University of Michigan, 2000.

[6] A. Fedorova, M. Seltzer, and M. D. Smith. Cache-fair thread scheduling for multi- core processors. Technical Report TR-17-06, Harvard University, Oct. 2006.

[7] J. M. Frailong, W. Jalby, and J. Lenfant. XOR-Schemes: A flexible data organization in parallel memories. In *ICPP*, 1985.

[8] R. Gabor, S. Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In *MICRO-3*

[9] K. J. Nesbit, N. Aggarwal, J. Laudon, and J. E. Smith. Fair queuing memory systems. In MICRO-39, 2006.

[10] C. Natarajan et al. A study of performance impact of memory controller features in multi-processor server environment. In WMPI, 2004.

[11] S. A. McKee et al. Dynamic access ordering for streamed computations. *IEEE Transactions on Computers*, 49(11):1255–1271, Nov. 2000

[12] K. Luo, J. Gummaraju, and M. Franklin. Balancing throughput and fairness in SMT processors. In ISPASS, 2001.

[13] R. Gabor, S. Weiss, and A. Mendelson. Fairness and throughput in switch on event multithreading. In MICRO-39, 2006.

Appendix

scheduler.c

```
#include <stdio.h>
#include "utlist.h"
#include "utils.h"

#include "memory_controller.h"

extern long long int CYCLE_VAL;

extern long long unsigned int
bank_access_parallelism;
long long unsigned int
bank_waiting_parallelism;


double alpha_value = 1.2;
double slowness[500];

double slowness_max;
int slowness_max_loc;
double slowness_min;
double slowness_max_min_ratio;

long long int t_alone[500];
long long int t_interference[500];
long long int extra_latency = 0;
int rank;
int bank;

void init_scheduler_vars()
{
// initialize all scheduler variables
here
return;
}

// 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 20

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

/* Each cycle it is possible to issue
a valid command from the read or
write queues
  OR
  a valid precharge command to
any bank
(issue_precharge_command())
  OR
  a valid precharge_all bank
command to a rank
(issue_all_bank_precharge_comma
nd())
  OR
  a power_down command
(issue_powerdown_command()),
programmed either for fast or slow
exit mode
  OR
  a refresh command
(issue_refresh_command())
  OR
  a power_up command
(issue_powerup_command())
  OR
  an activate to a specific row
(issue_activate_command()).

  If a COL-RD or COL-WR is
picked for issue, the scheduler also
has the
  option to issue an auto-precharge
in this cycle
(issue_autoprecharge()).

  Before issuing a command it is
important to check if it is issuable.
For the RD/WR queue resident
commands, checking the
"command_issuable" flag is
necessary. To check if the other
commands (mentioned above) can
be issued, it is important to check
one of the following functions:
is_precharge_allowed,
is_all_bank_precharge_allowed,
is_powerdown_fast_allowed,
is_powerdown_slow_allowed,
is_powerup_allowed,
is_refresh_allowed,
is_autoprecharge_allowed,
is_activate_allowed.
  */
```

```
void schedule(int channel, long
long int *t_shared)
{
request_t * rd_ptr = NULL;
request_t * wr_ptr = NULL;

if ((drain_writes[channel]) &&
(write_queue_length[channel] !=
0))
{
wr_ptr =
write_queue_head[channel];
bank_access_parallelism++;
}

if ((!drain_writes[channel]) &&
(read_queue_length[channel] != 0))
{
rd_ptr =
read_queue_head[channel];
bank_access_parallelism++;
}

if ((drain_writes[channel]) &&
(write_queue_length[channel] !=
0))
{
if (dram_state[channel][wr_ptr-
>dram_addr.rank][wr_ptr-
>dram_addr.bank].active_row !=
wr_ptr->dram_addr.row) // Check
if correct
                    {
extra_latency = 22; // tRCD + tRP
at 800 MHz
t_interference[0] =
t_interference[0] + (long long
int)(extra_latency/bank_access_par
allelism);
                    }
}

if ((!drain_writes[channel]) &&
(read_queue_length[channel] != 0))
{
if (dram_state[channel][rd_ptr-
>dram_addr.rank][rd_ptr-
>dram_addr.bank].active_row !=
rd_ptr->dram_addr.row) // Check if
correct
{
extra_latency = 22; // tRCD + tRP
at 800 MHz
t_interference[0] =
t_interference[0] + (long long
int)(extra_latency/bank_access_par
allelism);
}
}

int count = 0;
if ((drain_writes[channel]) &&
