TIME-BASED FAIRNESS-AWARE MEMORY SCHEDULING FOR MULTICORE PROCESSORS

By

Amr Saleh AboBakr Khalil Elhelw

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

GIZA, EGYPT

2015

TIME-BASED FAIRNESS-AWARE MEMORY SCHEDULING FOR MULTICORE PROCESSORS

By

Amr Saleh AboBakr Khalil Elhelw

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Electronics and Communications Engineering

Under the Supervision of

Assoc. Prof. Hossam A. H. Fahmy	Assist. Prof. Ali A. El-Moursy
Electronics and Communications	Computer and Systems
Faculty of Engineering, Cairo University	Electronics Research Institute

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

GIZA, EGYPT

2015

TIME-BASED FAIRNESS-AWARE MEMORY SCHEDULING FOR MULTICORE PROCESSORS

By

Amr Saleh AboBakr Khalil Elhelw

A Thesis Submitted to the
Faculty of Engineering at Cairo University
in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE
in

Electronics and Communications Engineering

Approved by the

Examining Committee

____

Associate Prof. Hossam A. H. Fahmy, Thesis advisor

____

Prof. Amin Mohamed Nassar, Internal member

____

Prof. Elsayed Mostafa Saad, External member

(Professor at Faculty of Engineering, Helwan University)

FACULTY OF ENGINEERING, CAIRO UNIVERSITY

GIZA, EGYPT

2015

Engineer: Amr Saleh Abobakr Khalil Elhelw

Date of Birth: 22/6/1986

Nationality: Egyptian

E-mail: amrkhalil4@hotmail.com

Phone: 01111372223

Address: Mohamed Elsharawy street, behind Mena Palace Hotel, Haram, Giza

Registration Date: 1/10/2010

Awarding:

Degree: Master of Science

Department: Electronics and Communications Engineering

Supervisors:

Associate Prof. Hossam A. H. Fahmy

Dr. Ali A. El-Moursy (Researcher at Electronics Research Institute)

Examiners:

Associate Prof. Hossam A. H. Fahmy (Thesis advisor)

Prof. Amin Mohamed Nassar (Internal examiner)

Prof. Elsayed Mostafa Saad (External examiner)

Title of Thesis:

Time-Based Fairness-Aware Memory Scheduling for Multi-core Processors

Keywords:

Multi-core; Memory Controller; Shared Resources; Memory Interference

Summary:

In the modern chip-multiprocessor system, concurrently executing applications/threads shares common resource such as main memory. Memory scheduling algorithms are developed to resolve memory contention between competing applications/threads so that throughput is high and fairness of the overall multi-core systems is guaranteed. Time-based Least Memory Intensive (TB-LMI) scheduling algorithm is a new memory scheduling algorithm introduced to improve multi-core processor’s throughput and fairness.

Acknowledgements

First of all I must thank ALLAH for his great mercy supporting me all the way till the end. If it weren’t for his help, I wouldn’t have reached this point.

I would like to thank my advisors, Dr. Ali El-Moursy, and Prof. Hossam A. H. Fahmy for giving me the opportunity to work in a fruitful research environment and for their continuous guidance and support, as well as for their successful discussions and encouragements.

Most importantly, I would like to thank my parents and my brother for their continuous encouragement, and my colleges in Egyptian Financial Supervisory Authority (EFSA) for their support.

To my parents

Table of Contents

List of Tables

List of Figures

List of Publications

Amr Elhelw, Ali A. El-Moursy, and Hossam A. H. Fahmy. Time-based least memory intensive scheduling. *In The 8th IEEE International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC-14), Aizu-Wakamatsu, Japan,* September 2014.

List of Abbreviations

ATLAS	Adaptive per-Thread Least-Attained-Service
BW	Bandwidth
CMP	Chip-level Multiprocessing
CPU	Central Processing Unit
DRAM	Dymanic Random Access Memory
FCFS	First Come First Serve
FIQMR	Fair Issue Queue Most Related
FLRMR	Fair Least Request Most Related
FR	First Ready
FR-FCFS	First-Ready First Come First Serve
FR-LREQ	First-Ready Least REQuest
FSM	Finite State Machine
Id	Identification number
ILP	Instruction Level Parallelism
IPC	Instruction Per Cycle
IQ-based	Issue Queue based
LAS	Least-Attained-Service
LREQ	Least Request
ME	Memory Efficiency
ME-LREQ	Memory Efficiency with Least REQuest
Modified-ROB_PF	Modified Reorder buffer Prioritization Factor
MPKI	Misses Per Kilo Instructions
MSHR	Miss Status Holding Register
MSU	Memory Scheduling Unit
OS	Operating System
PAR-BS	Parallelism Aware Batch Scheduling
RAM	Random Access Memory
ROB-based	Reorder Buffer based
RR	Round-Robin
SCHED	Stall Time Fair Memory
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multithreading
SQ	Schedule Quantum
TB-LMI	Time-Based Least Memory Intensive
TCM	Thread Cluster Memory
TLP	Thread Level Parallelism
TMA	Total Memory Access
TMAPB	Thread Memory Access Per Bank
TPSR	Thread Priority Storage Register

Abstract

In the modern chip-multiprocessor system, concurrently executing applications/threads shares common resource such as main memory. Memory scheduling algorithms are developed to resolve memory contention between competing applications so that throughput is high and fairness of the overall multi-core systems is guaranteed. This emphasizes the importance of the memory access scheduling to efficiently utilize memory bandwidth. Although memory access scheduling techniques have been recently proposed for performance improvement, most of them have overlooked the fairness among the running applications.

In this thesis, we present Time-based Least Memory Intensive (TB-LMI) scheduling that address both fairness and system performance. The main idea of TB-LMI is to prioritize threads according to their memory contentions every pre-defined period of cycles to improve system throughput and to guarantee fairness. We evaluate TB-LMI on a variety of multi-programmed workloads with different queue sizes of memory controllers and compare its performance to six previously proposed scheduling algorithms. TB-LMI achieves the best system throughput and fairness. Previously proposed algorithms were First-Ready First Come First Serve (FR-FCFS) scheduling, First-Ready Fair Least-Request Most Related (FR-FLRMR) scheduling, First-Ready Fair Issue-Queue based Most Related (FR-FIQMR) scheduling, First-Ready Modified Reorder Buffer based (FR-Modified_ROB-based) scheduling, First-Ready Least REQuest (FR-LREQ) scheduling and Thread Cluster Memory (TCM) scheduling. TCM, FR-LREQ, and FR-FLRMR showed competitive results against the new scheduling TB-LMI. On 8-core system, TB-LMI improves system throughput and fairness on average by 4.22% and 11.7% respectively compared to TCM which was the previous work that provides the best system throughput and fairness.

Chapter 1. Introduction

A Central Processing Unit (CPU) is typically referred to as a processor. A processor contains memory caches, decoders, and execution units. Memory caches may be separated to a cache for instructions (Instruction cache) and another one for data (data cache) or unified caches where one cache for both instruction and data. Execution units such as Arithmetic Logic Unit (ALU) are used in performing arithmetic or logical operations. In order to increase processor's throughput, recent processors are tending now days towards parallel architectures. In early days, Operating Systems (OS) were developed to support multiprogramming. Multiprogramming is a kind of parallel processing in which several applications can run at the same time. In case of single CPU, OS executes part of one program, then part of another. All programs are appeared to be executed at the same time. Recent processors contain more than one CPU (core), allowing different applications to execute in parallel such as Simultaneous Multi Processing (SMP), Chip-Level Multi Processing (CMP), and Simultaneous Multi Threading (SMT) (described later). In order to get the highest benefit from recent processors, running applications must have a lot of routines that can run simultaneously. As an example a user may use the desktop to surf the web, watch a video and play a flash game at the same time. In general, hardware these days is trending toward highly parallel architectures.

This move resulted in increasing the number of threads that execute in parallel. All these threads are competing for shared resources. One of these most important resources is system main memory. While execution, each thread sends requests to the main memory to serve the cache misses. This introduces the need of memory access schedulers to decide which requests should be served first to either improve throughput or fairness or both.

1.1 Simultaneous Multithreading

In processor design, there are two ways to increase the on-chip parallelism: the first is superscalar technique which tries to make full use of Instruction Level Parallelism (ILP); the second is Thread Level Parallelism (TLP). Superscalar means that a processor tries to execute multiple instructions at the same time within a single processor core. TLP means that a processor tries to execute instructions from multiple applications/threads at the same time.

Some core components are duplicated in SMT. For example; an SMT core might have duplicate resources of thread scheduling, so that the core looks like two separate processors although it has a single execution unit. One of the SMT processors implementations is Hyper-threading processors which were introduced by Intel [1]. A processor with Hyper-Threading Technology consists of two logical processors per core. Each logical processor has its process state (logical registers and program counter). Each logical processor acts as a single processor where it can be individually interrupted, stalled, or directed to execute a specific application independently from the other logical processor. As shown in Figure 1 Hyper-threading processors, logical processors share the instruction cache, fetch queue, decoder, and L2 cache but they have different execution unit.

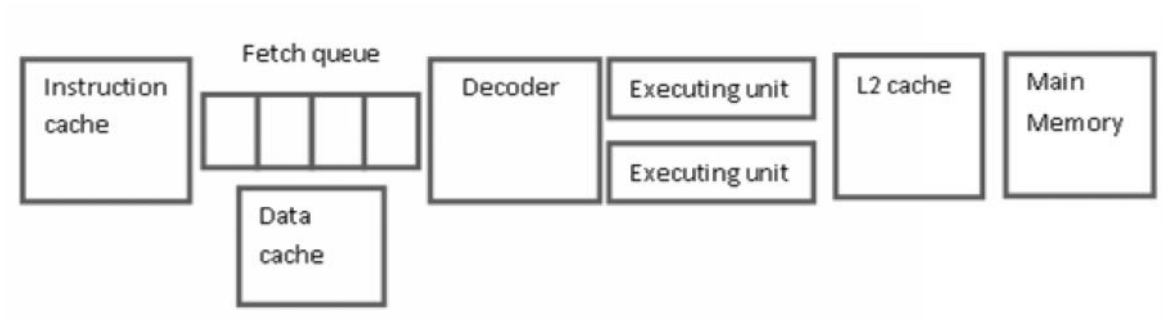


Figure 1.1: SMT architecture

1.2 Symmetric Multiprocessing

SMP stands for a symmetric multiprocessor system in hardware and software architecture. SMP consists of two or more identical processors that share common resources such as main memory interrupt system, and I/O devices. Each processor has its own Instruction cache, data cache, fetch unit, decoder, execution unit, and L2 cache. These identical processors are implemented on different chip. Each processor has its own chip and they share the common resources through a bus or a crossbar. Figure 1.2 shows SMP architecture. One of the first SMP processors is that what was introduced by IBM s/360 series in 1960s [2]. One of the main advantages of SMP processors are that if one processor fails the other can handle system requests. Also, if one application is multithreaded it can use more than one processor. Multithreaded applications may arise data inconsistency problem where data required may be obsolete (wrong). It is possible to have multiple copies of any instruction from a running application. One copy is in the main memory and a copy in each cache. Cache coherence guarantee that the changes in any shared data is updated to all caches and main memory. SMP disadvantages are its waste in power, energy, and area.

1.3 Multi-core processors

Multi-core processors are kind of processor that contains more than one core in one chip. These cores have their own instruction cache, Fetch stage, Decode stage, data cache, and execute stage but they share the same main memory and L2 cache. Figure 1.3 shows the architecture of multi-core processors. Multi-core processors are classified into homogenous multi-core which includes identical cores and heterogeneous multi-core that have non-identical cores [3].

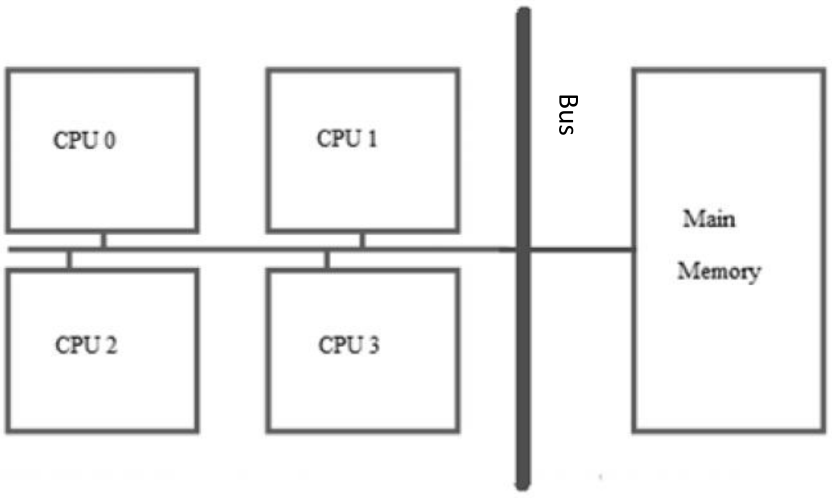


Figure 1.2: SMP architecture

In early 2000s, multi-core processors were developed and introduced by Intel, AMD and others. Multi-core processors may have two cores (for example Intel Core Duo and AMD Phenom II X2), four cores (for example Intel's i5 and i7), six cores (for example AMD Phenom II X6), eight cores (for example Intel Xeon E7-2820), ten cores (for example Intel XeonE7-2850), and more [4].

One of the advantages of multi-core processors is its die size. Multi-core processors require less area than SMP. In addition, multi-core processors allow higher performance at lower energy. It was discovered that the multi-core processor chip is more energy efficient than single large monolithic SMP [5]. In order to maximize the utilization of the computing resources provided by multi-core processors adjustments are required for both the OS support and to existing application software. As an example the Valve cooperation's (American video game development and digital distribution) source engine [6] offers multi-core support and Crytek [7] (German video game Company) has developed multi-core support for the game engine (software framework) CryEngine 2 to power their game.

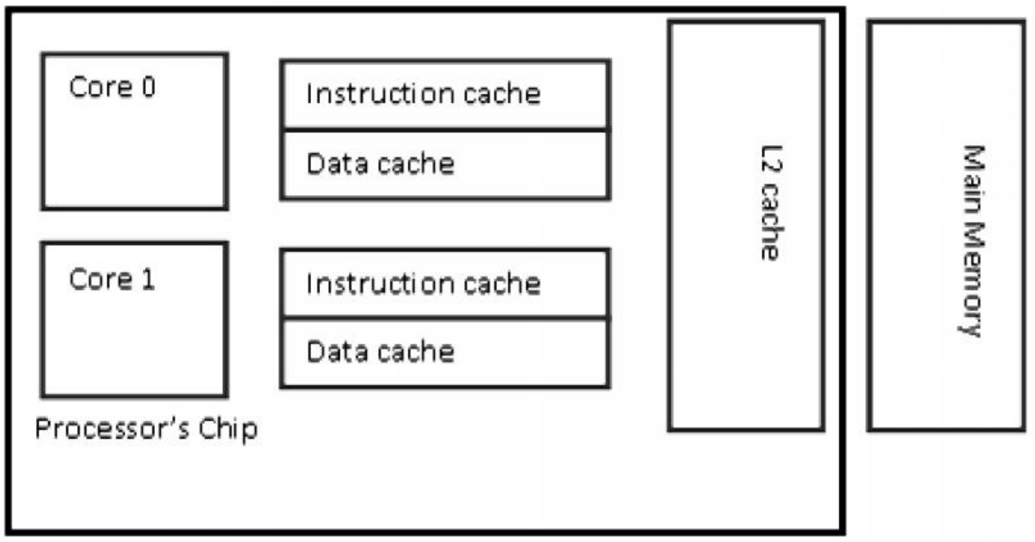


Figure 1.3: Multi-core processors

1.4 SMT/Multi-core throughput enhancement techniques

Multi-core, SMT, and SMP were new processor architecture developed to increase system's throughput. Also, different ideas were proposed in order to increase system's throughput.

- **Increasing Cache sizes**

Increasing the size of L1 and L2 caches is considered one of the simplest solutions to increases SMT, multi-core, and even single threaded processors throughput. This solution showed difficulties in implementation. First, increasing sizes of L1 or L2 or both leads to an increase in the chip area, energy and power consumption. Second, increasing size does not always guarantee increasing system's throughput. At a certain threshold running application are satisfied with the current cache resources available. Then any increase in cache sizes is considered a resource waste. Experimentally, increasing cache sizes does not increase system's throughput as expected [8].

- **Enabling high bandwidth**

One of the suggested proposals to improve system's throughput is to increase main memory/cache bandwidth. This can be done through several methods:

- **True multi-porting**: where the number of ports of caches and main memory is increased so that multiple accesses could be performed simultaneously. As an example L1 cache may has 2 read ports and 1 write port. This means that L1 cache can perform three requests simultaneously (2 load requests and 1 store request). Hardware, area, inefficient usage of energy and increase in power consumptions limits the increase in using this solution.

- **Virtual multi-porting:** which was introduced in IBM Power 2 and DEC 21264 [9] where access ports in main memory and caches are time shared. Limitations on maximum clock period and its scalability are disadvantages of this solution. It is not scalable beyond 2 ports

- **Multiple cache copies:** where a cache have multiple copies and have the same data stored. Multiple cache copies were used in DEC Alpha 21164 [10] and IBM Power 4. Load operations can be performed independently from caches while store operations are performed on all the caches as shown in Figure 1.4. Area increase and scalability are considered the main disadvantage of this design.

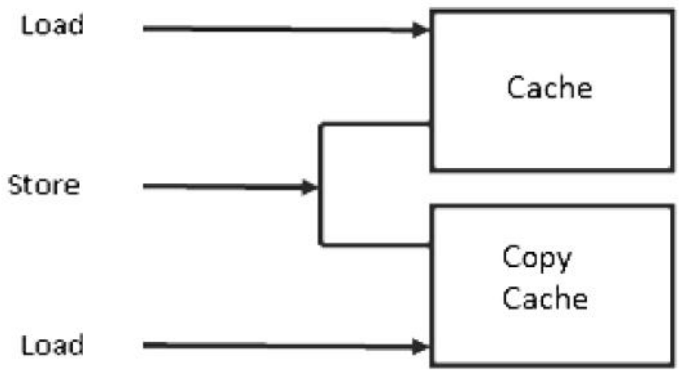


Figure 1.4: Multiple cache copies

- **Miss Status Holding Register (MSHR)**: MSHR is implemented in cache memories in order to track information about all the in-progress misses [11]. In systems that support non-blocking loads (in which the thread can continue running while it faced one or more load misses), there is a need for MSHR. After a load miss occur, most probably there will be more instructions that address the same line. Instead of generating new misses, MSHR handles these requests. When the memory line becomes available, MSHR responds to all the loads pending on this memory line. Each MSHR entry has the block address that is required, load/store instructions waiting for this block, the part of the block address load/store instructions are waiting for, and a valid bit. If all MSHR entries are valid, the cache should be blocked because there are no more entries to track miss information. Figure 1.5 shows the MSHR entry. Destination represents the destination register or store buffer entry address.

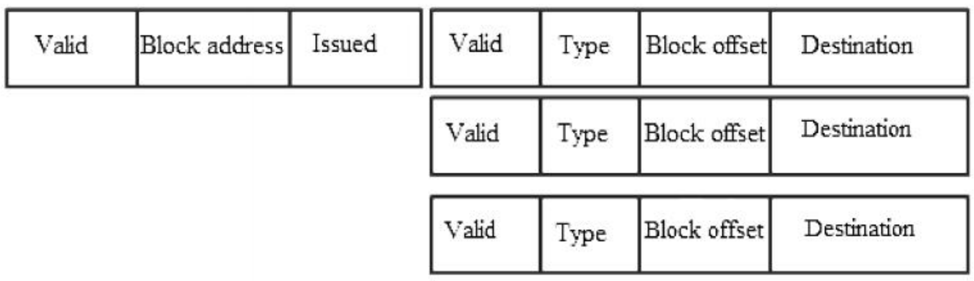


Figure 1.5: MSHR entry

- **Banking:** which was used in Intel Pentium where the memories are divided into banks and each bank can be accessed independently. Each bank does not depend on other banks so that several requests can be performed simultaneously. Although this design shows an increase in processor's throughput but its complex design, and difficulties in implementation reduced the usage of this solution. In addition, banking requires routing network and it must deal with bank conflicts.

- **Memory Access Scheduling**

Due to the gap in speed between core and main memory, memory access scheduling is considered one of the solutions to improve system's throughput and fairness. Since the number of cores integrated on-chip grows more rapidly than the off-chip pin bandwidth [12] which leads to an increase in the contention for main memory. One of the problems faces previous solutions is fairness when one or some applications are served at the expense of other running applications. As memory access behavior of different applications are different since some applications access main memory with high rate and others with low rate. This behavior leads to starvation of some applications which resulted in decreasing the fairness.

Memory access scheduling orders waiting main memory requests and decide which one shall be served first. Memory access scheduling has been developed for superscalar, multithreaded, and multi-core processors to enhance their performance. The concept of memory access scheduling is proposed for superscalar processors. However, in superscalar processors memory access scheduling was just re-ordering memory requests making use of memory hardware features to reduce memory access time. A good memory access scheduling is the one that improves either processor's throughput or fairness without degrading any of them. Different Memory access scheduling, advantages, and disadvantages of each one is introduced in Chapter 2 in details.

1.5 Thesis Contribution

In this thesis, we focus on the memory access scheduling for multi-core processors. We have selected multi-core processors that have separate instruction and data caches for each core, shared L2 cache and shared main memory as the micro architecture under investigation since it is the common design for todays processors and it is projected to be the design for years to come. We introduce a new memory scheduling algorithm called TB-LMI. TB-LMI was compared against several previous suggested memory access scheduling.

1.6 Thesis Outline

Chapter 2 includes literature survey about memory access scheduling. Chapter 3 proposes our new memory scheduling algorithm (TB-LMI) and how it works. Chapter 4 shows the simulator that used in our evaluation and modifications that were applied. Chapter 5 discusses the metrics, simulation parameters, and workloads used in evaluation. In chapter 6 we show the results of TB-LMI versus previous memory scheduling algorithms and it also contains a sensitivity analysis of TB-LMI. Chapter 7 summarizes the thesis and states our vision for the future work.

Chapter 2. Memory Access Scheduling In Literature

Memory access scheduling has been developed for superscalar, multithreaded, and multi-core processors to enhance/improve their performance. New memory scheduling algorithms were introduced and other scheduling algorithms were changed/ enhanced when we moved from single threaded processors to multithreaded and multi-core processors. Nowadays the concept of memory access scheduling is not limited to re-ordering memory requests from the same application/thread making use of memory hardware features. But it spans over scheduling requests from different applications/threads to allow better utilization of processor resources. In the following sections we will mention different memory access scheduling algorithms. Before that let's take a quick look on how main memory works.

2.1 Main memory operation mechanism

There are mainly two types of RAMs (SRAM and DRAM). SRAM is fast, more expensive, and less dense than the DRAM. SRAM is not used for high capacity such as main memory in personal computers. SRAM is typically used in the building of L1 and L2 caches in order to increase CPU speed. DRAM is typically used in the building of main memory [13].

Early, DRAM designs were implemented as one dimension buffer. This implementation causes main memory high latency. Also, it demonstrated difficulties in hardware implementation when the number of address lines are increased (i.e. main memory size is increased). In order to decrease main memory latency and to simplify hardware implementation, memory addresses were divided into row address and column address. In addition, main memory has a row buffer register. Memory data in row buffer register is usually available for other waiting requests with the minimum latency (described below).

Main memory is divided into banks in order to increase its bandwidth [14]. As main memory bandwidth increases, unicore CPU throughput increases. Each memory bank mainly consists of row address decoder, row buffer register, and a multiplexer as shown in Figure 2.1. When a memory request arrives to the memory bank, its address is divided to row address and column address. The row address is required to get the specific row from the bank storage and send it to the row buffer register. The column address is used through a multiplexer in order to get the required data.

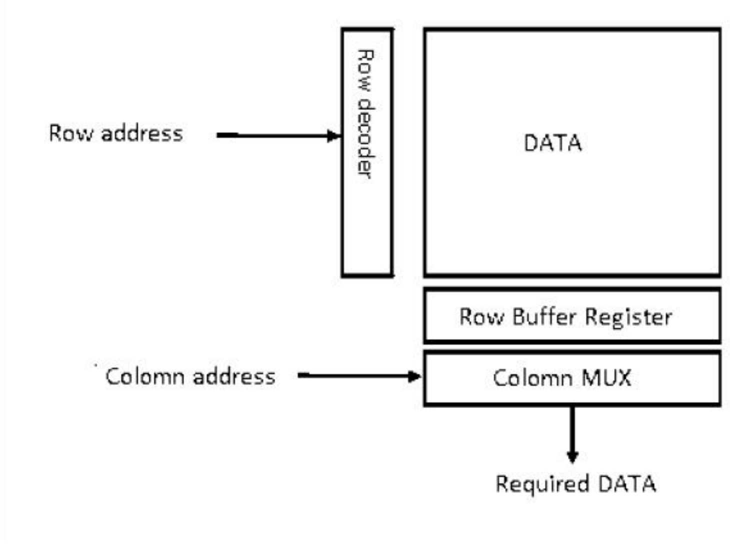


Figure 2.1: Memory bank

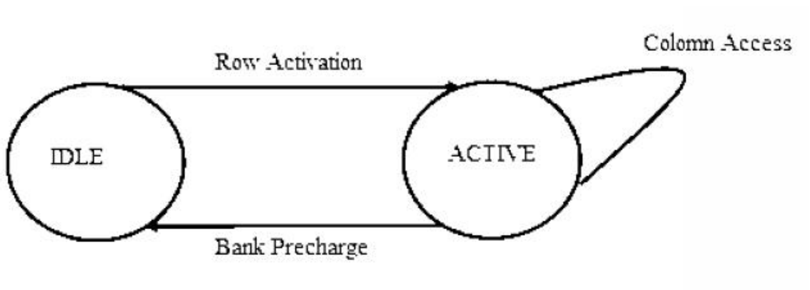


Figure 2.2: DRAM FSM

Modern DRAMs are three dimensions structure (bank, row, and column) [15]. Operations of modern DRAM are divided into three sub-operations which are row activation, column access, and bank precharge. A DRAM has two stable state, ACTIVE, and IDLE states as shown in DRAM Finite State Machine (FSM) in Figure 2.2. In the IDLE state, the DRAM is precharged and ready for row access. It will remain in this state until a row active operation is issued to the bank. Address lines must be used to select the bank and the row is active when it is sent to the row buffer register. Once the DRAM's activation latency has passed, the bank enters the ACTIVE state, where any number of column accesses (reads or writes) may be performed on this row. DRAM state is changed to IDLE state once column operations on this row are ended; bank precharge operation is issued to return the data back to its place.

Modern DRAM organization has divided the DRAM latency into three main categories, row hit latency, row closed latency, and row conflict latency. Row hit latency where there is data at the row buffer register and only column accesses are required. Row hit latency is the minimum latency of the main memory. Row conflict latency occurs when the required row is different from the current row in row buffer register. Row conflict requires bank precharge and row activation in order to perform any column access. Row conflict latency is the maximum latency of the main memory. Row closed latency occurs when row buffer register is empty, where row activation must be performed. Row closed latency is between row conflict and row hit latencies.

Modern processors have a main memory controller which is responsible of communication between main memory and processor. A main memory may have more than one memory controller to communicate with. Presence of main memory controller is mandatory as it handles requests to meet main memory latencies (row buffer hit, row buffer conflict, and row buffer closed latencies).

2.2 Schemes for Single Threaded Single Core processors

In single-threaded single-core processors, memory access scheduling focuses on re-ordering memory requests to improve memory performance by reducing the gap between processor speed and memory latency. Although main memory is a RAM device, its access pattern is not random. In other words, the ordering of memory requests change the latency time of memory access.

In [16] the authors focused on designing parallelized memory controller. They introduce SCHED which is a memory access scheduler. It is responsible for ordering the read/write requests, bank activates, precharges, and driving the SDRAM.

The Ph.D thesis [17] presented a compiler technology called access ordering. It tries to solve the memory bandwidth problem for scalar processors by utilizing memory system resources through memory accesses re-ordering.

In [18] the authors examined memory access ordering and tried to find the boundary for performance improvement. In [19] the authors introduced Memory Scheduling Unit (MSU). This unit is used to prefetch read requests, buffer write requests, and dynamically reorder the memory accesses in order to maximize the effective memory bandwidth. The main problem in the previously mentioned algorithms is that they are suitable for single-threaded processors only. They are not made to handle the case of ordering requests from different applications/threads.

There are some algorithms that were first proposed for single-threaded processors but then they were used in SMT and multi-core processors as mentioned in [20]. These algorithms are FCFS, hit-first algorithm, read-first algorithm, and age-based algorithm.

FCFS serves the request that arrives to the scheduler first regardless all other resources and factors. Its advantage is that it is very simple. Its disadvantage is that it does not take into account the criticality of resources or requests.

Hit-first algorithm gives row buffer hits more priority than row buffer misses. So, it gives more priority to requests that take less time. This algorithm corresponds to the same category of algorithms that exploits the memory hardware features to improve throughput.

Read-first algorithm gives memory read operations more priority than memory write operations. The idea behind this algorithm is that write operations are not a bottle because of the existence of write buffers.

Both hit-first and read-first algorithms can be used in collaboration with other algorithms. For instance, in [20] the authors used hit-first and read-first algorithms on top of request-based algorithm. In this case, read hit will always be scheduled before read miss, and read requests in general will be scheduled before write requests. In addition, the same type of requests is scheduled according to number of pending requests for each thread. i.e. the thread with the fewest number of pending requests is scheduled first.

As processor architectures changed from single core processors to multi-core processors, performance is not only the metric used to identify better processors. Multi-core processors should not only take care of performance but they also have to make sure that there is no application/thread suffers from starvation which reflects processor's fairness. Memory scheduling algorithms were introduced to guarantee the achievement of better performance, fairness or both in multi-core processors.

2.3 Schemes for SMT and Multi-core processors

The concept of memory access scheduling is discussed for SMT processors in [20]. Moving to SMT and multi-core, generally, increases contention on DRAM as the number of threads is increased. Improving multi-core processor's throughput is not the only objective of memory scheduling algorithms. Handling waiting main memory requests must be done precisely in order to increase the overall multi-core processors throughput without severely slowing down any running thread/application. That's to say without the correct handle of waiting memory requests, results may be devastating. Some of the running applications/threads may not execute an instruction for a long period time (fairness decreases or even suffer starvation). Processors that achieve high throughput in addition to high fairness are considered better than other processors. Memory scheduling is classified into thread un-aware memory scheduling and thread aware memory scheduling.

1. **Thread un-aware memory scheduling**

It is a type of memory scheduling where it has no information about the waiting memory requests in the main memory queue. As an example memory scheduling does not know which application/thread issued the waiting requests in the main memory queue and it does not know the waiting requests for each application/thread. Memory scheduling algorithms that were introduced for Single core processors are classified in this category.

First Come First Serve (FCFS) and First Ready First Come First Server (FR-FCFS) [21, 22] are considered the most popular thread un-aware memory scheduling for SMT and multi-core processors. FCFS where requests are served depending on which request has arrived first to the main memory. FR-FCFS is an algorithm that gives priority to requests with row buffer hit. In addition, read requests have higher priority than write requests.

2. Thread aware memory scheduling

It applies to memory scheduling algorithms that have some information of requests waiting in the main memory queue. As an example, scheduling knows the waiting requests that serve each application/thread and how many requests are issued from each application/thread. A large number of thread aware memory scheduling algorithms were introduced. We will quickly do a brief survey on most of the introduced thread aware memory scheduling algorithms.

- **Age-based scheduling**

  Age-based scheduling [23] gives the highest priority to oldest request when more than eight requests are presented to memory. This algorithm aims to improve fairness but it does not aim to improve throughput. Fairness is guaranteed because no thread will use the memory for large time alone.

- **Least REQuest (LREQ) scheduling**:

  Request priority depends on what application/thread issued this request and the total number of requests issued by this application/thread waiting in the main memory queue. A request from an/a application/thread that has the minimum number of waiting requests in the main memory queue has higher priority than other requests. In case of the presence of more than one request with the same priority, the oldest request has the highest priority. Although LREQ [24] scheduling showed an improvement in performance but it also showed degradation in fairness which is considered a disadvantage.

- **FR-LREQ scheduling**:

  FR-LREQ [25] is a modification of LREQ scheduling. FR-LREQ is a two level scheduling. First level that gives row buffer hit requests the highest priorities. Second level is when LREQ scheduling is applied.

  FR-LREQ may be classified in two ways in operation. First way, when row buffer hit requests have the highest priorities. In case of the presence of more than one request with row buffer hit, the oldest has the highest priority. FR-LREQ switches from level 1 to level 2 in case of the absence of row buffer hit requests. Second way, FR-LREQ starts to search for the application/thread that has the minimum number of waiting requests. FR-LREQ searched for row buffer hit requests from the application/thread with the minimum number of waiting requests (level 1). In case of the absence of row buffer hit requests, FR-LREQ starts to search for the application/thread that has the second minimum number of waiting requests. Level 1 starts again. FR-LREQ continues looping until a request is found. In case of the absence of level 1 request, level 2 starts alone. Figure 2.3 shows a flowchart that describes that second way of FR-LREQ. FR-LREQ showed a better performance than LREQ. FR-LREQ requires looping and checking the waiting requests before each memory access which adds extra latency.

In order to show the difference between FR-LREQ ways, suppose that two applications, application ‘A’ and application ‘B’ are running on a multi-core processor. ‘A’ has waiting requests in the main memory queue less than ‘B’. But ‘B’ has a row buffer hit request older than a row buffer hit request from ‘A’. FR-LREQ first way sets the highest priority to the row buffer hit request from ‘B’ although it has higher waiting requests in main memory queue than ‘A’. FR-LREQ second way sets the highest priority to the row buffer hit request from ‘A’ although it is not the oldest row buffer hit request in the main memory queue.

- **Fair Least Request Most Related (FLRMR) scheduling**:

FLRMR [24] targets both system’s performance and fairness. FLRMR uses approximately the same technique used by LREQ but with a different definition. FLRMR defines a threshold named FLRMR_PF that calculates waiting requests priorities. Requests with FLRMR_PF equals to 0 means that these requests are starving, they have higher priorities over others. In case FLRMR_PF of all waiting requests is greater than 0 which indicates there is no request is starving, the request with the minimum FLRMR_PF has the highest priority. In case more than one request has the same priority, the oldest has the highest priority. FLRMR_PF formula is shown in Eq. 2.1 where ‘*i*’ represents application/thread ID. Starvation time is shown in Eq. 2.2. If a waiting memory request waited in the main memory queue more than starvation time, *under_starvation_threshold* will be set to 0. This drives FLRMR_PF to 0, which means that this request has the highest priority. FLRMR scheduling flowchart is shown in Figure 2.4.

FLRMR was tested only on a single main memory bank (i.e. does not take the benefit of increasing main memory bandwidth) which is considered to be its main disadvantage. In addition, calculation of FLRMR_PF for each request must be performed with every access to the main memory which increases algorithm overhead.

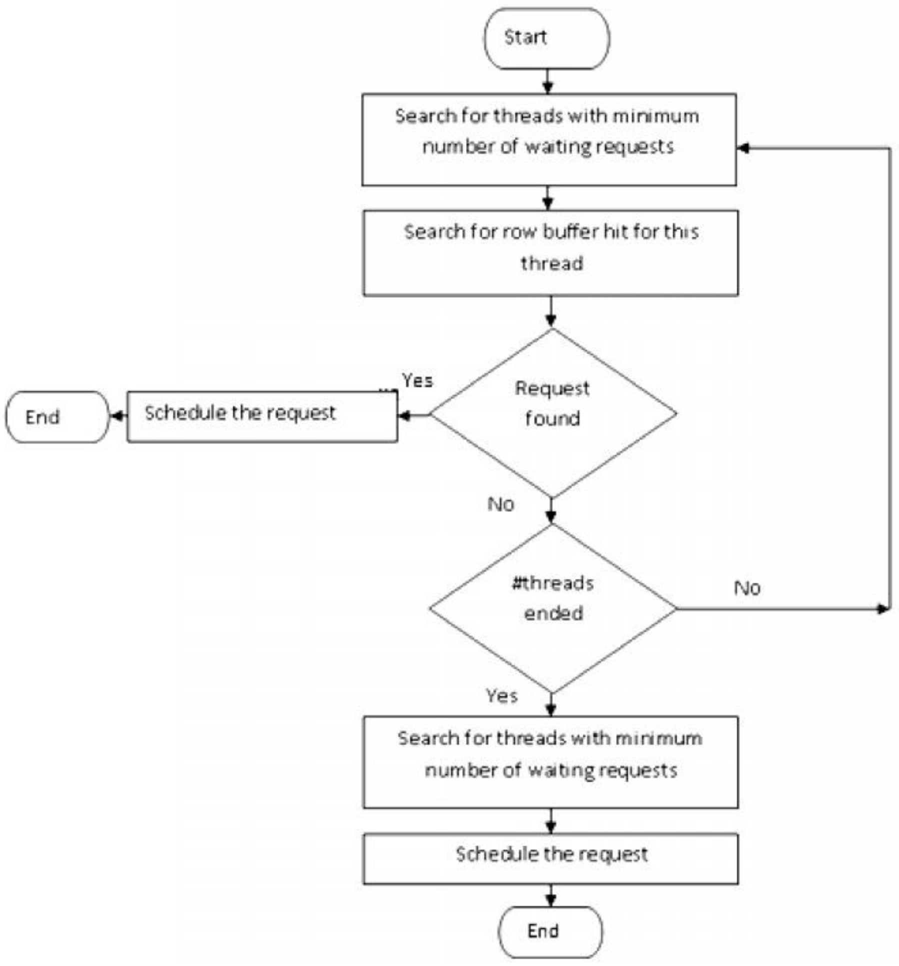

Start
Search for threads with minimum number of waiting requests
Search for row buffer hit for this thread
Request found
Yes
Schedule the request
End
No
#threads ended
No
Yes
Search for threads with minimum number of waiting requests
Schedule the request
End

Figure 2.3: FR-LREQ

$$FLRMR\_PF = \frac{\#\,pending\_requests_i^2 * under\_starvation\_threshold}{\#related\_requests_i + 1} \text{ .... (2.1)}$$

$$Starvation\_time = 2.5 * memory\_latency * number\_of\ \_threads \text{ ……... (2.2)}$$

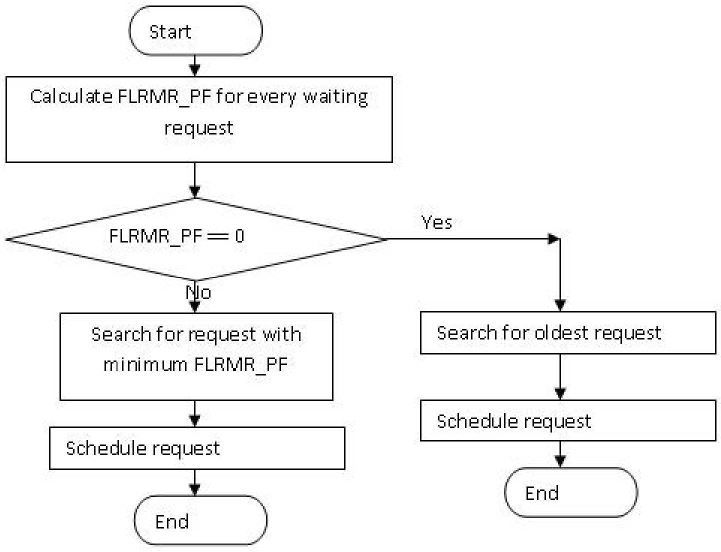

Start
Calculate FLRMR_PF for every waiting request
FLRMR_PF == 0
Yes
No
Search for request with minimum FLRMR_PF
Schedule request
End
Search for oldest request
Schedule request
End

Figure 2.4: FLRMR

- **Modified Re-Order Buffer (ROB) scheduling:**

Modified_ROB [24] is a modified version of ROB scheduling. ROB is implemented in processors that use algorithms to execute out-of-order instructions. ROB allows instructions to be committed correctly through register renaming. ROB scheduling gives the highest priorities to requests from the application/thread that has the highest number of reorder buffer entries. The idea behind this algorithm is that serving a request from the application/thread that has the highest number of reorder buffer entries most probably will release more waiting instructions than serving a request from other threads. Of course, this algorithm will help more in the cases when there is contention on reorder buffer.

Modified_ROB scheduling used a formula named Modified_ROB_PF to calculate priorities of waiting requests. The main target of this formula is to decrease the probability of finding starved waiting requests in main memory queue. Modified_ROB_PF formula is presented in Eq. 2.3 where *i* represents application/thread ID. The larger this factor is, the higher priority requests from an/a application/thread will have. If a waiting request is starved, *under_starvation_time* will be 0. This sets Modified_ROB_PF to infinity that gives this request the highest priority. Modified_ROB_PF flowchart is shown in Figure 2.5. Starvation time uses the formula that was introduced in Eq. 2.2. The main disadvantage of Modified_ROB scheduling is that it was tested only in a single bank memory. In addition, prioritizing a thread with the highest number of requests in ROB entries does not always guarantee the increase in overall processor's throughput, or fairness, or both.

$$Modified\_ROB\_PF = \frac{\#ROB\_entries_i^2 * (\#related\_requests + 1)}{under\_starvation\_threshold} \text{ ….. (2.3)}$$

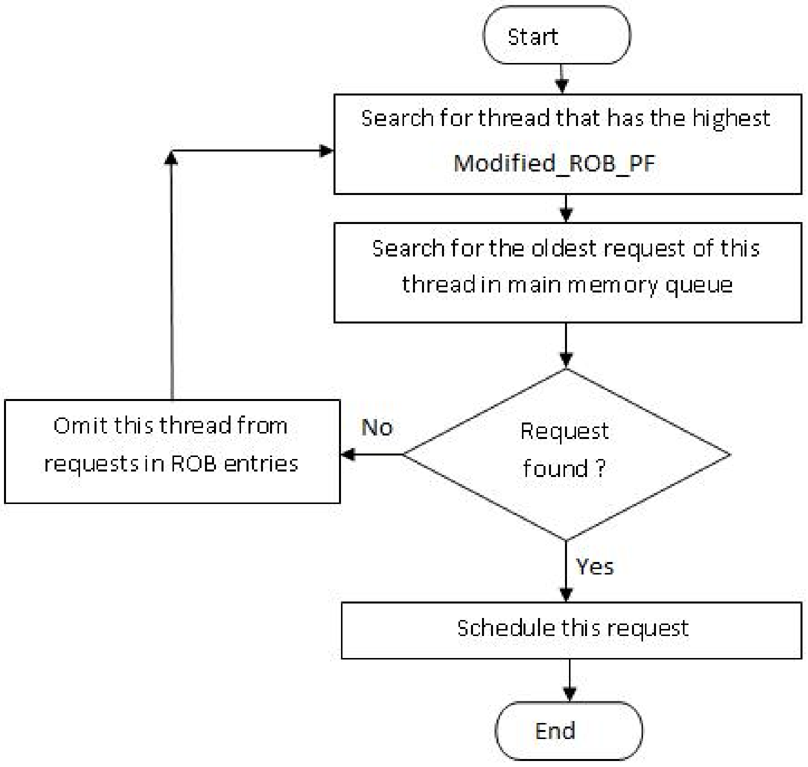

Start
Search for thread that has the highest Modified_ROB_PF
Search for the oldest request of this thread in main memory queue
Request found ?
No
Omit this thread from requests in ROB entries
Yes
Schedule this request
End

Figure 2.5: Modified_ROB

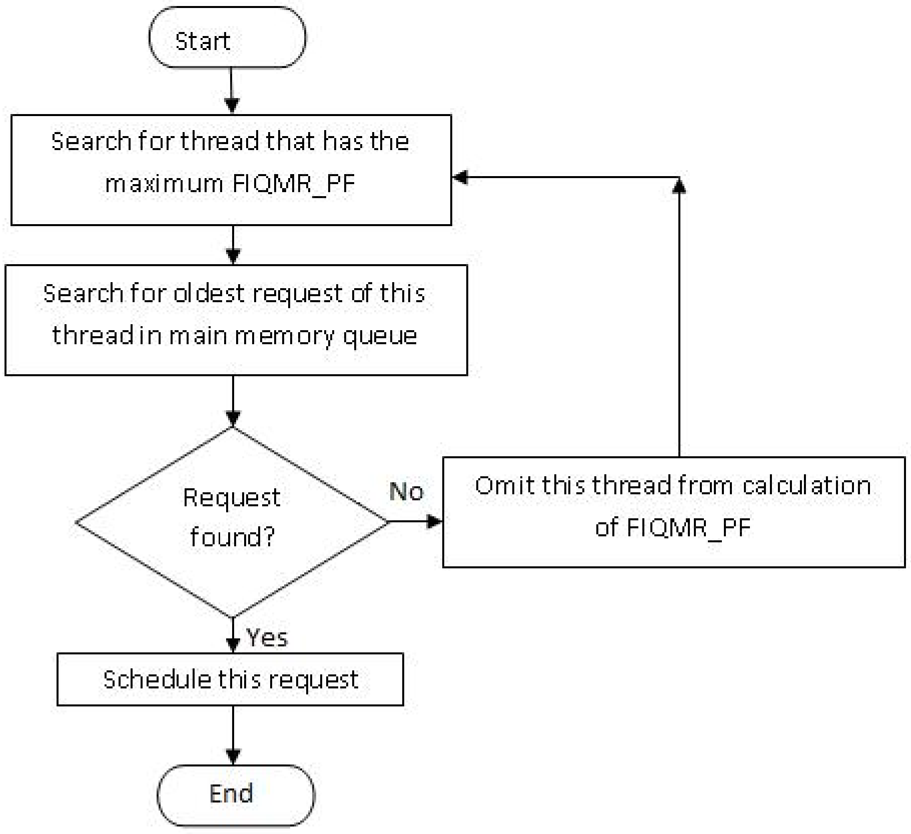

Start
Search for thread that has the maximum FIQMR_PF
Search for oldest request of this thread in main memory queue
Request found?
No
Omit this thread from calculation of FIQMR_PF
Yes
Schedule this request
End

Figure 2.6: FIQMR

- **Fair Issue Queue Most Related (FIQMR) scheduling:**

FIQMR [24] scheduling is a modified scheduling from IQ-based scheduling. IQ-based algorithm gives highest priority to requests from application/thread that has the highest number of issue queue entries. The idea behind this algorithm is that serving a request from the application/thread that has the highest number of issue queue entries most probably will release more waiting instructions than serving a request from other applications/threads. It will help in case there is a contention on issue queue.