(write_queue_length[channel] !=
0))
{
LL_FOREACH(write_queue_head[
channel], wr_ptr)
{
if (wr_ptr->command_issuable)
{
count++;
}
}
count--;
bank_waiting_parallelism = count;
}

if ((!drain_writes[channel]) &&
(read_queue_length[channel] != 0))
{
LL_FOREACH(read_queue_head[
channel], rd_ptr)
{
if (rd_ptr->command_issuable)
{
count++;
}
}
count--;
bank_waiting_parallelism = count;
}

int latency;
if ((drain_writes[channel]) &&
(write_queue_length[channel] !=
0))
{
wr_ptr =
write_queue_head[channel];
latency = wr_ptr->completion_time
- wr_ptr->arrival_time; //needs
change
}

if ((!drain_writes[channel]) &&
(read_queue_length[channel] != 0))
{
```

```
rd_ptr =
read_queue_head[channel];
latency = rd_ptr->completion_time
- rd_ptr->arrival_time;
}

count = 0;
if ((drain_writes[channel]) &&
(write_queue_length[channel] !=
0))
{
LL_FOREACH(write_queue_head[
channel], wr_ptr)
{
count++;
if (wr_ptr->command_issuable)
{
t_interference[count] =
t_interference[count] +
(latency/(0.5 *
bank_waiting_parallelism));
}
}
}

count = 0;
if ((!drain_writes[channel]) &&
(read_queue_length[channel] != 0))
{
LL_FOREACH(read_queue_head[
channel], rd_ptr)
{
count++;
if (rd_ptr->command_issuable)
{
t_interference[count] =
t_interference[count] +
(latency/(0.5 *
bank_waiting_parallelism));
}
}
}

count = 0;
if ((drain_writes[channel]) &&
(write_queue_length[channel] !=
0))
{
LL_FOREACH(write_queue_head[
channel], wr_ptr)
{
count++;
if (wr_ptr->command_issuable)
{
t_alone[count] = t_shared[count] -
t_interference[count];
slowness[count] =
t_shared[count]/t_alone[count];
}
}
}

if ((!drain_writes[channel]) &&
(read_queue_length[channel] != 0))
{
LL_FOREACH(read_queue_head[
channel], rd_ptr)
{
count++;
if (rd_ptr->command_issuable)
{
t_alone[count] = t_shared[count] -
t_interference[count];
slowness[count] =
t_shared[count]/t_alone[count];
}
}
}

slowness_max = slowness[0];
for (int i = 1; i<500 ; i++)
{
if (slowness[i] > slowness_max)
{
slowness_max = slowness[i];
slowness_max_loc = i;
}
}

slowness_min = slowness[0];
for (int i = 1; i< 500 ; i++)
{
if (slowness[i] < slowness_min)
{
slowness_min = slowness[i];
}
}

slowness_max_min_ratio =
(slowness_max/slowness_min);

if (slowness_max_min_ratio <
alpha_value)
{
// if in write drain mode, keep
draining writes until the
// write queue occupancy drops to
LO_WM
```

```
if (drain_writes[channel] &&
(write_queue_length[channel] >
LO_WM)) {
drain_writes[channel] = 1; // Keep
draining.
}
else {
drain_writes[channel] = 0; // No
need to drain.
}

// initiate write drain if either the
write queue occupancy
// has reached the HI_WM , OR, if
there are no pending read
// requests
if(write_queue_length[channel] >
HI_WM)
{
drain_writes[channel] = 1;
}
else
{
if (!read_queue_length[channel])
drain_writes[channel] = 1;
}


// If in write drain mode, look
through all the write queue
// elements (already arranged in the
order of arrival), and
// issue the command for the first
request that is ready
if(drain_writes[channel])
{
LL_FOREACH(write_queue_head[
channel], wr_ptr)
{
if(wr_ptr->command_issuable)
{
issue_request_command(wr_ptr);
break;
}
}
return;
}


// Draining Reads
// look through the queue and find
the first request whose
// command can be issued in this
cycle and issue it
// Simple FCFS
if(!drain_writes[channel])
{
LL_FOREACH(read_queue_head[
channel],rd_ptr)
{
if(rd_ptr->command_issuable)
{
issue_request_command(rd_ptr);
break;
}
}
return;
}
}
else
{
if(drain_writes[channel])
{
LL_FOREACH(write_queue_head[
channel], wr_ptr)
{
if ((wr_ptr + slowness_max_loc)-
>command_issuable)
{
issue_request_command(wr_ptr +
slowness_max_loc);
break;
}
}


return;
}

if(!drain_writes[channel])
{
LL_FOREACH(read_queue_head[
channel], rd_ptr)
{
if ((rd_ptr + slowness_max_loc)-
>command_issuable)
{
issue_request_command(rd_ptr +
slowness_max_loc);
break;
}
}

return;
}
}
}
```

```
void scheduler_stats()
{
  /* Nothing to print for now. */
}
```

scheduler.h

```
#ifndef __SCHEDULER_H__
#define __SCHEDULER_H__

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

#endif //__SCHEDULER_H__
```