FIQMR proposed a prioritization factor formula named FIQMR_PF to calculate waiting memory requests priorities. FIQMR_PF formula is shown in Eq. 2.4 where *i* represents application/thread ID. The larger this factor, the higher priority a waiting request will have. If a waiting request is starving (i.e. starvation time threshold has passed since its arrival), *under_starvation_threshold* will be 0 which means that FIQMR_PF of this request will be infinity. So, this request is starving as it will have the largest FIQMR_PF, and it will be served first. If no requests are starved then *under_starvation_threshold* will be 1, the oldest request with the highest FIQMR_PF will have the highest priority and will be served first. FIQMR scheduling flowchart is shown in Figure 2.6.

$$FIQMR\_PF_i = \frac{\#IQ\_entries_i^2 * (\#related\_requests_i + 1)}{under\_starvation\_threshold} \quad \text{……………. (2.4)}$$

- **Thread Clustering Memory (TCM) scheduling:**

TCM [26] scheduling classifies running applications/threads into memory intensive applications/threads and memory non-intensive application/threads (discussed in chapter 5). Memory non-intensive application/threads are added in the latency cluster. Memory intensive application/threads are inserted in the bandwidth cluster. Applications/Threads classifications are changed every pre-defined number of cycles called Quantum. TCM scheduling is divided into two algorithms clustering algorithm and insertion shuffling algorithm.

Clustering algorithm is scheduled every quantum where TCM calculates the total memory bandwidth (BW) and memory BW usage for each running application/thread, and insert it in the correct cluster. Initially, TCM calculates the total memory BW used by all threads through Eq. 2.5 where *i* indicate application/thread ID. TCM uses the equation introduced in Eq. 2.6 to calculate the memory BW usage of each application/thread. $BW\ usage_j$ is the memory BW of application/thread j. Also, TCM uses the equation introduced in Eq. 2.7 to classify applications/threads and sent it to the correct cluster. *ClusterThresh* is a predefined threshold and it is set to any value between *2/N* and *6/N* where *N* stands for the total number of applications/threads.

$$TotalBWusage_i = \sum_i BWusage_i \quad \text{………………………………………….. (2.5)}$$

$$SumBW = SumBW + BW_{usage\ j} \quad \text{…………………………………………… (2.6)}$$

$$SumBW \leq ClusterThresh\,.\,TotalBWusage \quad \text{……………………………….. (2.7)}$$

Insertion shuffling algorithm runs within the quantum. The main purpose of this algorithm is to increase fairness. Shuffling reduces memory interference by exploiting heterogeneity in the bank-level parallelism and row buffer locality among running applications/threads. TCM introduces the niceness metric shown in Eq. 2.8, where *i* stands for application/thread ID, *b* stands for bank-level parallelism, and *r* stands for row buffer locality. Every quantum, threads are sorted based on their niceness value to yield a ranking. The nicest application/thread receives the highest rank. TCM defines another pre-defined number of cycles less than quantum called *ShuffleInterval*. Every *ShuffleInterval*, the insertion shuffle algorithm perturbs nice ranking in a way that reduces the time during which the least nice applications/threads are prioritized over the nicest applications/threads, ultimately resulting in less interference. Implementation complexity and poor efficiency for low core count (2-core, and 4core systems) are among the main disadvantages of TCM.

$$Niceness_i = b_i - r_i \quad \text{……………………………………………………... (2.8)}$$

- **Parallelism Aware Batch Scheduling (PAR-BS):**

PAR-BS [27] scheduling is based on two main ideas (batch scheduling and parallelism-aware scheduling). Batch scheduling idea aims to group some memory requests into batches according to their arrival times and their requesting applications/threads. The requests within the oldest batch will have the highest priority. So, this algorithm is starvation-free. Parallelism-aware idea tries to make use of bank-level parallelism within the same batch.

- **Memory Efficiency (ME) LREQ scheduling:**

ME-LREQ was introduced in [25]. ME-LREQ prioritizes memory requests hitting on the row buffers and from cores that can utilize the memory more efficiently and where pending memory requests are fewer. ME of an/a application/thread is calculated from Eq. 2.9 where *i* represents application/thread ID. ME-LREQ depends on either offline profiling or online profiling. In [25] they used offline profiling due to the difficulties in implementation. They used lookup tables that store every possible pending request of threads. Eq. 2.10 shows how priorities are calculated. Applications/Threads with higher ME have higher priority.

$$ME_i = \frac{IPC^i_{\sin gle}}{BW^i_{\sin gle}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(2.9)}$$

$$\Pr iority_i = \frac{ME_i}{pending\_read_i} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots... \text{(2.10)}$$

- **Adaptive per-Thread Least-Attained-Service (ATLAS) scheduling:**

ATLAS scheduling which is introduced in [28] is based on LAS, long-time quantum, starvation free, and preserving single thread performance. LAS is used to increase the throughput. Long-term quantum is used to provide scalability; scalability does not need frequent and large information exchange. ATLAS is a starvation free scheduling algorithm as it defines a threshold value. ATLAS preserves single thread performance by preserving applications/threads bank-level parallelism. ATLAS increases processor’s throughput at the cost of fairness because the most memory intensive applications/threads receive the lowest priority and incur very high slowdowns which is considered a disadvantage.

- **Round Robin (RR) scheduling:**

RR [29] loops on applications/threads that have pending memory requests and serves a request from each application. RR targets fairness between applications/threads. RR does not take aging (how long the requests are in main memory queue) into consideration. RR scheduling is the best scheduling in fairness but it does not show any progress in performance. RR targets fairness only which is the main disadvantage.

- **Stall Time Fair Memory (STFM) scheduling**:

STFM [30] scheduling mainly aims to improve fairness. STFM estimates two values (T_{shared} and T_{alone}). Memory stall-time experienced by an/a application/thread is represented by T_{shared} when an/a application/thread runs among other applications/threads in the memory system. T_{alone} represents the memory stall-time experienced by the thread if it had been running alone in the memory system. Estimating T_{shared} is done by incrementing a counter each time the application/thread can not commit an instruction due to L2 cache miss. However, estimating T_{alone} is not straightforward. It is calculated by Eq. 2.11 where $T_{\text{int}erference}$ is the extra memory stall-time that an/a application/thread is suffering because memory requests from other applications/threads are being scheduled before the requests of the thread itself (if it has available waiting requests).

$$T_{alone} = T_{shared} - T_{\text{int}erference} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(2.11)}$$

After calculating T_{alone} and T_{shared}, the slowdown for each application/thread should be calculated by the Eq. 2.12. Then the maximum slowdown and the minimum slowdown of applications/threads that have at least one pending request are calculated. The ratio shown in Eq. 2.13 is calculated, if this ratio exceeds a predefined threshold, the next request scheduled is from the application/thread that has the highest slowdown. If the ratio is less than the predefined threshold, FR-FCFS scheduling is applied.

$$\frac{T_{shared}}{T_{alone}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(2.12)}$$

$$\frac{Maximum\ slowdown}{Minimum\ slowdown} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(2.13)}$$

2.4 Summary

In this thesis we are interested in SMT and multi-core architectures. Scheduling that was developed and introduced for single threaded processors are out of our scope. We used in our comparison different types of scheduling. FCFS and FR-FCFS were used from thread un-aware memory scheduling. FR-LREQ, FR-FLRMR, FR-FIQMR, FR-Modified_ROB, and TCM were used in our evaluation from thread aware memory scheduling.

FR-LREQ was used in our evaluation as it shows a competitive performance results compared to other scheduling. Although FR-FLRMR, FR-FIQMR, and FR-Modified_ROB were tested only for a single memory bank, but these scheduling showed competitive results in performance and fairness. One of the main reasons to use these scheduling was to see their effect when main memory bandwidth are increased. Also, TCM was used in our evaluation as it is considered the newest introduced multi-core scheduling. TCM showed the best results in performance and fairness against several previous scheduling such as PAR-BS.

The main purpose of this thesis is to introduce a new scheduling algorithm for multi-core processors. This new scheduling aims to improve multi-core processor's throughput without causing a devastating results in fairness for any running application/thread.

Chapter 3. Time Based Least Memory Intensive (TB-LMI)

3.1 Motivation

Some of memory scheduling algorithms discussed in chapter 2 address only scheduling in case the main memory with one bank and with no row buffer register such as FLRMR, Modified_ROB, and FIQMR. In addition, none of the previous memory scheduling algorithms even those who were tested on 8-core and 4-core processors have used a realistic main memory queue size. All of them used a main memory queue with 128 entries. These factors were a motivation to introduce a new memory scheduling algorithm.

Some of the previously introduced memory scheduling algorithms were modified in order to improve processor's performance and fairness. These algorithms are:

- **FR-FLRMR scheduling:**

  FR-FLRMR is a modified version of FLRMR. FR-FLRMR is presented in this thesis in order to take the benefit of row buffer hits. It was experimentally proven that FR-FLRMR showed a better improvement in processor's throughput and fairness than FLRMR as we will see in chapter 6.

  FR-FLRMR prioritizes row buffer hit requests over any other waiting request. If more than one row buffer hits are waiting, the oldest request has the highest priority. In case of the absence of row buffer hit requests, FLRMR scheduling is applied. FR-FLRMR has the same drawbacks of FLRMR.

- **FR-Modified_ROB scheduling:**

  FR-Modified_ROB scheduling is a modified version of Modified_ROB scheduling. It takes the advantage of row buffer hits. FR-Modified_ROB works in the same way that FR-FLRMR works. Also, FR-Modified_ROB scheduling shows a better processor's performance than Modified_ROB as we will see in chapter 6.

- **FR-FIQMR scheduling:**

  FR-FIQMR scheduling is a modified version of FIQMR scheduling. It takes the advantage of row buffer hits. FR-FIQMR works in the same way that FR-Modified_ROB works. Also, FR-FIQMR scheduling shows a better processor's performance than FIQMR as we will see in chapter 6.

Although these incremental enhancements show better processor's performance and fairness but those improvements are slim. Improvements in performance and fairness were around 2% and they did not show competitive results with other previous memory scheduling algorithms (TCM, and FR-LREQ). That is why a new memory scheduling algorithm is required.

3.2 TB-LMI Overview

TB-LMI is memory scheduling designed for multi-core processors. TB-LMI depends on online profiling of application/thread memory requests. TB-LMI is divided into two levels. Level 1 is responsible of setting row buffer hit requests with the highest priority. Level 2 is responsible for changing threads priorities every predefined cycle.

- **Level 1**:

  Although prioritizing row buffer hit requests reflects an improvement in processors performance but the main function of level 1 in TB-LMI is to guarantee fairness. No application/thread suffers from starvation.

  Applications/Threads are classified to memory non-intensive applications/threads and memory intensive applications/threads (described in Chapter 5 and Appendix A). Memory non-intensive threads have higher presence of row buffer hit requests than memory intensive applications when they run on a single core processor [28]. But when the main memory queue is decreased, the presence of row buffer hit requests from memory intensive applications/threads became higher than row buffer hit requests from memory non-intensive applications/threads (shown in Chapter 5).

- **Level 2**:

  Level 2 is responsible of calculating the number of memory accesses for each thread across all banks. Level 1 alone does not guarantee fairness and throughput. If level 1 causes unfairness or causes throughput degradation, level 2 will stand up to compensate the drawbacks of the priorities of applications/threads in level 1. Every thread has a counter in each bank memory controller which is incremented every access to a memory bank; this counter is reset every predefined cycle called *Schedule-Quantum* (SQ).

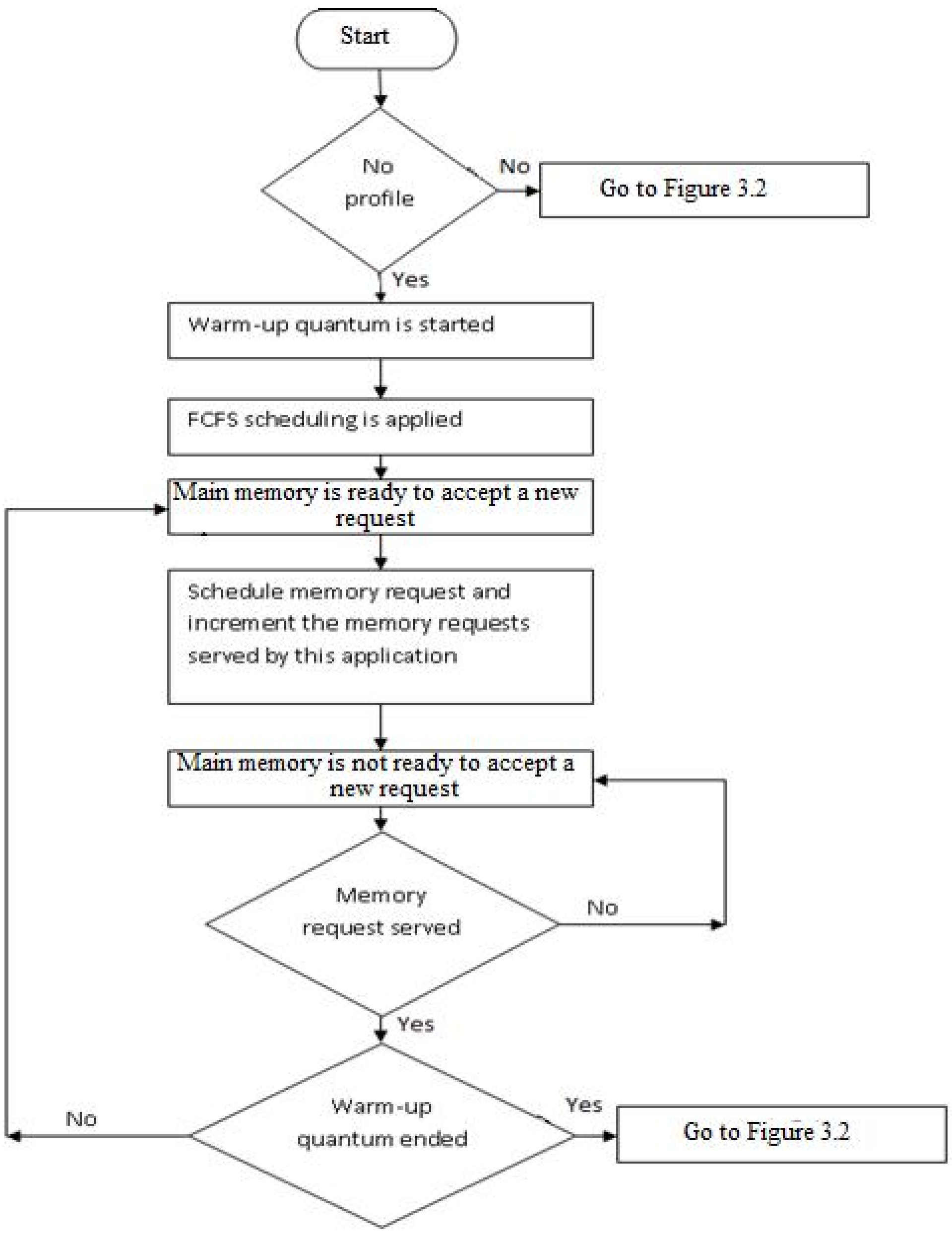
3.3 TB-LMI scheduling algorithm

TB-LMI starts scheduling when any bank memory controller is ready to handle a memory request and it ends once a request is scheduled. TB-LMI uses the profile calculated by itself (as will be described later in this section) and FCFS scheduling to set threads priorities. TB-LMI starts from level 1 where it searches for row buffer hit requests. If more than one row buffer hit request is found, the oldest request has the highest priority. In case of the absence of row buffer hit requests, TB-LMI switches to level 2. In level 2, TB-LMI sets the highest priority to requests issued from applications/threads that has the minimum number of scheduled memory access to the main memory. If an/a application/thread has more than one waiting request, the oldest waiting request will have the highest priority.

Initially when an application/thread starts execution on the processor, multi-core processor has no information about running applications/threads memory requests. Application’s/Thread’s classifications are unknown, what thread is memory intensive and what is memory non-intensive. TB-LMI defines a new quantum called warm-up quantum. During warm-up quantum FCFS scheduling where the oldest request in the main memory queue has the highest priority. FCFS is applied where an initial profile of running applications/threads are created. Every memory access, the running scheduling algorithm discovers which application/thread issued the scheduled request, and increment the number of memory requests served by this thread. At the end of *warm-up* quantum, application’s/thread’s priorities are calculated in Meta memory controller, and TB-LMI scheduling starts. Every SQ thread’s priorities are calculated. Figure 3.1 shows a flowchart that describes how TB-LMI performs in case of warm-up cycle.

At the end of SQ, every bank memory controller sends what it has collected about applications/threads profile (the number of memory accesses for each application/thread) to the Meta memory controller. The Meta memory controller accumulates the received profile results from each bank memory controller of each application/thread together and with the previous profile stored in the Meta memory controller. Meta memory controller starts to set priorities to threads. Applications/Threads with lower memory accesses than others have higher priorities. Meta memory controller broadcasts applications/threads priorities to all bank memory controllers to be used in the next SQ. Figure 3.2 shows a flowchart that describes how TB-LMI performs in case of SQ. Figure 3.3 shows how priorities are calculated in Meta memory controller. An example will be discussed in section 3.5 to show TB-LMI operation.

Applications/threads priorities are unified among all memory bank controllers as handling requests from several banks from the same application/thread increases the performance as introduced in [26, 27, 28]. In addition, every main memory bank acts as it is a separate main memory. A bank memory controller does not need any information from other bank memory controllers. It only exchanges information with Meta memory controller every SQ. At last, bank memory controllers do not have to wait until new applications/threads priorities received from the Meta memory controller. They can use the previous threads priorities.



Start
No profile
No
Go to Figure 3.2
Yes
Warm-up quantum is started
FCFS scheduling is applied
Main memory is ready to accept a new request
Schedule memory request and increment the memory requests served by this application
Main memory is not ready to accept a new request
Memory request served
No
Yes
Warm-up quantum ended
No
Yes
Go to Figure 3.2

Figure 3.1: TB-LMI in case warm-up cycles

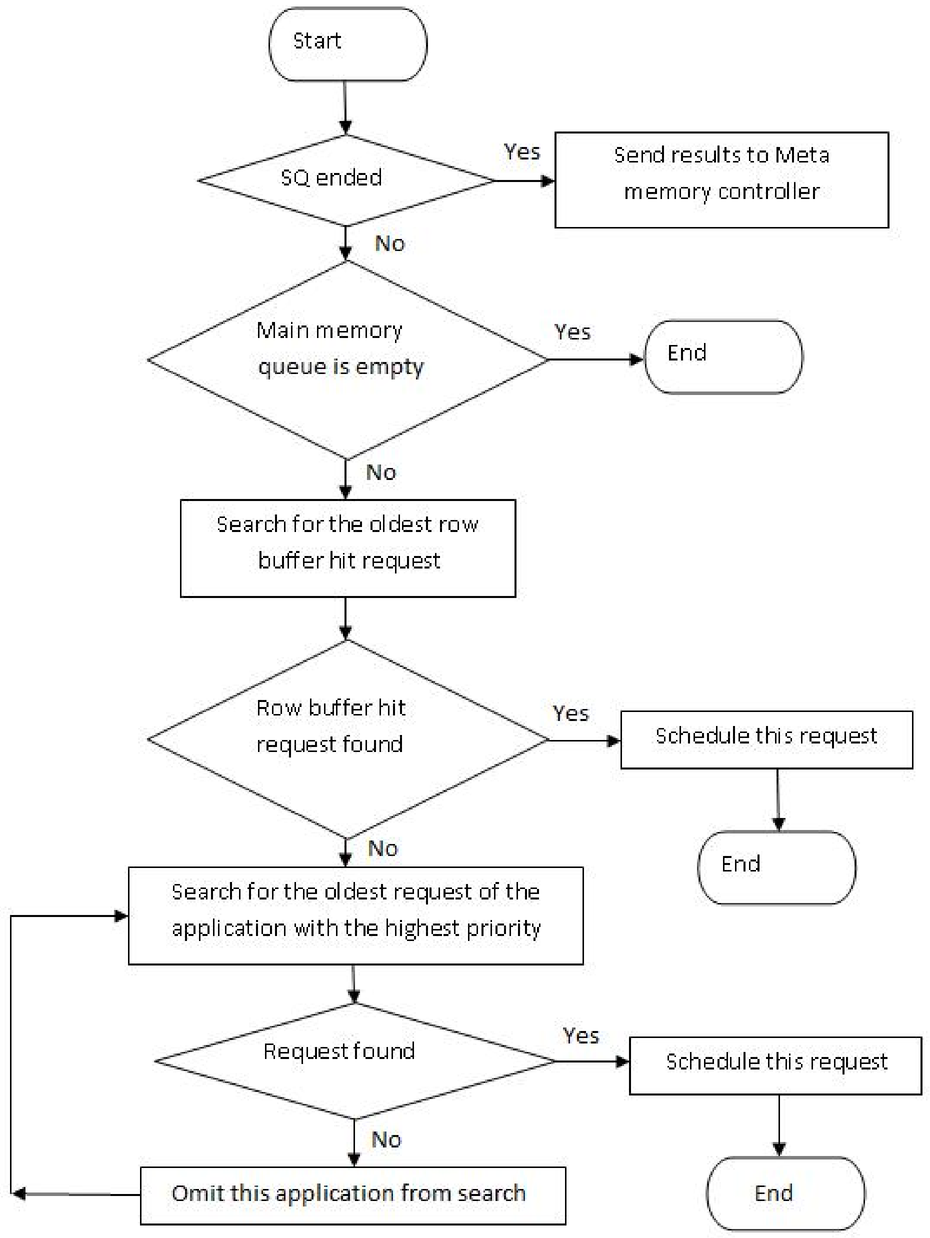

Start
SQ ended
Yes
Send results to Meta memory controller
No
Main memory queue is empty
Yes
End
No
Search for the oldest row buffer hit request
Row buffer hit request found
Yes
Schedule this request
End
No
Search for the oldest request of the application with the highest priority
Request found
Yes
Schedule this request
End
No
Omit this application from search

Figure 3.2: TB-LMI in case SQ cycles

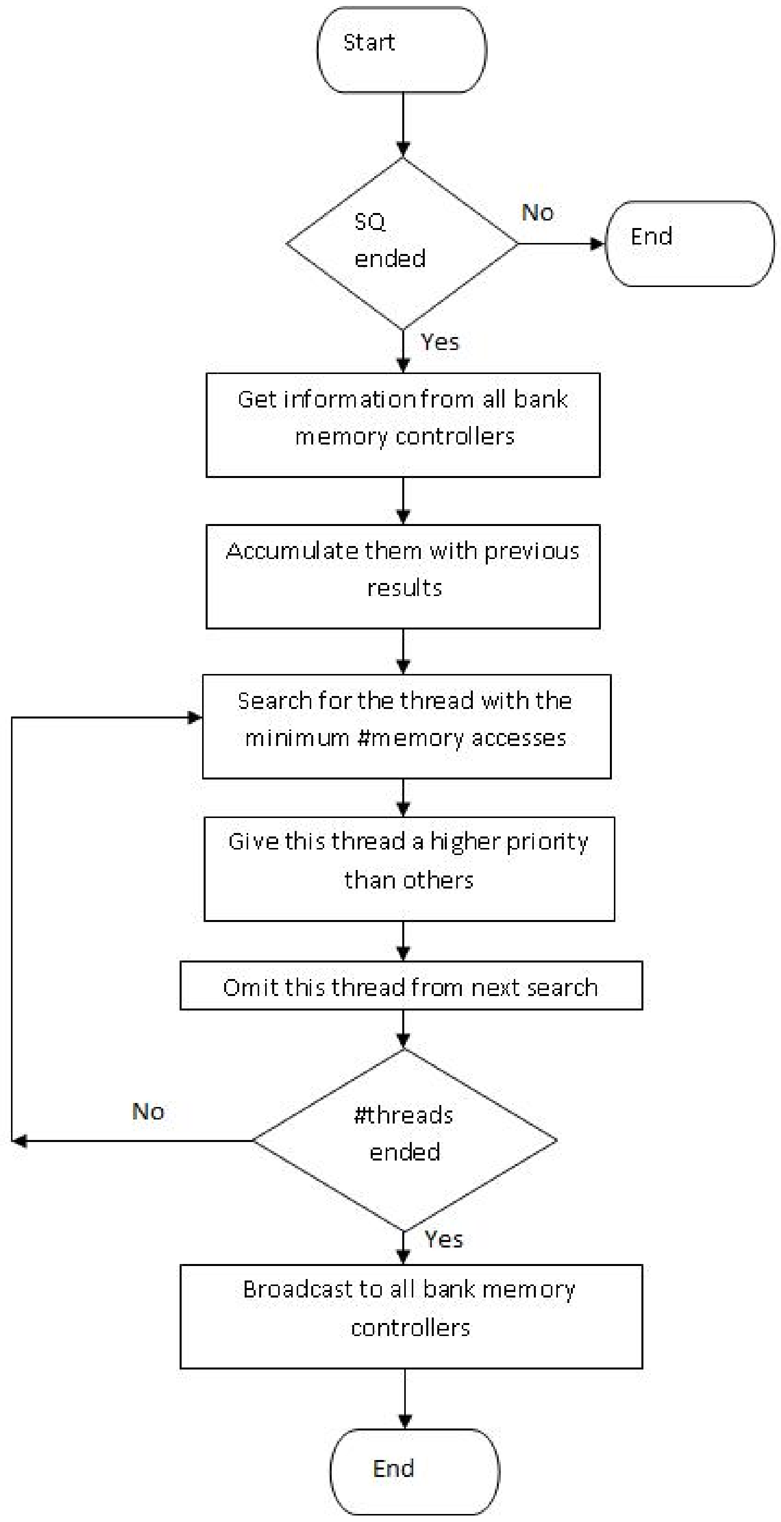

Start
SQ ended
No
End
Yes
Get information from all bank memory controllers
Accumulate them with previous results
Search for the thread with the minimum #memory accesses
Give this thread a higher priority than others
Omit this thread from next search
No
#threads ended
Yes
Broadcast to all bank memory controllers
End

Figure 3.3: Threads priorities calculation in Meta memory controller

3.4 Implementation and Hardware cost

TB-LMI requires hardware support to 1) monitor application's/thread's memory access behavior and 2) schedule memory requests as described before. Thread Memory Access Per Bank (TMAPB) and Thread Priority Status Register (TPSR) are added to each bank memory controller. TMAPB is a register that is reset after the *warm-up* quantum and every SQ. TMAPB size was chosen to fulfill the worst case in memory bank accesses as stated in Eq. 3.1. The worst case is encountered when an/a application/thread always access a single memory bank, a row buffer hit always occur, and only one application/thread is running in the multi-core processor. TPSR is added to store information sent form Meta memory controller as will be described in the next paragraph.

$$TMAPB = \log_2 \frac{Quantum}{row\,buffer\ hit\ latency} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \text{(3.1)}$$

The presence of a Meta memory controller is required as it is responsible of collecting information from all bank memory controllers to do the calculation required for applications/threads priorities. Applications/Threads priorities are saved in Meta-TPSR which is broadcasted to TPSR in all bank memory controllers. TPSR size equals to $N_{threads} * \log_2 N_{threads}$. In case of 8-core system, TPSR is of 24-bit size where each 3 bits represent an/a application/thread. TPSR register arranges application/thread priorities from high to low. The most three significant bits in TPSR represents the highest priority application/thread and the least three significant bits represents the lowest priority application/thread. Also, SQ register is implemented in the Meta memory controller. SQ register is incremented every clock cycle and it is reset when it reaches SQ value. As will be discussed in chapter 5, SQ value is set to 1M cycles. This makes a 20-bit register is enough for SQ register. In order to determine whether SQ is reset or not, an OR operation is performed on all bits in SQ register. If OR operation result is zero then the SQ is reset. At last, Total Memory Access (TMA) register is implemented in the Meta memory controller. TMA is designed to keep the history of a application/thread memory accesses in a multi-core processor for 100 million instructions of execution. TMA is designed to fulfill the worst case of memory accesses. The worst case is when only one thread is running on a multi-core processor and every instruction requires a memory access with row buffer conflict latency (maximum main memory latency). TMA register size is calculated through equation stated in Eq. 3.2.

$$TMA = \log_2 (row\,buffer\ hit\ latency * Max\ Instruction\ Number) \ldots\ldots\ldots\ldots\ldots \text{(3.2)}$$

Table 3.1: Hardware required for each bank memory controller

Memory Intensity		
Storage	**Function**	**Size (bits)**
TMAPB	A thread's memory accesses per bank.	$N_{threads}$ .TMAPB
Thread prioritization		
Storage	**Function**	**Size (bits)**
TPSR	Rank the threads according to their memory access	$N_{threads}.\text{Log}_2\, N_{threads}$

Table 3.2: Hardware required for Meta memory controller

Meta-Memory Controller		
Storage	**Function**	**Size (bits)**
TMA	A thread's memory accesses.	$N_{threads}$ .TMA

Additional bits are added to every memory request address to differentiate between different threads. The added bits equals to $\log_2 N_{threads}$. This adds 3 bits to every memory request address in case of 8 core processors while 2 bits are added in case of 4 core processors. In addition, First-Ready bit is added to determine requests with row buffer hits in memory queue. If First-Ready bit is set then it is a row buffer hit request.

Calculations are performed on memory requests searching for the right request to access the main memory does not require extra ALU. Calculations can be performed on any core with an idle ALU when the main memory bank is busy. In case of the absence of any idle ALU, calculations are not latency critical because the previous ranking can be used in the controllers while the next ranking is being computed or transferred.

Table 3.1, and Table 3.2 summarizes the required resources for both TMAPB, and TMA, for the whole multi-core processor. We can notice that the TMA, and TMAPB is directly proportional to the number of applications/threads, banks, and number of instructions. As the number of applications/threads and/or banks increases, the required hardware needed increases which is considered a drawback. Experimentally, as will be discussed in Chapter 5, applications profile store in Meta memory controller can be flushed every 100 millions instructions without degrading multi-core processor's throughput or fairness.

3.5 TB-LMI operation and request scenario

To illustrate how TB-LMI works and to make the scheduling algorithm clear we introduce a detailed example. Assume that we have a 4 core processor, a main memory with 2 banks, and a Meta memory controller. At processor's startup there is no information about running applications/threads. Warm-up quantum starts. At the end of the warm-up cycle, application's/thread's profiles were performed. As shown in Figure 3.4, a bank memory controller shows that cores 1, 2, 3, and 4 have 10, 2, 21, and 15 main memory requests have been served respectively. The other memory bank controller shows that cores 1, 2, 3, and 4 have 2, 6, 10, and 12 main memory requests have been served respectively. This information is sent from memory bank controllers to Meta memory controller where they are accumulated in TMA register and threads priorities are calculated in Meta TPSR. Also, TMAPB registers are reset. The Meta memory controller shows that the total number of main memory requests served for cores 1, 2, 3, and 4 are 12, 8, 31, and 27 as shown in Figure 3.4. Also, it shows that cores priorities from high to low are 2, 1, 4, and 3. Priorities are broadcasted and saved in TPSR in all memory bank controllers.

Now, the SQ starts, and TB-LMI starts. Let's consider 3 scenarios and show these scenarios in one memory bank controller. The first scenario in case of the presence of row buffer hit. As shown in Figure 3.5 (a), level 1 in TB-LMI starts and it searches for the oldest row buffer hit request. A request from core 4 has this property (circled) and it is scheduled before any other waiting request. The second scenario is when there are no row buffer hit requests. Level 1 returns a null value. TB-LMI switches to level 2 and starts to search for the highest priority requests. TPSR in the bank memory controller shows that requests from core 2 have the highest priorities. TB-LMI schedules the oldest request from core 2 (circled) in the main memory queue as shown in Figure 3.5 (b). Third scenario where level 1 returns null and there is no requests from core 2. Level 2 searches for the oldest request from core 1 (the second highest priority among cores) and schedule it. This technique is also applied on the other bank memory controllers.

At the end of SQ, a new profile is performed. As shown in Figure 3.6 a bank memory controller shows that the total number of requests served for cores 1, 2, 3, and 4 are 5, 17, 3, and 2 respectively. The other bank memory controller shows that the total number of requests served for cores 1, 2, 3, and 4 are 7, 10, 11, and 1 respectively. This information is sent to the Meta memory controller where they are accumulated with previous results and new priorities are calculated. Figure 3.6 shows that the total number of requests served for cores 1, 2, 3, and 4 are 24, 35, 45, and 30 respectively. Also, it shows that core's priorities from high to low are 1, 4, 2, and 3. This information is broadcasted to all bank memory controllers and the new priorities are applied in the next SQ.

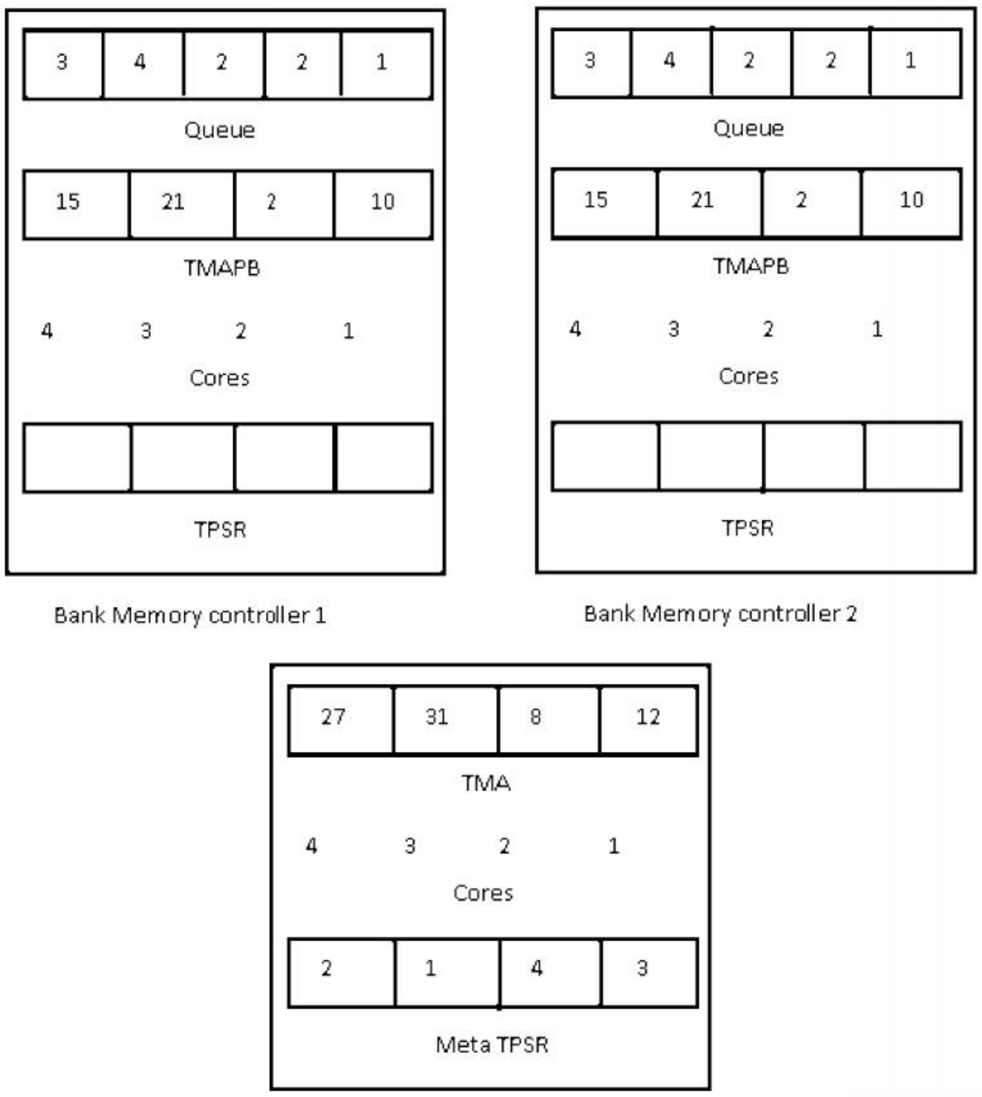


Figure 3.4: End of warm-up cycle

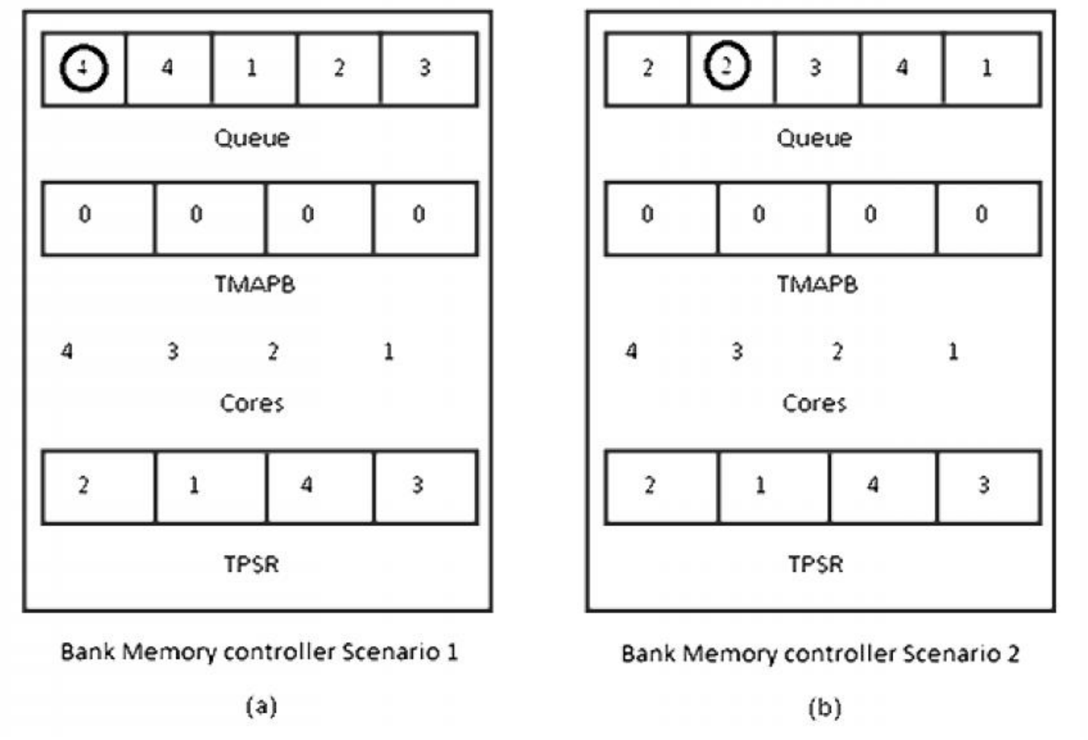


Figure 3.5: Bank memory controllers during SQ cycles

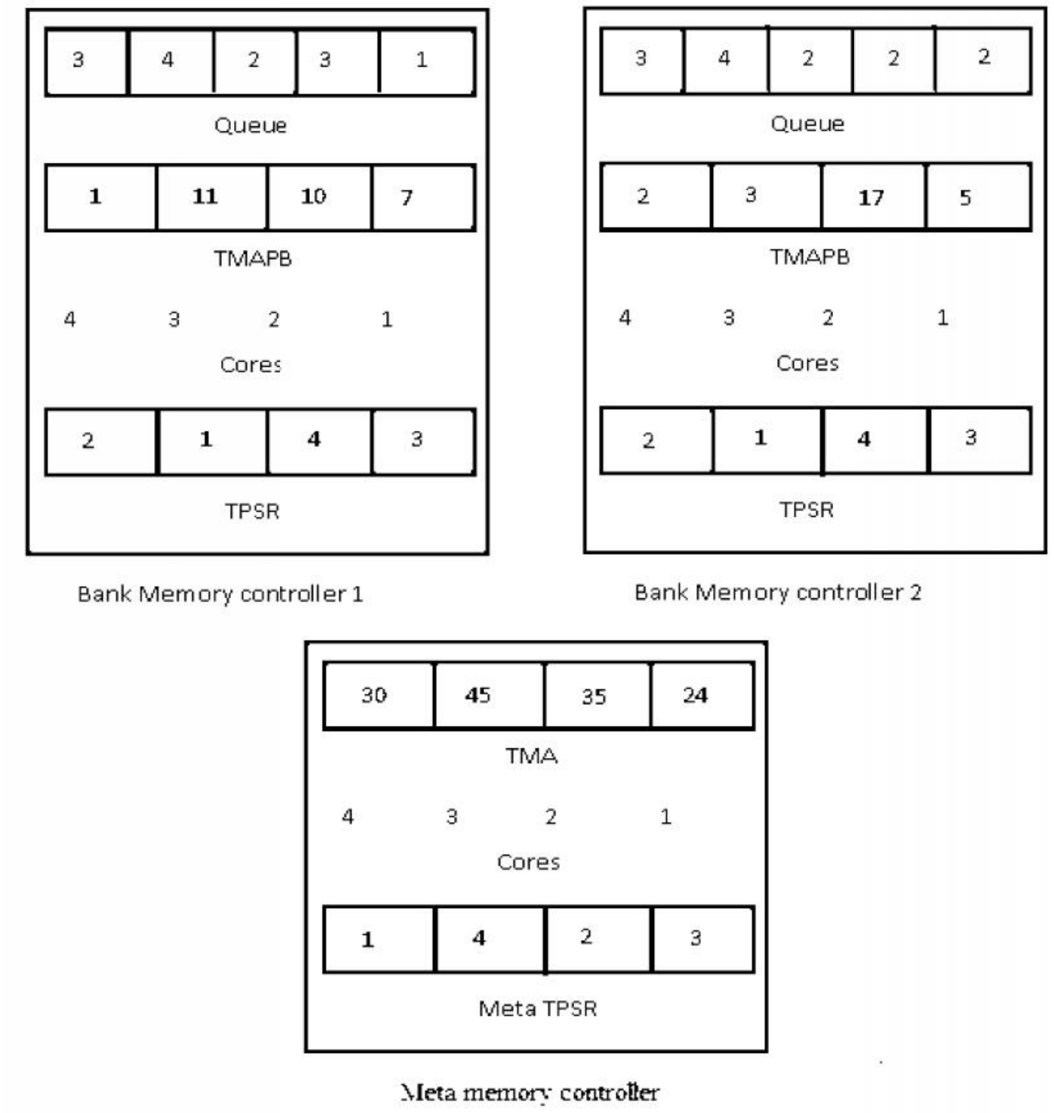


Figure 3.6: Bank memory controllers at the end of SQ cycles

Chapter 4. Simulator

4.1 Overview

Computer architecture simulators can be classified into functional simulators and cycle-accurate simulators. Functional simulators implement what programmers see. Functional simulators perform the actual execution so they keep track of the process state including architecture, registers, program count (PC) and memory. Cycle-accurate simulators are also called performance simulators. Cycle-accurate simulators implement the micro architecture, models processor resources, measures time, and implements hardware structures. Another simulator classification is according to its input; trace-based simulators and execution-driven simulators. Trace-based simulators read a trace of instructions saved from previous execution and then starts the simulation. Trace-based simulators are easier in implementation as they do not require functional components. Execution-driven simulators execute the program from scratch. Execution-driven simulators allow dynamic change of instructions to be executed depending on different input data which is considered an advantage. Execution-driven simulators give a wider area of simulation capabilities. Difficultly in implementation is considered one of the most disadvantages of execution-driven simulation.

There are a lot of simulators that could be used in our evaluation but we will introduce only two simulators 1) SimpleScalar simulator and 2) Multi2sim simulator. SimpleScalar and Multi2sim are freeware simulators. Both SimpleScalar and Multi2sim simulator can be classified as execution-driven simulators and they support both functional and cycle-accurate simulations. They are very authentic, widely used, and trusted in the research community.

1. **SimpleScalar simulator**

   It was developed by Todd Austin [31] while he was a PhD student at the University of Wisconsin Madison. SimpleScalar [32] is an open source computer architecture simulator written using ‘C’ programming language. SimpleScalar models a complete computer system (CPU, Cache and Memory Hierarchy). SimpleScalar was modified to support multithreaded and multi-core processors [33]. Also, it was modified in [24] in order to support different scheduling algorithms such as FLRMR, and LREQ. One of the main disadvantages of this simulator is that it does not support banking in main memory. Only main memories with one bank can be used.

2. **Multi2Sim simulator**

   Multi2Sim [34] is a simulation framework for CPU-GPU heterogeneous computing written in ‘C’ programming language. It includes models for superscalar, multithreaded, and multi-core CPUs, as well as GPU architectures. Multi2Sim is an open-source simulator and can be downloaded from the website in

[35]. Multi2Sim supports x86 CPU model, AMD Evergreen GPU, and AMD Evergreen Southern Islands GPU model. Also, it supports both functional and cycle accurate simulation. In addition, X86 binary files of SPEC2006, PARSEC 2.1, Mediabench, and SPLASH-2 benchmarks are free to be downloaded from the website in [35] and ready as inputs to Multi2sim. Multi2sim was used in our evaluation as it supports banking in main memory.

Multi2Sim has a property called Fast-Forwarding. Fast-Forwarding is functionally used to execute a program to a pre-defined number of instructions. A lot of modifications were performed in Multi2sim so that it could be used in our evaluation. These modifications are listed in the next section.

4.2 Multi2Sim simulator

Modifications that were performed on Multi2Sim simulator are:

1. Unpipelined memory

The default simulator uses a pipelined version of caches and main memory. A pipelined memory is a type of memory ready to accept another request after 1 cycle from the scheduled previous request. Pipelined memory makes all memory scheduling algorithms has the same results as there are no waiting requests so that memory scheduling algorithms could work on. Multi2Sim was modified to use a pipeline caches only. Main memory locks its port until a scheduled request is served.

2. Same_Block_Store_First

Store instructions issued in the default simulator from a core to L1 have to wait until all previous instructions are served whether they are load or store instructions. In order to increase processor's performance store instruction must not wait until previous instructions are served by L1. But an/a application/thread may be executed wrongly if a store instruction is performed before previous instructions accessing the same cache block. Applications/threads will use wrong data. Multi2sim was modified where store instructions wait only for any previous instruction only if they are accessing the same block in L1 cache.

3. Load_Access_First

Load instructions issued from cores to L1 in the default simulator have to wait for any previous store instructions whether they are accessing the same cache block or not. Multi2sim was modified where load instructions wait only for any previous store instructions only if they are accessing the same block in layer 1 cache. Load_Access_Fist idea is close to a FCFS and Read-First (FCFS-RF) [25] memory scheduling algorithm. FCFS-RF is an enhancement for FCFS. FCFS-RF serves read requests before write requests as the read requests will cause the processors to stall. Load_Access_First serves load instructions from L1 before store instructions to L1 to improve processor's performance.

4. Same_address_access_merge

Same_address_access_merge is an enhancement to Load_Access_First. Load instructions that are accessing the same cache block of previous store instructions are prioritized over any previous instructions. These load instructions do not access the layer 1 cache. These instructions load their data from the store instruction immediately.

5. Memory bank queue and row buffer register implementation

The default simulator uses no limitation on main memory queue. In addition, the implementation of row buffer for main memory and setting timing for row buffer hit, or row buffer conflict are not implemented (discussed in chapter 2).

Row buffer for main memory, limiting main memory queue, and main memory timings were implemented. A queue for each memory bank is implemented and its size can be controlled. In case a bank queue is full and a Layer 2 miss occurs, the memory bank sends a request to Layer 2 to be stalled. If Layer 2 is stalled, it stalls Layer 1 in case of the occurrence of Layer 1 miss which in return stalls the corresponding core(s). The flow starts to work again only in case of the presence of at least one free entry in the memory bank queue that caused stalling.

6. Implementing previously proposed memory scheduling algorithms

The default simulator uses FCFS when it accesses the main memory. FR-FCFS, FLRMR, FR-FLRMR, FR-Modified_ROB, Modified_ROB, FIQMR, FR-FIQMR, and TCM were implemented.

7. Multi2sim new options

New options were added in order to simplify the knowledge on how Multi2Sim simulator works and to make sure that modifications work correctly. These options are ***mini-debug***, ***workload-histogram***, ***main-memory-effective-latency***, ***mem-schedule-debug*, and *bank-parallelism-debug***. These new functions can be shown at **Memory System Options** by typing ***m2s --help*** in the terminal.

--mini-debug is responsible for discovering how Multi2Sim simulator works. It is added at any place in the code when we require knowing the output of the simulator at this part. It is recommended to be performed when a small number of instructions are simulated.

--workload-histogram calculates the histogram of running workload. ***workload histogram*** calculates the number of waiting main memory requests in every main memory queue every clock cycle. It can, also calculates the histogram of one benchmark as a workload can be only one benchmark.

--main-memory-effective-latency calculates the effective latency of the main memory. It calculates the actual number of cycles required to serve a memory request. It calculates the number of cycles of a memory request once it enters the main memory queue until the request is served by the main memory.

--mem-schedule-debug shows how the applied memory scheduling algorithm works. It is the same as ***mem-debug*** function introduced by Multi2Sim. But ***mem-schedule-debug*** gives only enough information about handled requests to the main memory that helps in troubleshooting.

--bank-parallelism-debug is the same as ***mem-schedule-debug***. It is used to make sure that banking in the main memory works correctly.

Chapter 5. Evaluation Metrics and Workloads

5.1 Metrics

For multi-core processors, metrics are classified into speedup metrics and fairness metrics. Speedup metrics measures multi-core processor performance. Fairness metric measures the immunity of a multi-core system in starving any working application. It is advisable to use speedup metric and fairness metric when we differentiate between multi-core processors. Using a single metric in comparison is not favored [27, 25, 36].

- **Speedup metric**

There are several ways in calculating the speedup of multi-core processors such as geometric mean [24] shown in Eq. 5.1, weighted speedup [26] shown in Eq. 5.2, and harmonic mean [26] shown in Eq. 5.3. '*N*' stands for the number of cores. IPC_{new} is the committed instructions per cycle in case of applying any memory scheduling. $IPC_{baseline}$ is the committed instructions per cycle in case of baseline scheduling is applied. Baseline scheduling could be any previous introduced scheduling such as FCFS. Weighted speedup measures multi-core processor's throughput. Harmonic mean measures a balance of multi-core processors fairness and throughput.

Geometric mean can be used alone in measuring multi-core processor's speedup. One of the main advantages of the geometric mean is that if there is a performance improvement in one thread and an identical performance degradation in another running thread, both changes in performance will be affecting the overall performance equally. On the contrary, neither the weighted speedup nor the harmonic mean are advisable to be used alone in evaluating multi-core processor's speedup [36].

$$Geometric\ mean = \sqrt[N]{\prod_{N} \frac{IPC_{new}}{IPC_{baseline}}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \ (5.1)$$

$$Weighted\ speedup = \frac{\sum_{N} \frac{IPC_{new}}{IPC_{baseline}}}{N} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \ (5.2)$$

$$Harmonic\ mean = \frac{N}{\sum_{N} \frac{IPC_{baseline}}{IPC_{new}}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots \ (5.3)$$

- **Fairness metric**

The same as speedup metrics, fairness metric has several formulas to calculate for multi-core processors. One of the formulas used is to measure the unfairness in [24]. Unfairness for a certain workload is defined as the ratio between the maximum slowdown to the minimum slowdown among all the applications/threads running in this workload. Slowdown is defined as the ratio between the application/thread stall time because of loads when it is running among other applications/threads in the workload to its stall time when it is running alone. Eq. 5.4 describes how to calculate unfairness, where '*i*' represents application/thread ID. The $stall\ time_i^{shared}$ term is the stall time faced by application/thread *i* when it runs with other applications/threads. Also, $stall\ time_i^{alone}$ term is the stall timed faced by application/thread *i* when it runs alone.

Another fairness metric is introduced in [26] by measuring the maximum slowdown caused to any running application as shown in Eq. 5.5. As the maximum slowdown increases, unfairness increases.

$$Mem\ Slowdown_i = \frac{Stall\ time_i^{shared}}{Stall\ time_i^{alone}},\ unfairness = \frac{\max_i\ MemSlowdown_i}{\min_j\ MemSlowdown_j} \ldots\ldots\ldots\ldots\ (5.4)$$

$$Maximum\ Slowdown = \max_i \frac{IPC_i^{alone}}{IPC_i^{shared}} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ (5.5)$$

- **Summary**

In [36] the authors suggested to use the weighted speedup shown in (5.2), ANTT which is the reciprocal of the harmonic mean shown in Eq. 5.6 with any fairness metric. In [36] the authors proved that weighted speedup and ANTT are more accurate and more intuitive conclusion than other metric. Fairness is also incorporated in the ANTT metric. In our evaluation we used the metrics that was suggested in [36]. We used a modified version of the fairness metric shown in Eq. 5.7 as it does not require further modification on Multi2sim. We used FCFS as a baseline scheduling.

$$ANTT = \frac{\sum_N \frac{IPC_i^{baseline}}{IPC_i^{new}}}{N} \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots\ (5.6)$$

$$Maximum\ Slowdown = \frac{\max_i \frac{IPC_i^{alone}}{IPC_i^{shared}}}{Maximum\ slowdown^{baseline}} \quad \ldots\ldots\ldots\ldots\ldots\ldots\ldots\ldots (5.7)$$

5.2 Simulation Environment

We have used the default CPU specification suggested by Multi2Sim. Table 5.1 summarizes the CPU specifications. Table 5.2 summarizes layer 1 cache specifications and Table 5.3 summarizes layer 2 cache specifications. Table 5.4 shows DRAM chip parameters.

Table 5.1: CPU specifications

Core	4 and 8 cores
Threads	1 thread/core
L1 cache	Separate instruction/data cache per core
Fast Forward	1 billion instructions
Fetch queue size	64 bytes
Re-order buffer size	64 Uops
Issue queue size	40 Uops
Load/store queue size	20 Uops
Branch predictor	2 level

Table 5.2: L1 specification

Size	64 KBytes
Sets	512
Associatively	2-way
Block size	64 bytes
Latency	2 clock cycles
Replacement policy	LRU
Ports	2

Table 5.3: L2 specification

Size	4 MBytes
Sets	16384
Associatively	4-way
Block size	64 Bytes
Latency	11 clock cycles
Replacement policy	LRU
Ports	4

Table 5.4: DRAM chip parameters

Number of Banks	4
Row buffer	256 row buffer/bank
Number of Controller	1 controller/bank
Row buffer hit latency	108 clock cycles
Row buffer conflict latency	216 clock cycles
Row buffer closed latency	140 clock cycles

5.3 Workloads

We have classified SPEC2006 benchmarks into memory intensive benchmarks and memory non-intensive benchmarks. Benchmarks that have high miss rate in L2 are classified as memory intensive benchmarks. Benchmarks that have low miss rate in L2 are classified as memory non-intensive benchmarks. For more details refer to Appendix A.

Workloads are designed to be 100% memory intensive workloads or 50% memory intensive workloads. 100% memory intensive workloads are workloads that have all benchmarks are memory intensive benchmarks. 50% memory intensive workloads are workloads that have half of the benchmarks are memory intensive benchmarks while the other half are memory non-intensive benchmarks. We have used all the benchmarks introduced in Table A.6 in Appendix A and all benchmarks are equally represented. Table 5.5 shows the workloads used in evaluation for 8-core processors. Table 5.6 shows the workloads used in evaluation for 4-core processors. Workloads are named where the first number represents the number of cores in a multi-core processor. The last number represents the workload ID. Workloads are differentiated (whether it is 100% memory intensive, or 50% memory intensive) through "*mem*", and "*mix*" words between the first and the last numbers. "*mem*" stands for 100% memory intensive workloads. "*mix*" stands for 50% memory intensive workloads. As an example, *8mem1* is workload number 1 for an 8-core processor and it is 100% memory intensive workload. *4mix3* is workload number 3 for a 4-core processor and it is 50% memory intensive workload.

- **Unlimited memory bank queue size analysis**

Workloads histograms for memory requests were calculated for FCFS scheduling. We have used the simulation parameters that were discussed in section 5.2. We used a cycle-accurate simulation for 100 million instructions. We have calculated and measured the histograms of workloads in each memory bank. Figure 5.1 and Figure 5.2 show the histogram for 4-core and 8-core workloads on average per single memory bank respectively. Average value is calculated where workloads histogram of memory banks are accumulated and divided by the number of memory banks. X-axis represents the number of waiting requests in main memory queue. Y-axis represents the number of cycles that the main memory queue has been occupied by waiting requests. Also, Table 5.7 and Table 5.8 show the average value of waiting requests in the main memory queue, how many cycles the main memory queue is busy, and how many cycles the main memory queue is free (main memory queue characteristics). It is observed from Figures 5.1 and 5.2 that the histograms of the workloads are close to exponential distribution. In case of 4-core workloads the average number of waiting requests does not exceed 4 waiting requests. 4-core workloads require at maximum a 26 entries memory bank queue. As the number of cores increases, memory requests increases, so the average number of waiting requests increases and the busy time of main memory queue increases. In case of 8-core workloads the average waiting requests does not exceed 6 requests. In addition, processor's performance is not improved or decreased when memory bank queue increases which is considered a waste of resources (will be discussed with further details in the next chapter). As an example, Table 5.9 shows a detailed analysis of a memory bank in case of 8mem1. Table 5.9 shows how many cycles a main memory queue entry was occupied by a waiting request. 8mem1 requires at maximum 39 entries from the main memory queue. Entry number 39 is only occupied by a waiting request for 155 cycles. Entry 39 in main memory queue is only busy for 2.51e-5% of the total number of 8-core processor cycles.

Table 5.5: 8-core system workloads

Workloads	Benchmarks	Memory intensity
8mem1	429.mcf, 462.libquantum, 483.xalancbmk, 401.bzip2, 471.omnetpp, 410.bwaves, 470.lbm, 436.cactusADM.	100%
8mix1	429.mcf, 462.libquantum, 483.xalancbmk, 401.bzip2, 473.astar, 464.h264ref, 403.gcc, 456.hmmer.	50%
8mix2	473.astar, 464.h264ref, 403.gcc, 456.hmmer, 471.omnetpp, 410.bwaves, 470.lbm, 436.cactusADM.	50%

Table 5.6: 4-core system workloads

Workloads	Benchmarks	Memory intensity
4mem1	429.mcf, 483.xalancbmk, 471.omnetpp, 470.lbm.	100%
4mem2	462.libquantum, 401.bzip2, 410.bwaves, 436.cactusADM.	100%
4mix1	429.mcf, 462.libquantum, 473.astar, 464.h264ref.	50%
4mix2	483.xalancbmk, 401.bzip2, 403.gcc, 456.hmmer.	50%
4mix3	471.omnetpp, 410.bwaves, 473.astar, 456.hmmer.	50%
4mix4	470.lbm, 436.cactusADM, 403.gcc, 464.h264ref.	50%

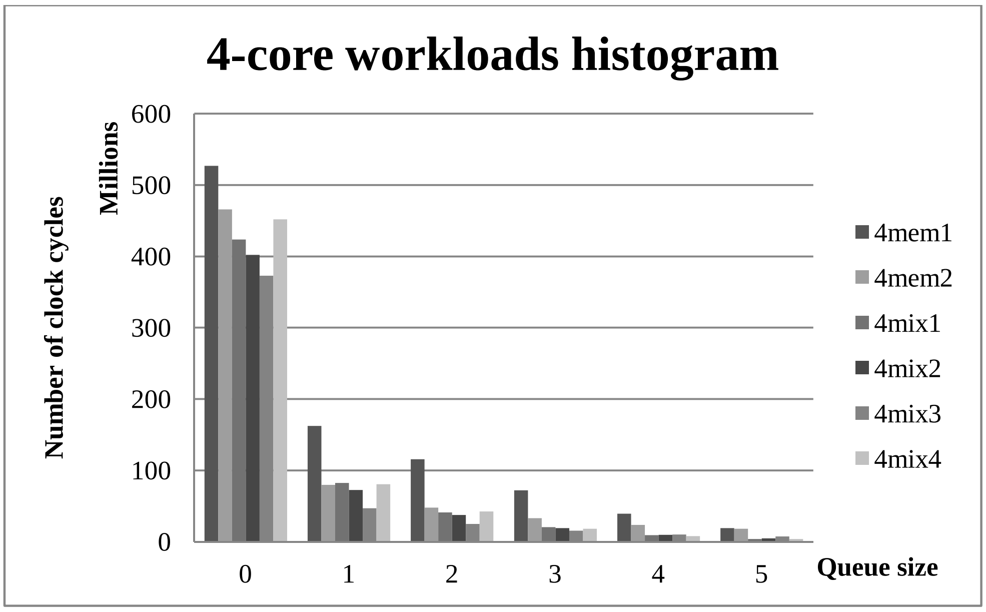


Figure 5.1: 4-core workloads histogram \ unlimited memory queue

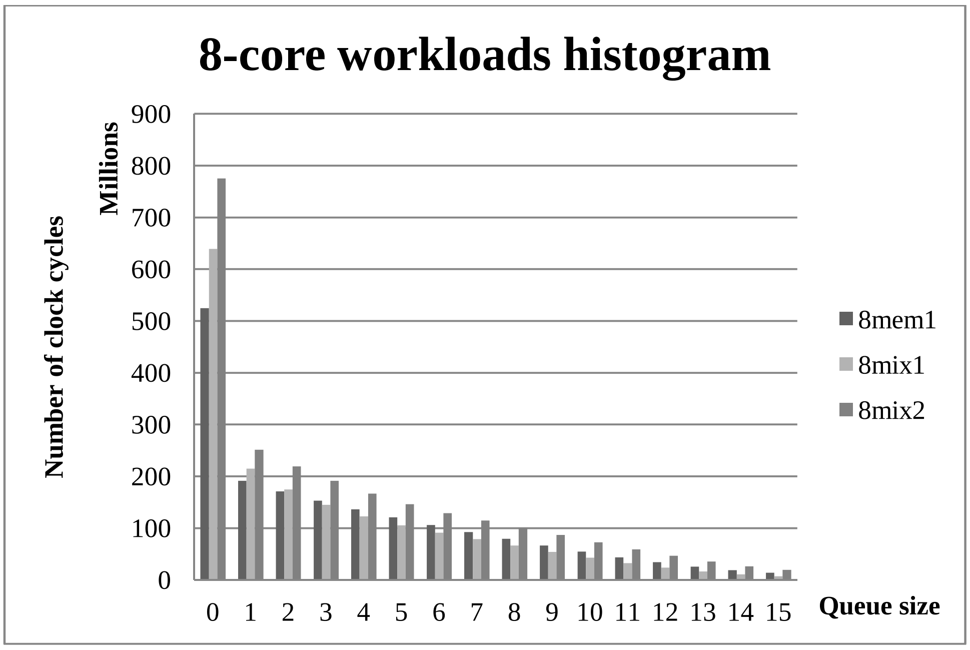


Figure 5.2: 8-core workloads histogram \ unlimited memory queue

Table 5.7: 4-core system workloads memory characteristics

Workload	4mem1	4mem2	4mix1	4mix2	4mix3	4mix4
Number of cycles	947216758	715462985	582623277	550951502	490305643	611820969
Queue free (%)	55.6033	65.1203	72.683	72.957	76.02	73.861
Queue busy (%)	44.3967	34.88	27.3167	27.043	23.98	26.139
Queue max (%)	0.11302	1.0944	0.0266	0.1333	0.467	0.171
Average of waiting requests	2.24	3.33	1.86	2.068	2.668	2.013

Table 5.8: 8-core system workloads memory characteristics

Workload	8mem1	8mix1	8mix2
Total number of cycles	1860148029	1835934804	2479952824
Queue free (%)	28.20484408	34.81306598	31.25208536
Queue busy (%)	71.79515592	65.18693402	68.74791464
Queue max (%)	2.08317E-06	1.79745E-06	2.4194E-06
Average of waiting requests	5.55	4.89	5.6266

TABLE 5.9: 8mem1 detailed histogram for memory bank 0 / unlimited memory bank queue

Memory bank queue entry	Number of clock cycles	Memory bank queue entry	Number of cycles
1	166852771	21	1814744
2	158683505	22	1165538
3	150911127	23	741460
4	142158298	24	474154
5	132290187	25	301054
6	122031662	26	181880
7	110723887	27	109366
8	98132202	28	68279
9	84313993	29	44453
10	70233529	30	21812
11	56606103	31	9334
12	44059628	32	4903
13	33352896	33	3470
14	24681769	34	2988
15	17909201	35	1236
16	12724362	36	668
17	8873526	37	467
18	6098337	38	1003
19	4133808	39	512
20	2788130	40	155

- **Limited size of memory bank queue analysis**

We have put a limit to the main memory queue sizes to be 8, 16, 24, and 32 entries to decrease area and to study its effect on multi-core processors. Figure 5.3 to Figure 5.14 show 8mem1, 8mix1, and 8mix2 workloads histograms for each memory bank when each memory bank queue has 8, 16, 24, and 32 entries respectively. Table 5.10 to Table 5.13 show the main memory queue characteristics in case each memory bank queue has 8, 16, 24, and 32 entries.

We can observe that the memory banks have the same histogram across workloads. Workloads histogram in case of limited memory bank queue is different than unlimited memory bank queue. Workloads histograms are close to X^2 distribution in case of 8 entries memory bank queue. Workloads histograms are changed to exponential distribution when memory bank queue were increased to 16, 24, and 32 entries. Also, Workloads histograms show that the latest entry in memory bank queue occupancy with waiting requests is increased when we switch from unlimited memory bank queue size to limited size of memory bank queue. As an example, in case of 8mem1 and memory bank 0, and 8 entries memory bank queue, entry 8 occupancy with waiting requests is increased from 98 million cycles to 191 million cycles (doubled). This was expected as the main memory queue size was decreased from 39 entries to 8 entries in case of 8-core workloads the occupancy of memory bank queue with 8 waiting requests is increased.

The average number of occupied entries was increased from 4.7 waiting requests when the bank memory queue has 8 entries to 5.74 waiting requests when the bank memory queue has 32 entries in case of 8mem1. The same observation is seen on 8mix1 and 8mix2 but with different numbers. Also, the main memory queue busy time increases as main memory queue increases. This results in increasing processor's power usage as more energy will be required to save the data in memory bank queue. Thorough power analysis is not performed in thesis but it is part of our future research directions. In case the memory bank queue size has 8 entries the main memory queues are fully occupied or has any number of entries (busy) for 61%, 56%, and 55% of the total number of executed cycles for 8mem1, 8mix1, and 8mix2 workloads respectively. When the memory bank queue is increased to 32 entries the main memory queues are busy for 66%, 58%, and 56% from the total number of executed cycles for 8mem1, 8mix1, and 8mix2 workloads respectively. When we compare the average number of waiting requests and the busy time of memory bank queue in case it is 8 entries and in case it is 32 entries. We can observe that the average value was increased by 20%, 5.05%, and 10.26% when we switch from 8 entries memory bank queue to 32 entries memory bank queue in case of 8mem1, 8mix1, and 8mix2 respectively. Memory bank queue occupancy was increased by 7.7%, 4.49%, and 2% when we switch from 8 entries memory bank queue to 32 entries memory bank queue for 8mem1, 8mix1, and 8mix2 respectively. As the memory bank queue busy time increases, more energy is consumed. This indicates an increase in power dissipation of the multi-core processor and more complexity in design.

In summary, it is observed that the occupancy of large entries in the main memory queue by memory requests is low. Limiting the main memory queue size does not affect processor's performance and/or fairness which will be observed in the next chapter. It is preferred to use memory bank queue size with small number of entries to decrease multi-core processor's area and dissipation power.

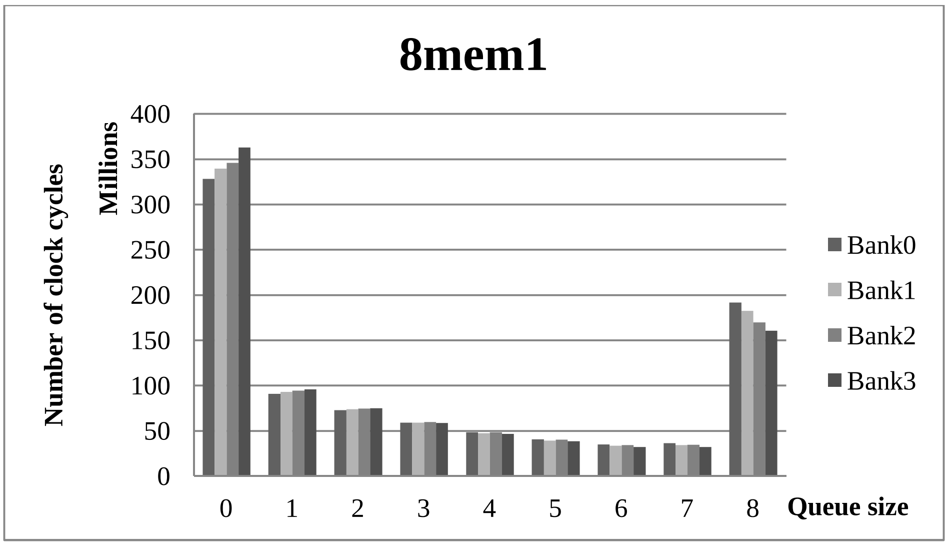


Figure 5.3: 8mem1 histogram \ 8 entries bank queue

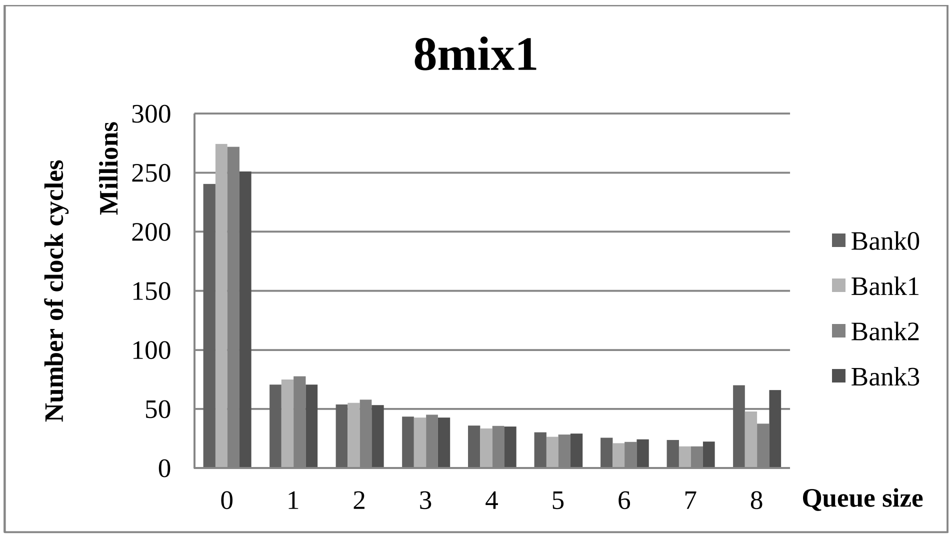


Figure 5.4: 8mix1 histogram \ 8 entries bank queue

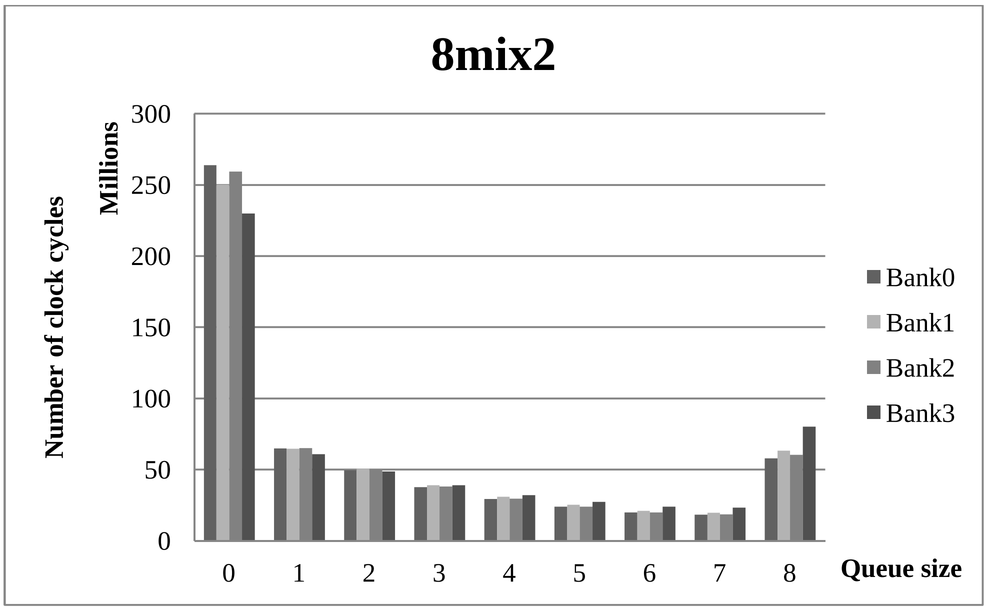


Figure 5.5: 8mix2 histogram \ 8 entries bank queue

Table 5.10: Memory characteristics \ 8 entries bank queue

Workload	8mem1	8mix1	8mix2
Total number of cycles	902836282	594418149	565075165
Queue free (%)	38.1204461	43.65324556	44.38336872
Queue busy (%)	61.8795539	56.34675444	55.61663128
Queue max (%)	19.51296547	9.322834833	11.58385128
Average of waiting requests	4.77	3.96	4.19

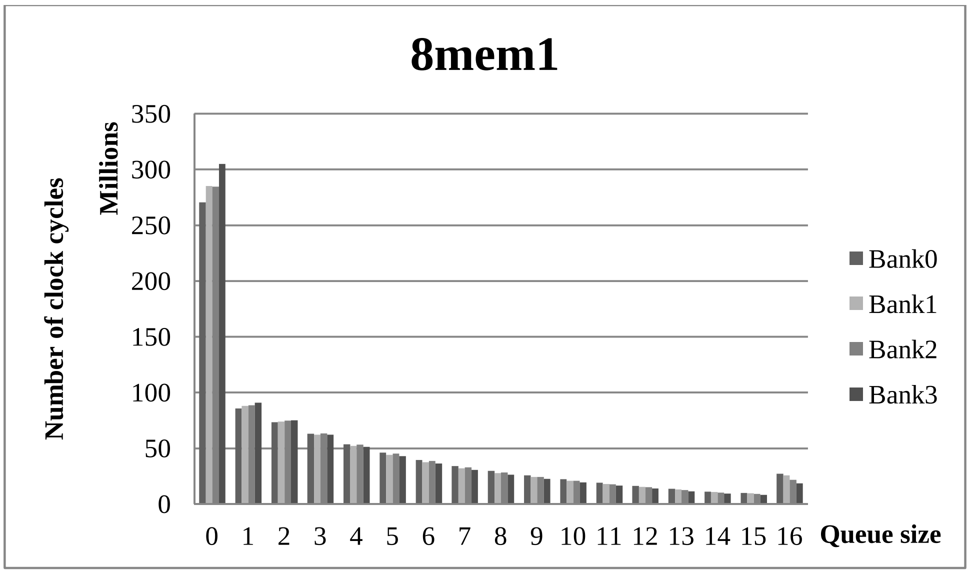


Figure 5.6: 8mem1 histogram \ 16 entries bank queue

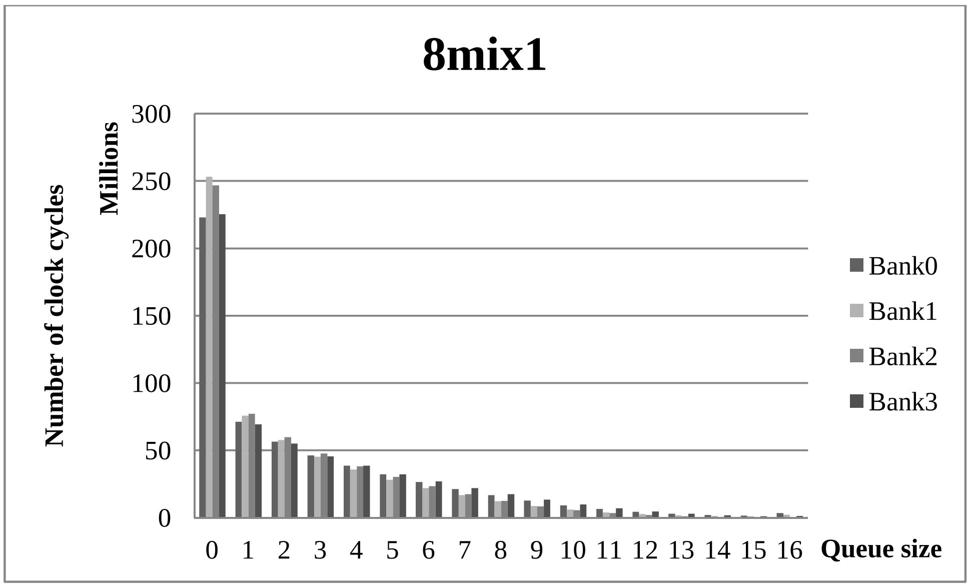


Figure 5.7: 8mix1 histogram \ 16 entries bank queue

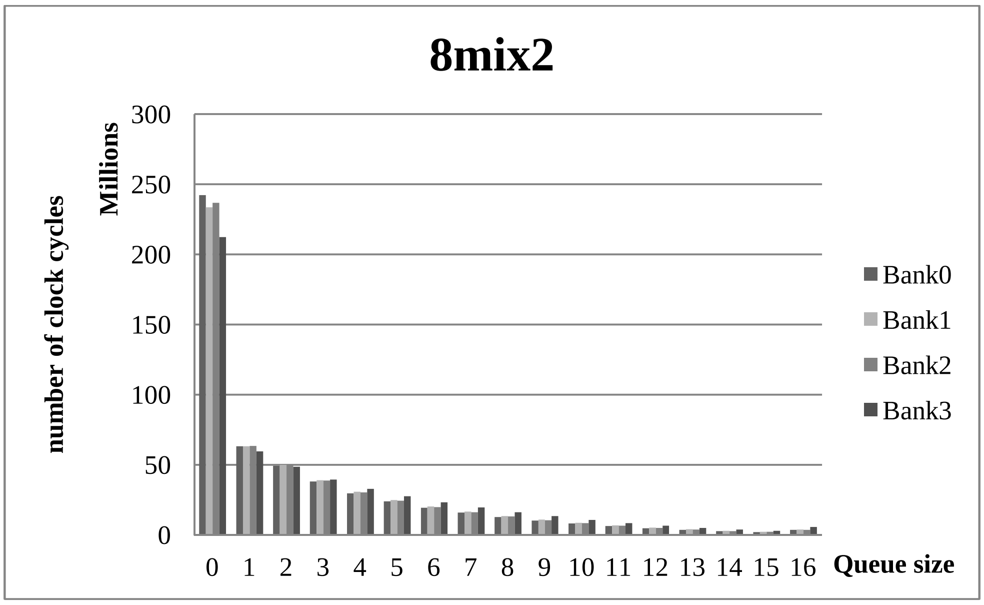


Figure 5.8: 8mix2 histogram \ 16 entries bank queue

Table 5.11: Memory characteristics \ 16 entries bank queue

Workload	8mem1	8mix1	8mix2
Total number of cycles	841526803	575042081	535487308
Queue free (%)	34.01969275	41.21936243	43.19377122
Queue busy (%)	65.98016787	58.78034142	56.80579086
Queue max (%)	3.335152386	2.572413696	2.593555495
Average of waiting requests	5.71	4.14	4.63

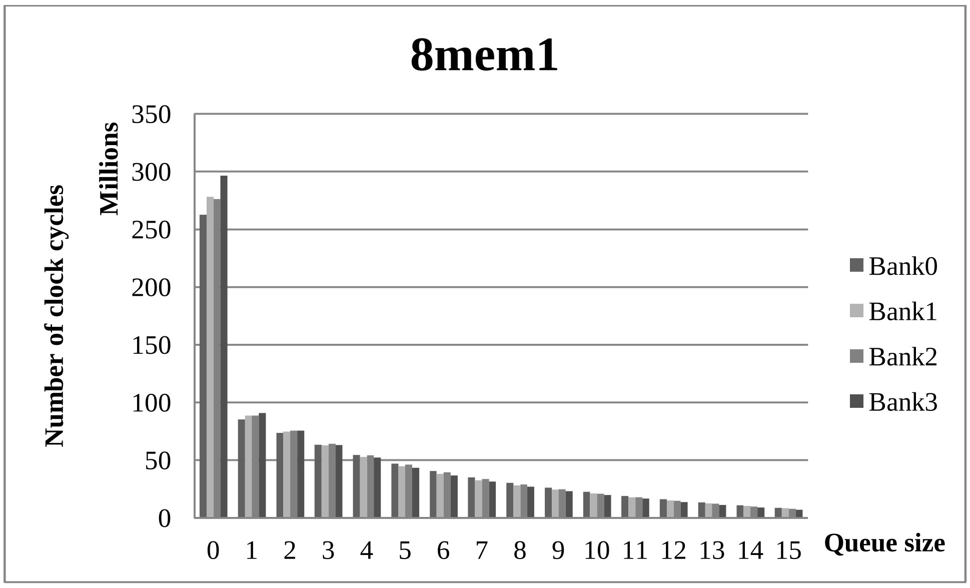


Figure 5.9: 8mem1 histogram \ 24 entries bank queue

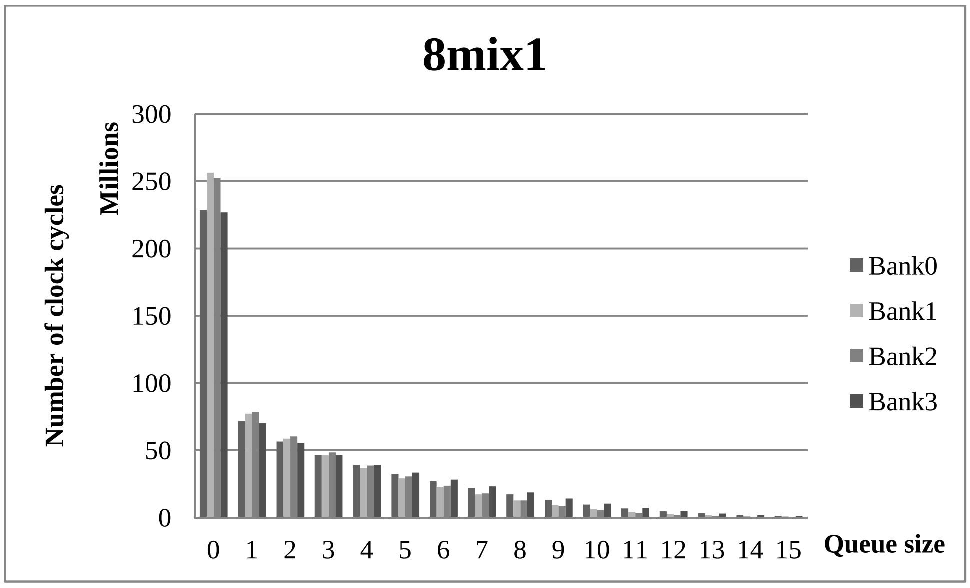


Figure 5.10: 8mix1 histogram \ 24 entries bank queue

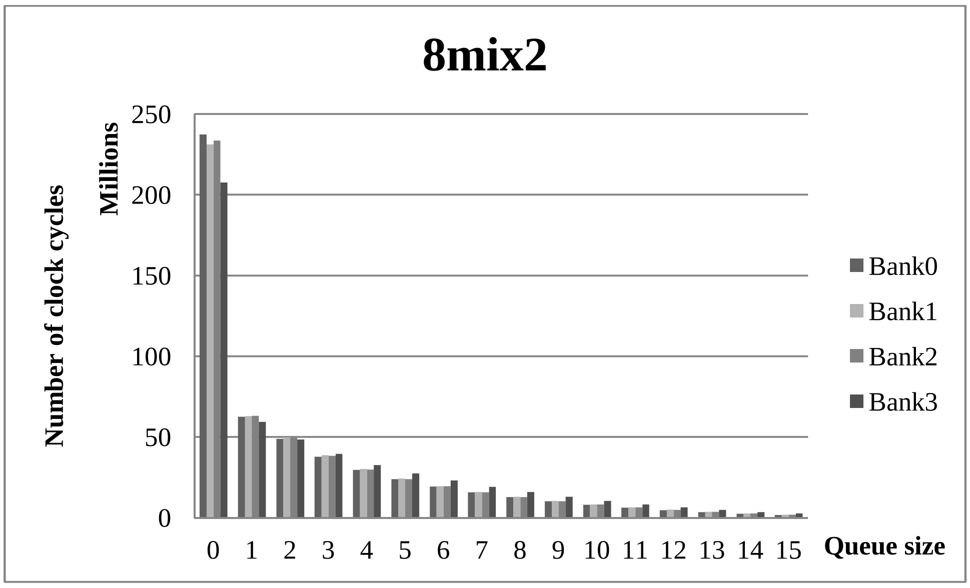


Figure 5.11: 8mix2 histogram \ 24 entries bank queue

Table 5.12: Memory characteristics \ 24 entries bank queue

Workload	**8mem1**	**8mix1**	**8mix2**
Total number of cycles	836157743	584931450	527992043
Queue free (%)	33.29253772	41.20510455	43.07106495
Queue busy (%)	66.70714846	58.79471441	56.92879964
Queue max (%)	3.427846897	2.616898271	2.577793914
Average of waiting requests	5.74	4.17	4.62

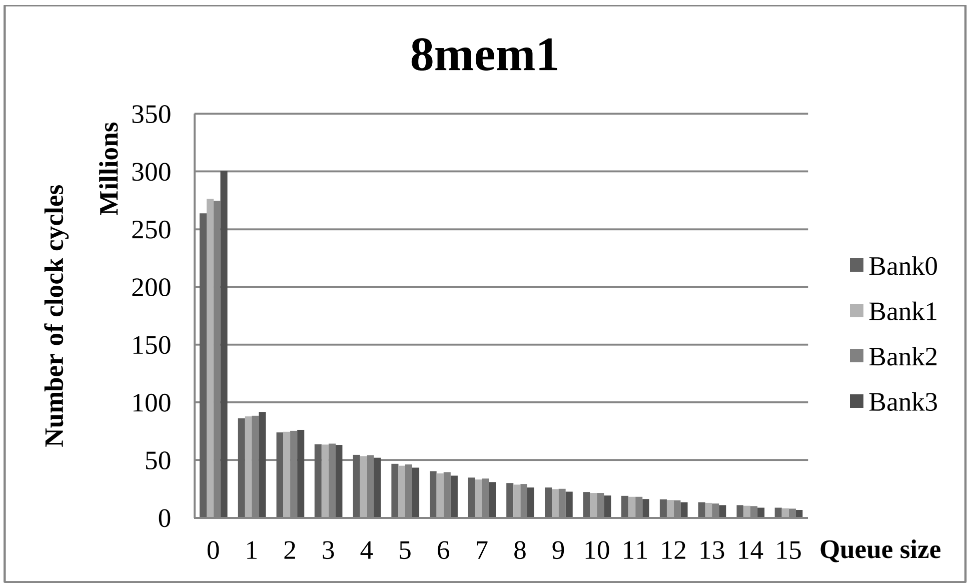


Figure 5.12: 8mem1 histogram \ 32 entries bank queue

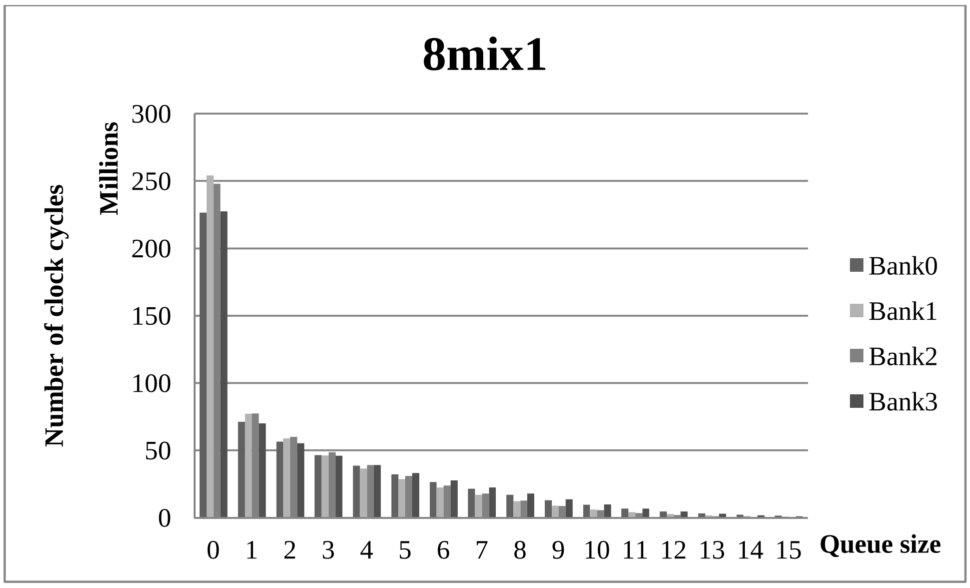


Figure 5.13: 8mix1 histogram \ 32 entries bank queue

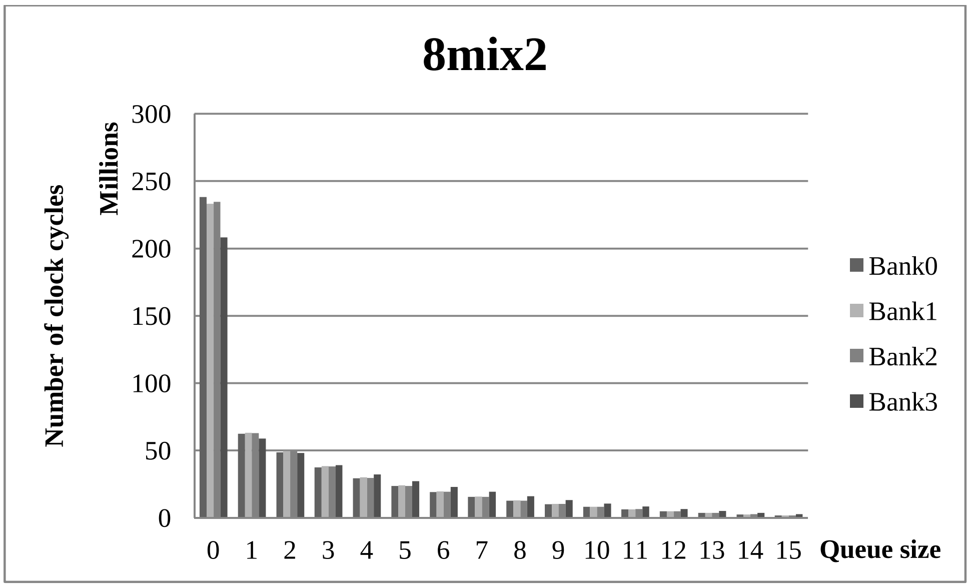


Figure 5.14: 8mix2 histogram \ 32 entries bank queue

Table 5.13: Memory characteristics \ 32 entries bank queue

Workload	**8mem1**	**8mix1**	**8mix2**
Total number of cycles	836731377	581194292	528219397
Queue free (%)	33.31179921	41.12389704	43.26880048
Queue busy (%)	66.68795937	58.8758681	56.73094394
Queue max (%)	3.416528803	2.582664989	2.578615454
Average of waiting requests	5.74	4.16	4.62

Chapter 6. Results

In this chapter we will show the results of TB-LMI against previous memory scheduling algorithms. TB-LMI was compared to FCFS, FR-FCFS, FR-LREQ, FR-FLRMR, FR-Modified_ROB, FR-FIQMR, and TCM. We will show the results of memory scheduling algorithms in case of 4-core/8-core processors. Figures will show the weighted speedup and ANTT which reflects processor’s performance, and maximum slowdown which reflects processor’s fairness among running applications/threads. FCFS is the baseline unless otherwise mentioned.

6.1 Sensitivity analysis

Sensitivity analysis was performed on SQ and row buffer hit requests to measure their effect on performance and fairness in case of TB-LMI. As shown in figure 6.1, SQ was set to different values in case of 8-core processor and memory bank queue size has 8 entries. X-axis represents the weighted speedup and Y-axis represents the maximum slowdown. SQ with value at the bottom right of the figure has the highest speedup and the best fairness. It is observed that TB-LMI with $1\ \text{Kcycles} < SQ < 10\ Mcycles$ has the highest weighted speedup (best performance) and the lowest maximum slowdown (highest fairness).

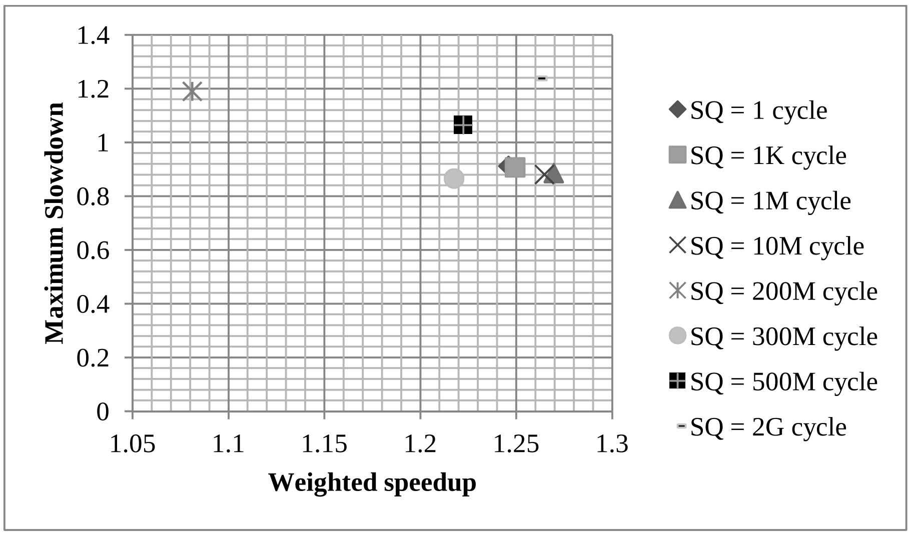


Figure 6.1: Weighted speedup versus Maximum slowdown 8-core \ 8 entries memory bank queue (SQ sensitivity analysis)

In addition, sensitivity analysis was performed to show the effect of row buffer hit requests on performance and fairness in case of TB-LMI. A First-Ready Threshold (FRT) is defined to limit successive row buffer hits that can occur in TB-LMI (level 1) before it switches to level 2. FRT is applied per memory bank. FRT = 0 means that there is no search for row buffer hit requests. FRT = 2 means that TB-LMI can perform only 2 successive row buffer hits before it switches to level 2. Figures 6.2 and 6.3 show the row buffer hit sensitivity analysis in case of 8-core processors when memory bank queue has 8 entries and 32 entries respectively. It is observed that TB-LMI improves both processor's performance and fairness as FRT increases. TB-LMI with FRT ≥ 2 has the highest performance and the best fairness. TB-LMI with FRT = ∞ has improved performance by 5.5% and 5.2% than TB-LMI with FRT = 0 when 8 entries and 32 entries memory bank queues were used respectively. Also, TB-LMI with FRT ≥ 2 has improved fairness by 6% and 10% than TB-LMI with FRT = 0 when 8 entries and 32 entries memory bank queue were used respectively. TB-LMI design was chosen to use FRT = ∞ as it does not need extra hardware implementation. TB-LMI with FRT ≤ ∞ will require an extra register implementation per memory bank. An extra register counts the number of successive row buffer hits per memory bank. The extra register is reset once its value reaches the FRT and gives the highest priorities to row buffer conflict requests in the memory bank queue. In addition, TB-LMI with FRT = ∞ decreases the processing in any multi-core system as there is no need to keep track of the number of successive row buffer hits for any memory bank.

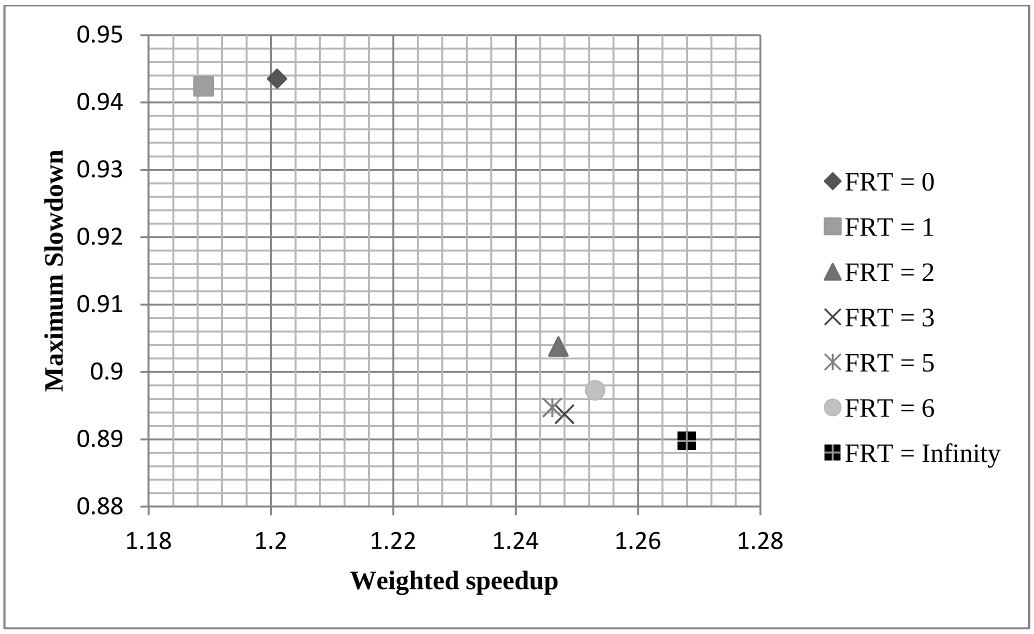


Figure 6.2: Weighted speedup versus Maximum slowdown 8-core \ 8 entries memory bank queue (FR sensitivity analysis)

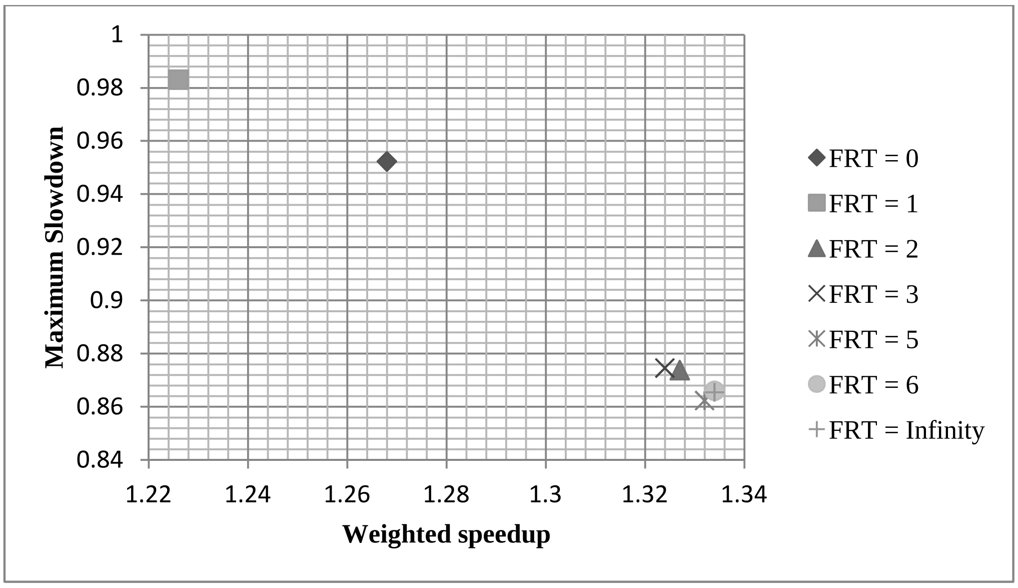


Figure 6.3: Weighted speedup versus Maximum slowdown 8-core \ 32 entries memory bank queue (FR sensitivity analysis)

6.2 Main memory effective latency

Main memory effective latency of TB-LMI, TCM, FR-LREQ and FCFS were calculated. Calculations were performed on 8-core processor when a memory bank queue has 8 entries. Results show that the main memory effective latency is the same among different workloads. Figure 6.4, Figure 6.5, and Figure 6.6 show the main memory effective latency characteristics of 8mem1, 8mix1, and 8mix2 respectively. Table 6.1 summarizes average value, variance, and standard deviation of main memory effective latency of different scheduling after dividing it with the main memory effective latency in case of FCFS (baseline main memory effective latency is FCFS).

It is observed that TB-LMI and TCM have the same average, variance, and standard deviation in case of 8mem1 but TB-LMI aims to improve fairness at the expense of performance. In case of 8mix1, TB-LMI has increased the average, variance, and standard deviation memory latency by 3.9%, 9.3%, and 4.6% respectively. Although TB-LMI has increased the average, variance, and standard deviation memory latency in 8mix1 but TB-LMI technique used in prioritizing waiting requests from applications/threads is better than TCM. In case of 8mix2, TB-LMI has decreased the average, variance, and standard deviation memory latency by 3.5%, 6.6%, and 3.2%.

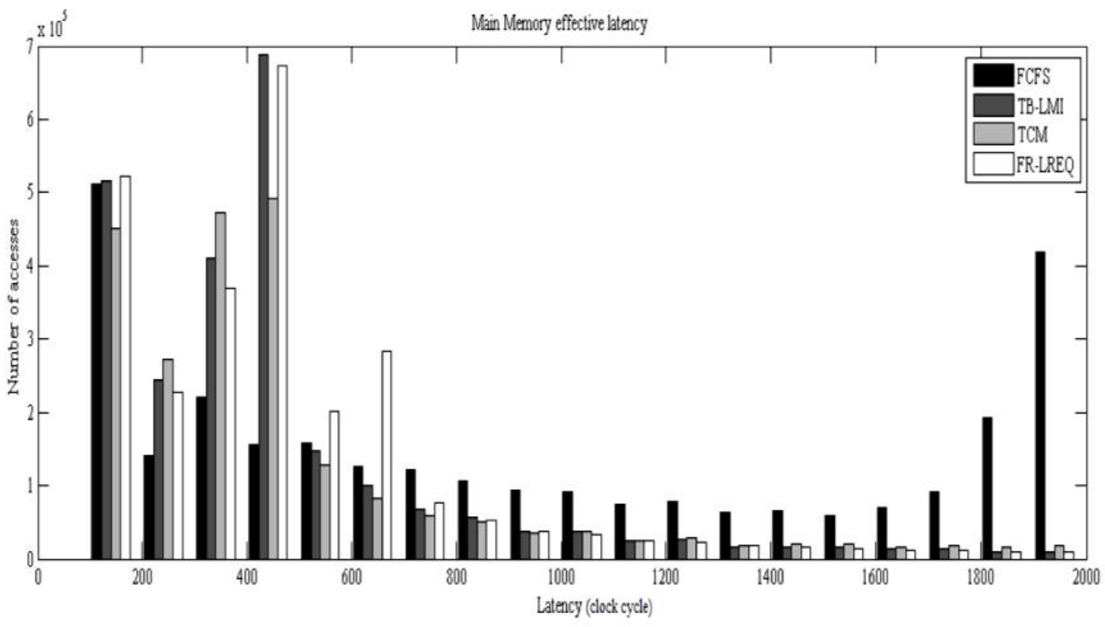


Figure 6.4: Main Memory Effective Latency Histogram for 8mem1

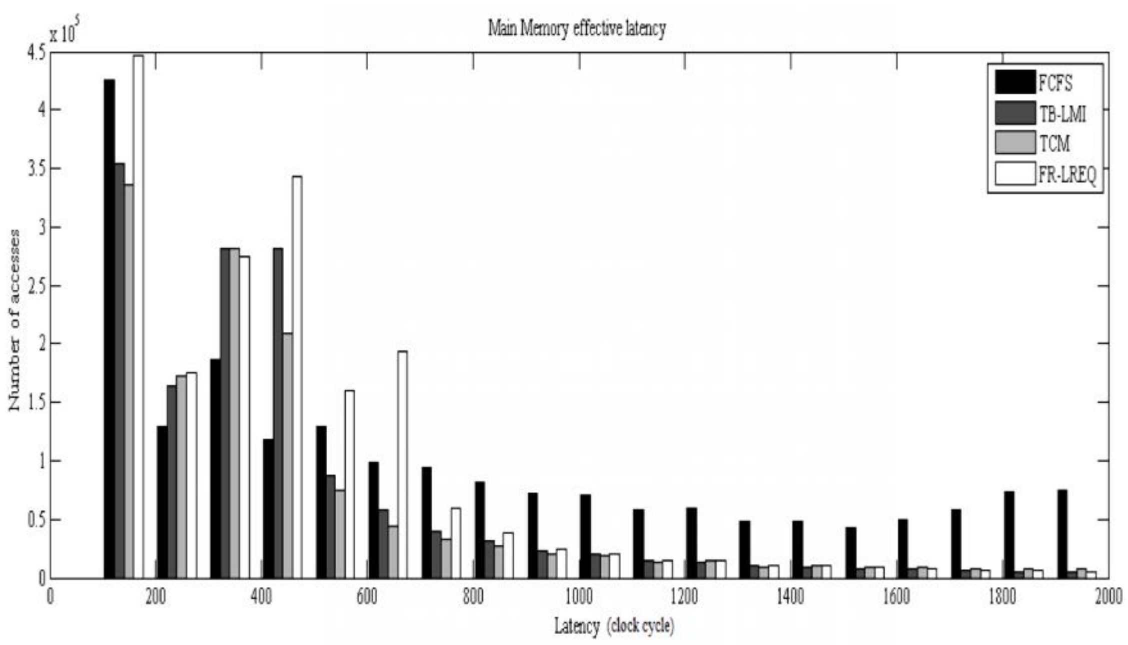


Figure 6.5: Main Memory Effective Latency Histogram for 8mix1

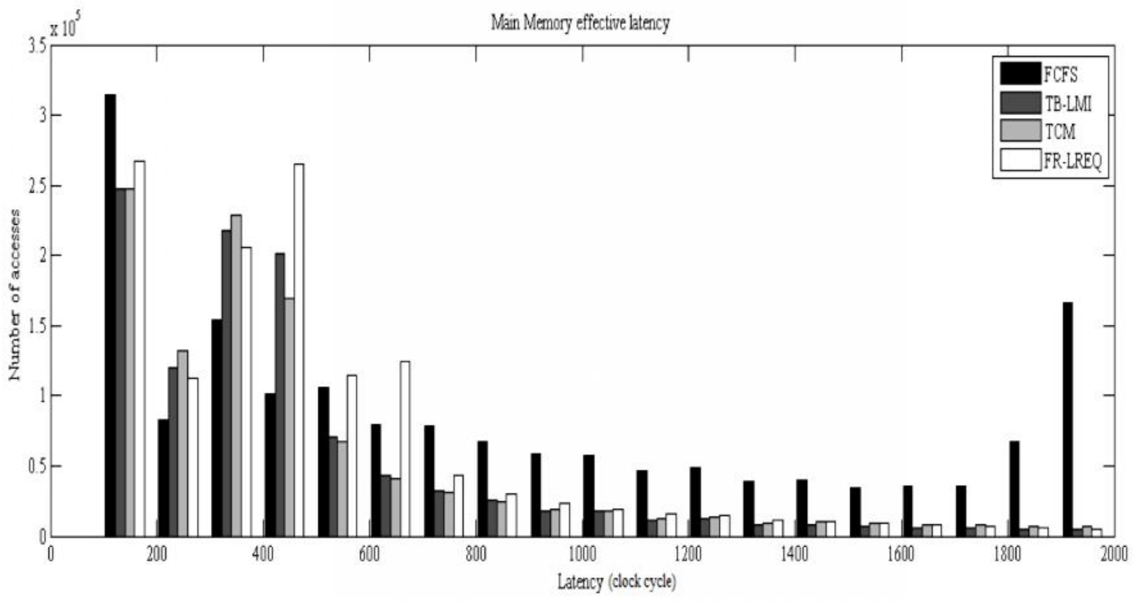


Figure 6.6: Main Memory Effective Latency Histogram for 8mix2

Table 6.1: Main memory effective latency characteristics \ Baseline scheduling algorithm is FCFS

	8mem1			
	FCFS	**TB-LMI**	**TCM**	**FR-LREQ**
Average	1	0.96363207	0.95079081	1.00534075
Variance	1	0.90640126	0.90482454	1.02078556
Standard deviation	1	0.9520511	0.95122267	1.01033935
	8mix1			
	FCFS	**TB-LMI**	**TCM**	**FR-LREQ**
Average	1	0.83185046	0.80077922	1.12936673
Variance	1	0.7007208	0.64532885	1.27857006
Standard deviation	1	0.83709068	0.80332363	1.13073872
	8mix2			
	FCFS	**TB-LMI**	**TCM**	**FR-LREQ**
Average	1	0.70086173	0.72578495	0.92127878
Variance	1	0.49760009	0.53076991	0.84989313
Standard deviation	1	0.70540775	0.72853958	0.92189649

6.3 Performance analysis

We have used TB-LMI when FRT = ∞ in our analysis as it does not need extra hardware implementation or CPU processing (discussed in section 1.1). In addition, we used memory bank queue with 8, 16, 24, and 32 in our performance analysis. It is expected that increasing memory bank queue size than 32 entries will not show a significant performance or fairness improvement. As discussed in section 6.1 in case of 8-core system, TB-LMI when FRT = ∞ has improved performance and fairness by 3.9% and 3% respectively when the memory bank queue size is increased from 8 entries to 32 entries. This concludes that TB-LMI when FRT = ∞ shows a small improvement in performance and fairness when memory bank queue size is increased from 8 entries to 16 entries and 24 entries (will be discussed in this chapter).

6.3.1 4-core

TCM, FR-LREQ, and FR-FLRMR showed competitive results against TB-LMI. Other scheduling algorithms showed approximately the same performance and fairness improvements. We will discuss and state memory scheduling algorithms that showed competitive results.

Figure 6.7, Figure 6.8, and Figure 6.9 show the weighted speedup, ANTT, and maximum slowdown when the memory bank queue has 8 entries. It is observed that TB-LMI, FR-LREQ, and TCM achieve maximum performance and fairness in case of memory intensive workloads (4mem1 and 4mem2). TB-LMI compared to FCFS achieves a maximum weighted speedup improvement increase by 8.22%, ANTT decrease by 4%, and a decrease in maximum slowdown by 19.7% in case of 4mem1. FR-LREQ compared to FCFS achieves a maximum weighted speed increase by 8%, ANTT decrease by 5%, and a decrease in maximum slowdown 17% in case of 8mem2. TCM compared to FCFS achieves a maximum weighted speedup improvement increase by 5.9%, ANTT decrease by 4.5%, and a decrease in maximum slowdown by 16.5% in case of 4mix5. TCM achieves the same results approximately in case of 4mem1.

We can notice that TB-LMI is only in the second place after FR-LREQ in case of 4mem2. In case of 4mem2, TB-LMI compared to FCFS increases weighted speedup by 6.2%, ANTT decreases by 5%, and maximum slowdown decrease by 7.9%. In case 4mem1, FR-LREQ compared to FCFS increases the weighted speedup by 6%, ANTT decrease by 4.7%, and a decrease in maximum slowdown by 8%. 4mem2 has 462.libquantum benchmark. 462.libquantum is a memory intensive benchmark and has a high row buffer hit rate [26, 27, 28]. FR-LREQ targets memory requests from running applications/threads that has the minimum number of waiting requests in memory bank queue. FR-LREQ assigns requests 462.libquantum with the highest priority as it is expected to have the minimum waiting requests in the memory bank queue. This results in releasing more requests from 462.libquantum which in return improve processor's performance. Table 6.2 shows IPC for each benchmark in 4mem2. It is observed that FR-LREQ has increased 462.libquantum IPC from 0.8331 in case of TB-LMI to 1.13. In return FR-LREQ has decreased IPC for benchmarks 436.cactusADM and 410.bwaves. IPC in case of TB-LMI are closer to each other than FR-LREQ. TB-LMI targets fairness firstly then it tries to improve the performance which will be discovered in 8-core performance analysis.

TABLE 6.2: Detailed IPC of 4mem2 in case of TB-LMI and FR-LREQ

Benchmark	IPC achieved in TB-LMI	IPC achieved in FR-LREQ
436.cactusADM	0.9674	0.8354
410.bwaves	0.4971	0.4089
401.bzip2	0.7868	0.8555
462.libquantum	0.8331	1.13

At last, FR-FLRMR compared to FCFS achieves a maximum weighted speedup by 3% increase, a decrease in ANTT by 3%, and a decrease in maximum slowdown by 2.7%. FR-FLRMR does not show any decrease in performance or fairness on any of the workloads. At least FR-FLRMR works as FCFS.

On average, TB-LMI is the best memory scheduling algorithm. FR-LREQ is in the second place. TCM is in the third place. Figure 6.10 shows the average values of weighted speedup versus maximum slowdown when memory bank queue has 8 entries. The memory scheduling algorithm at the bottom right of figure 6.4 has the maximum performance and the best fairness which is TB-LMI.TB-LMI compared to TCM has a better speedup by 4%, less ANTT by 4.6%, and less maximum slowdown by 5.55%. FR-LREQ compared to TCM has better speedup by 2.8%, less ANTT by 4.6%, and less maximum slowdown by 5.5% than TCM. FR-FLRMR compared to TCM has the same speed up, less ANTT by 2.33%, and same maximum slowdown when it is compared to TCM.

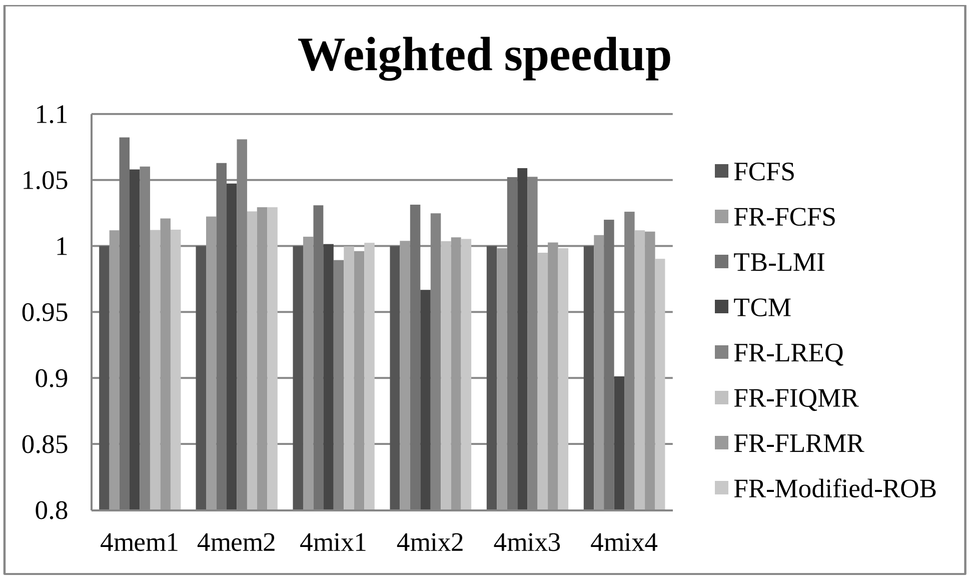


Figure 6.7: Weighted speedup 4-core \ 8 entries memory bank queue

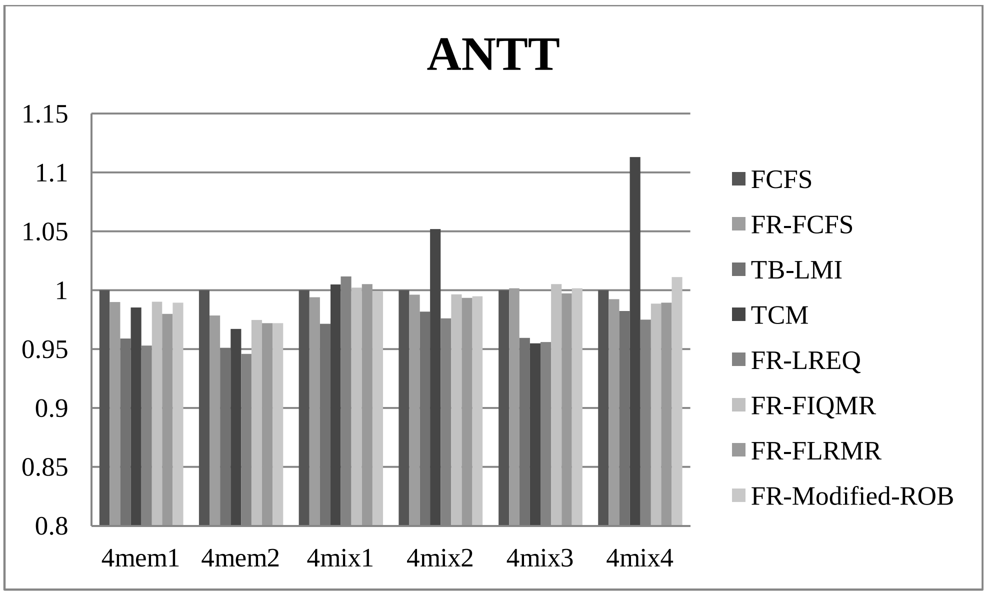


Figure 6.8: ANTT 4-core \ 8 entries memory bank queue

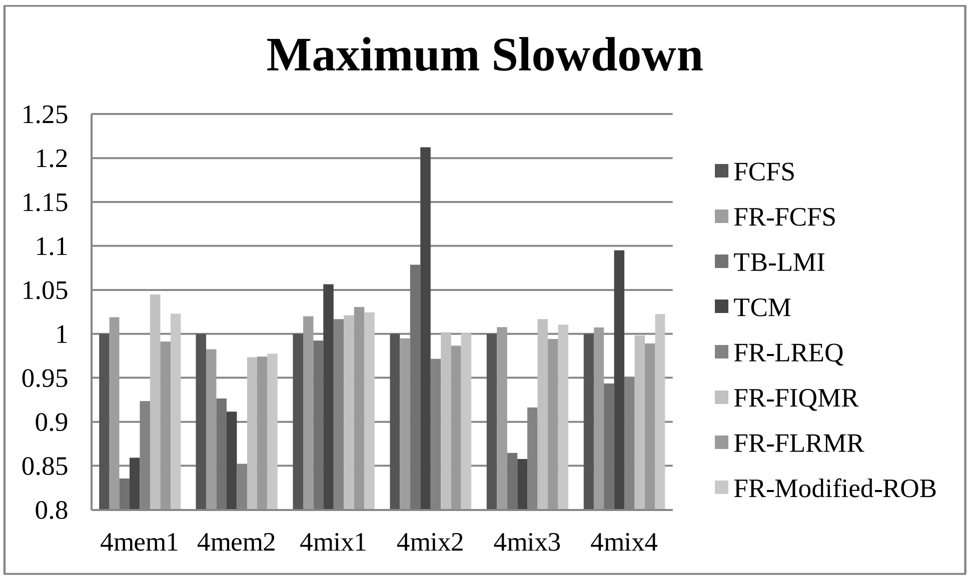


Figure 6.9: Maximum slowdown 4-core \ 8 entries memory bank queue

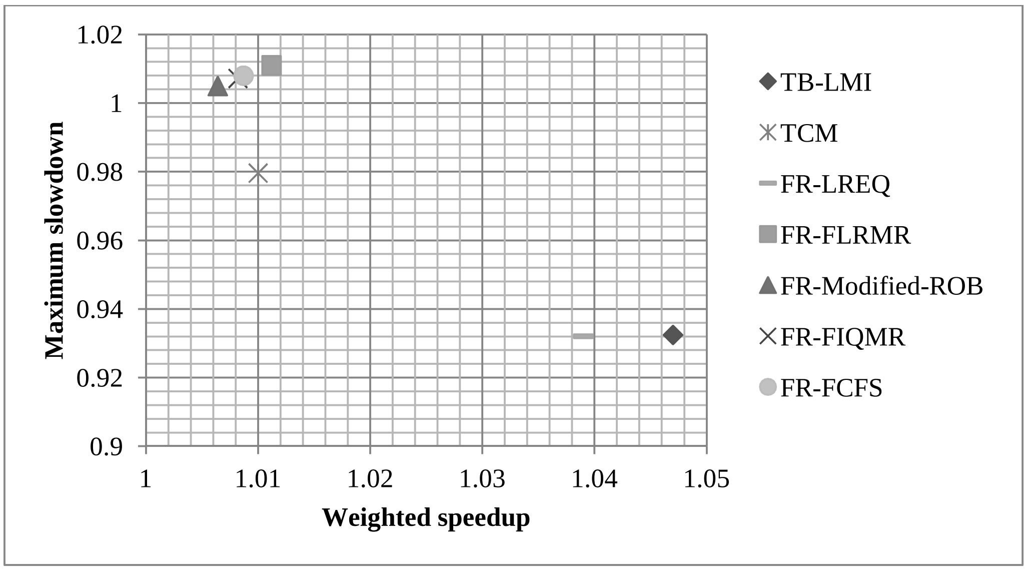


Figure 6.10: Weighted speedup versus Maximum slowdown 4-core \ 8 entries memory bank queue

Figure 6.11, Figure 6.12, and Figure 6.13 show the weighted speedup, ANTT, and maximum slowdown when the memory bank queue has 16 entries. Increasing the main memory queue size from 8 entries to 16 entries does not show any performance or fairness improvement. Also, it is observed that TB-LMI, FR-LREQ, and TCM achieve the maximum performance and fairness in case of 100% memory intensive workloads (4mem1 and 4mem2). Again TB-LMI and FR-LREQ have the best performance, ANTT, and fairness improvements across all workloads. FR-FLRMR is in the second place. TCM is in the third place. TB-LMI compared to FCFS achieves a maximum weighted speedup by 8.8%, decrease in ANTT by 4.6%, and a decrease in maximum slowdown by 20.7% in case of 4mem1. We can notice that TB-LMI is only in the second place after FR-LREQ in case of 4mem2. At last, FR-FLRMR does not show any decrease in performance or fairness on any of the workloads

We can conclude that there is neither performance nor fairness improvement when the memory bank queue is increased from 8 entries to 16 entries. It is expected that neither performance nor fairness improvement will be achieved when the memory bank queue size is increased to 24 entries or 32 entries. Also, the maximum performance and fairness improvement from TB-LMI will be achieved only in case of 100% memory intensive workloads. In case of 100% memory intensive workloads, the average number of waiting requests in memory bank queue is higher than 50% memory intensive workloads. This increases the working time of TB-LMI. Also, we can notice that TCM compared to FCFS showed a decrease in performance and fairness in some of the workloads which reflects the disadvantage of TCM. TCM is not one of the best memory scheduling algorithms for low core count processors (2-core and 4-core).

Figures 6.14 to Figure 6.19 show the results of memory scheduling algorithms in case of 24 entries and 32 entries memory bank queue. Memory scheduling algorithms achieve the same results that were discussed in 8 entries and 16 entries memory bank queue.

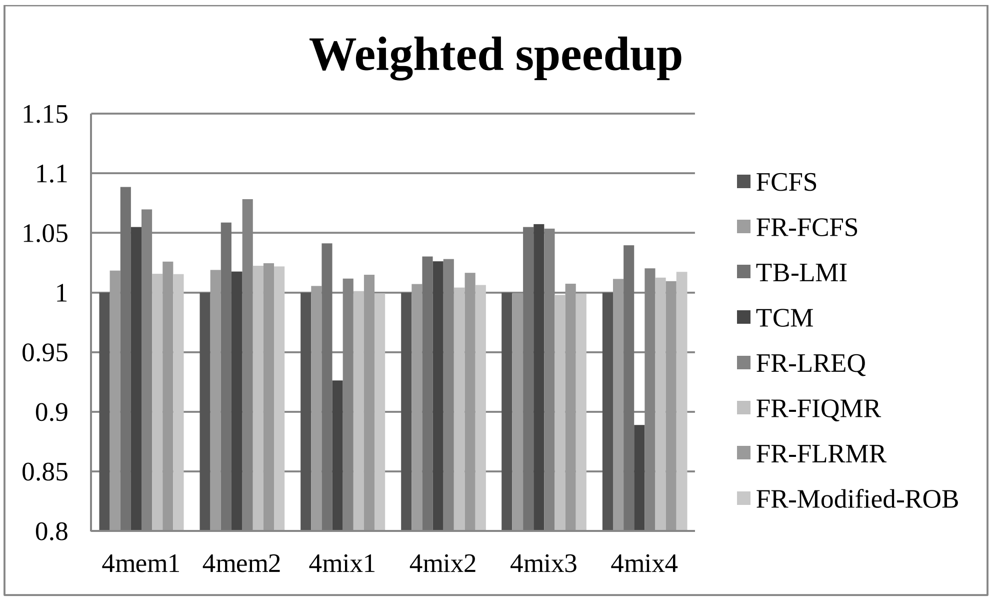


Figure 6.11: Weighted speedup 4-core \ 16 entries memory bank queue

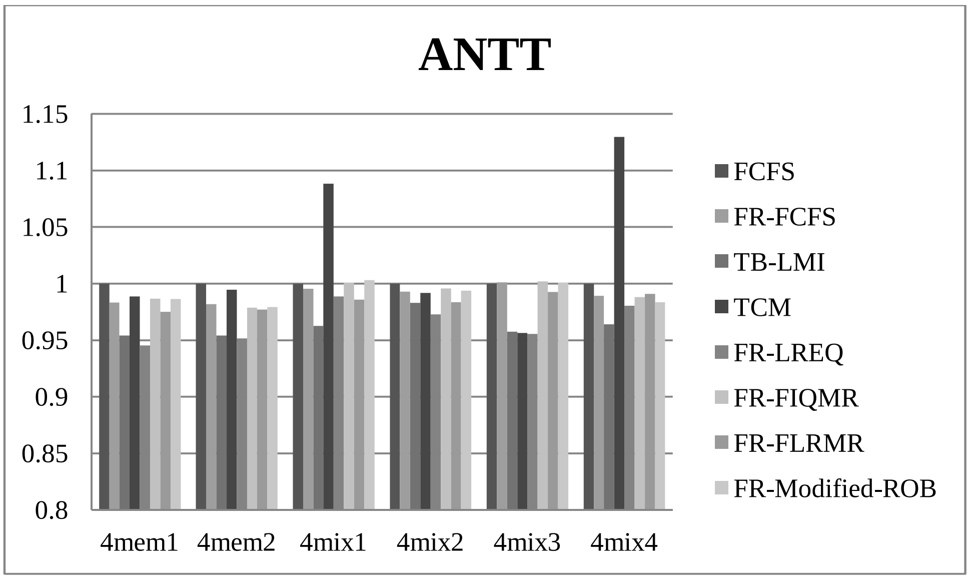


Figure 6.12: ANTT 4-core \ 16 entries memory bank queue

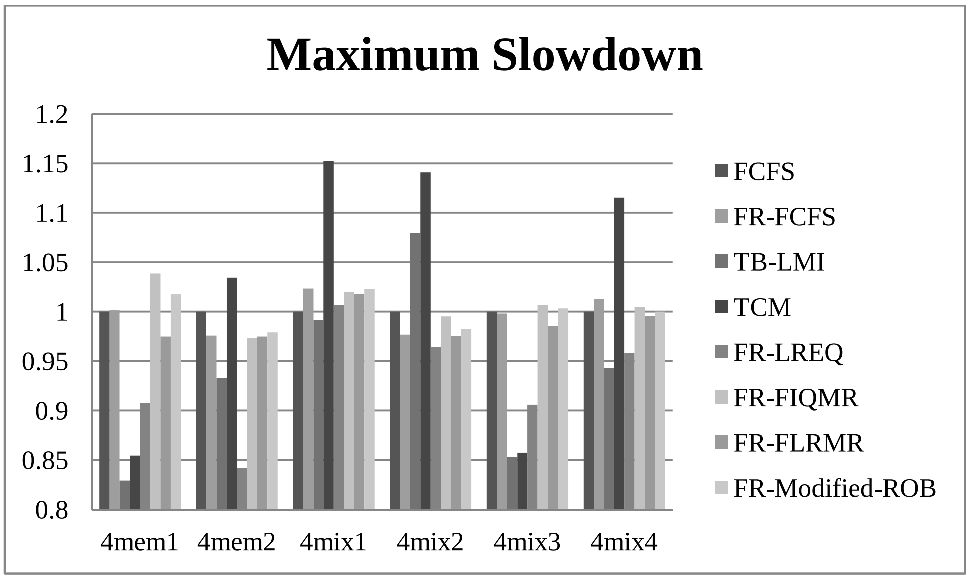


Figure 6.13: Maximum slowdown 4-core \ 16 entries memory bank queue

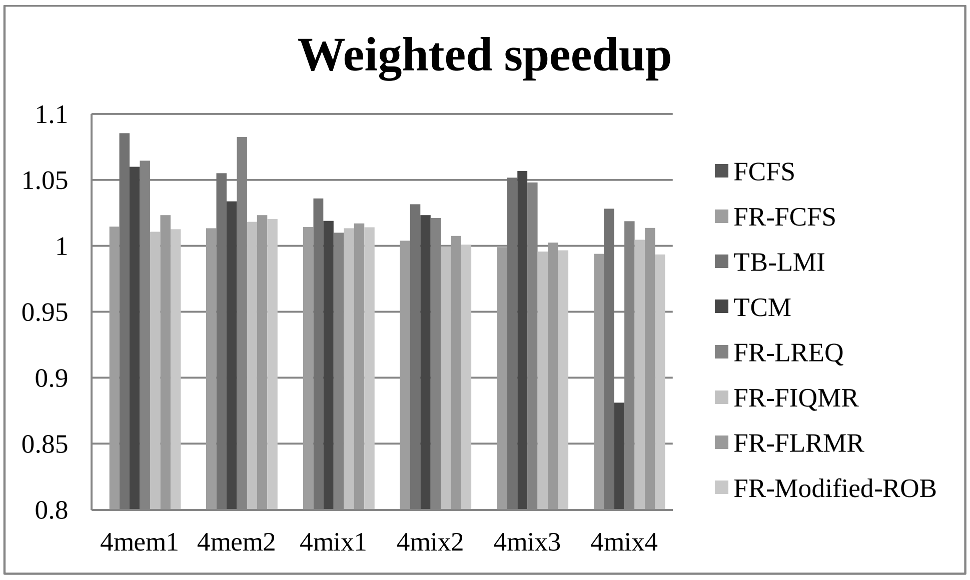


Figure 6.14: Weighted speedup 4-core \ 24 entries memory bank queue

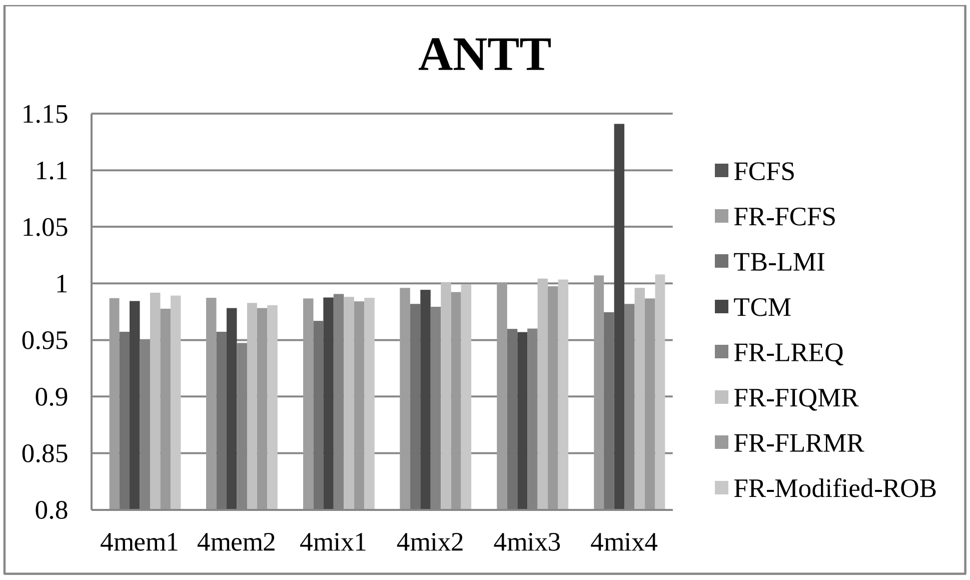


Figure 6.15: ANTT 4-core \ 24 entries memory bank queue

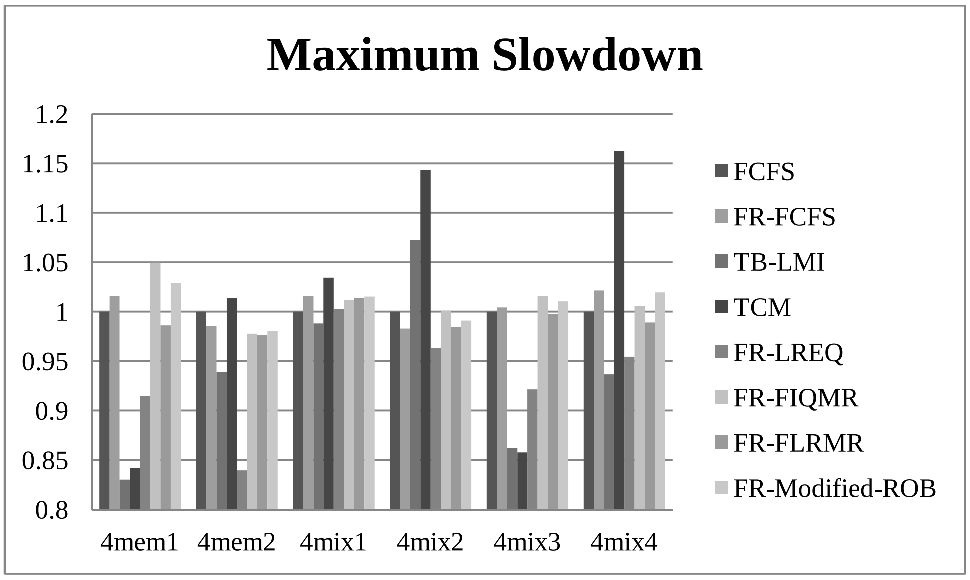


Figure 6.16: Maximum slowdown 4-core \ 24 entries memory bank queue

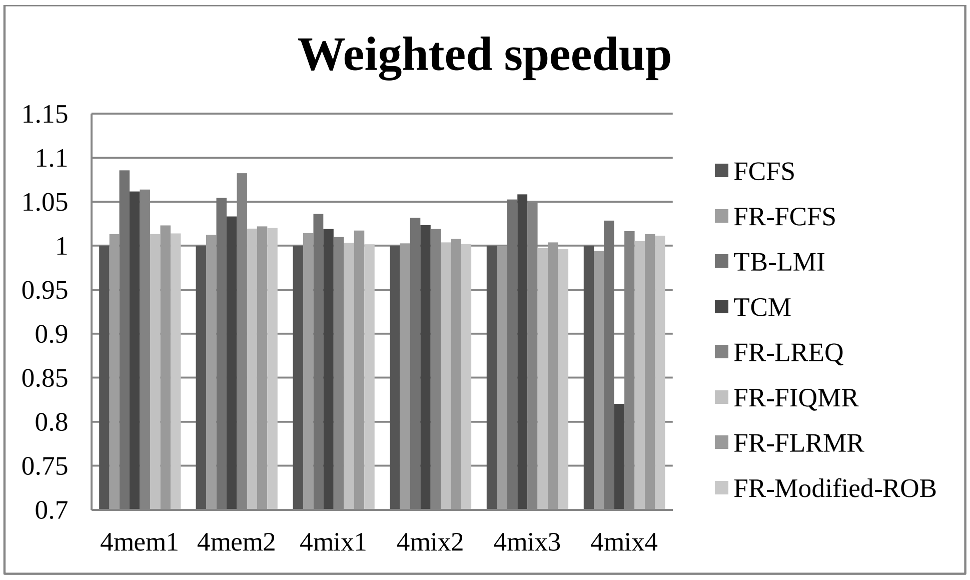


Figure 6.17: Weighted speedup 4-core \ 32 entries memory bank queue

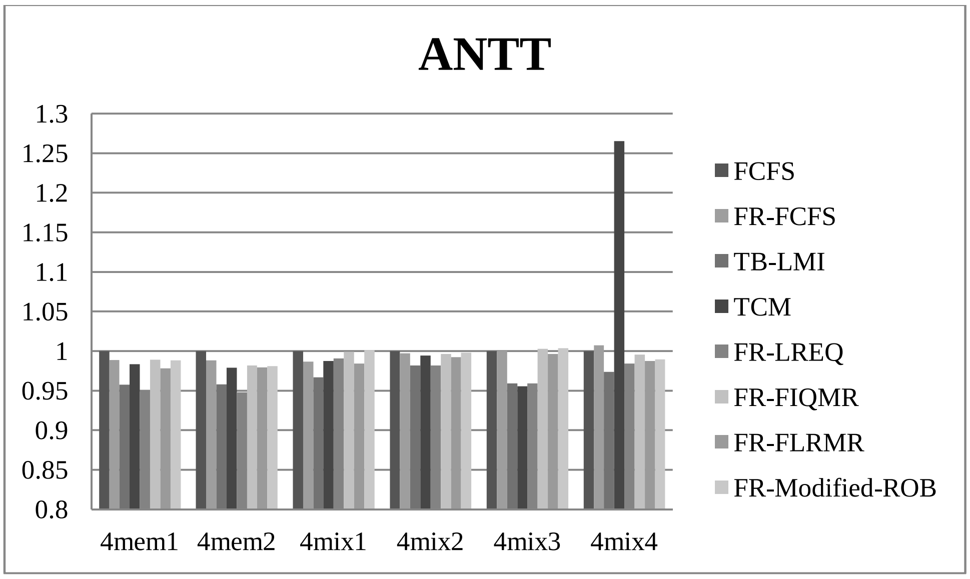


Figure 6.18: ANTT 4-core \ 32 entries memory bank queue

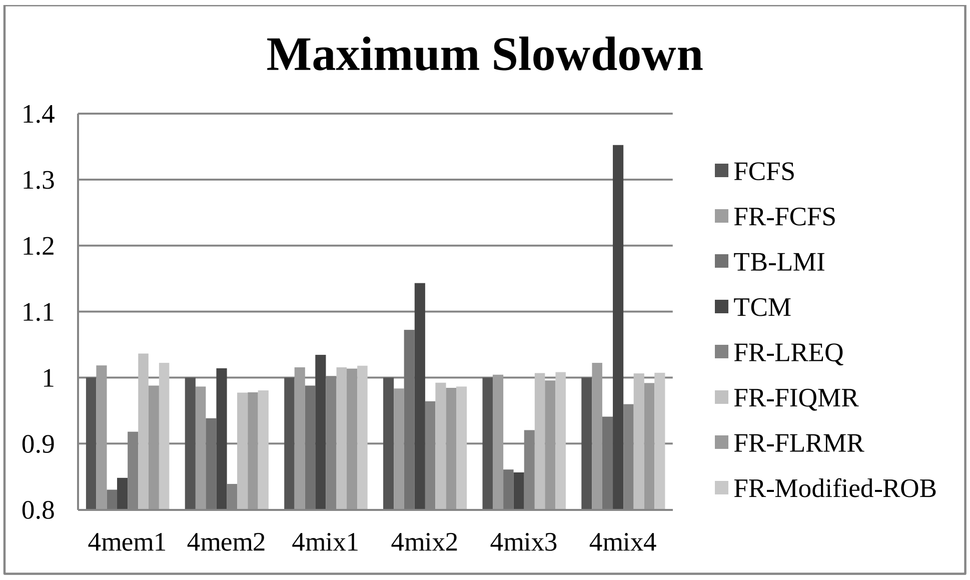


Figure 6.19: Maximum slowdown 4-core \ 32 entries memory bank queue

6.3.2 8-core

TCM, FR-LREQ, and FR-FLRMR showed competitive results versus TB-LMI. Our discussion focuses on competitive memory scheduling algorithms.

Figure 6.20, Figure 6.21, and Figure 6.22 show the weighted speedup, ANTT, and maximum slowdown when the memory bank queue has 8 entries. TB-LMI compared to TCM has a maximum weighted speedup by 6%, less ANTT by 3.3%, and less maximum slowdown by 42% in case of 8mix1. TB-LMI compared to TCM shows a decrease in weighted speedup by 2.4%, increase in ANTT by 4.7% and less maximum slowdown by 5.6% than TCM in case of 8mem1. In case of 8mix2, TB-LMI compared to TCM shows an increase in weighted speedup by 8.8%, ANTT was decreased by 2.6%, but maximum slowdown was increased by 16.6%. These results confirm our conclusion in 4-core processors where TB-LMI targets fairness in the first place and then the performance. TB-LMI targets fairness in case of 100% memory intensive benchmarks as in 8mem1. TB-LMI targets performance and fairness in case of 50% memory intensive workloads that contains high memory intensive benchmarks (429.mcf, and 462.libquantum) as in 8mix1 (refer to appendix A for more details). TB-LMI targets performance in case of 50% memory intensive workloads that has no high memory intensive benchmarks as in 8mix2.

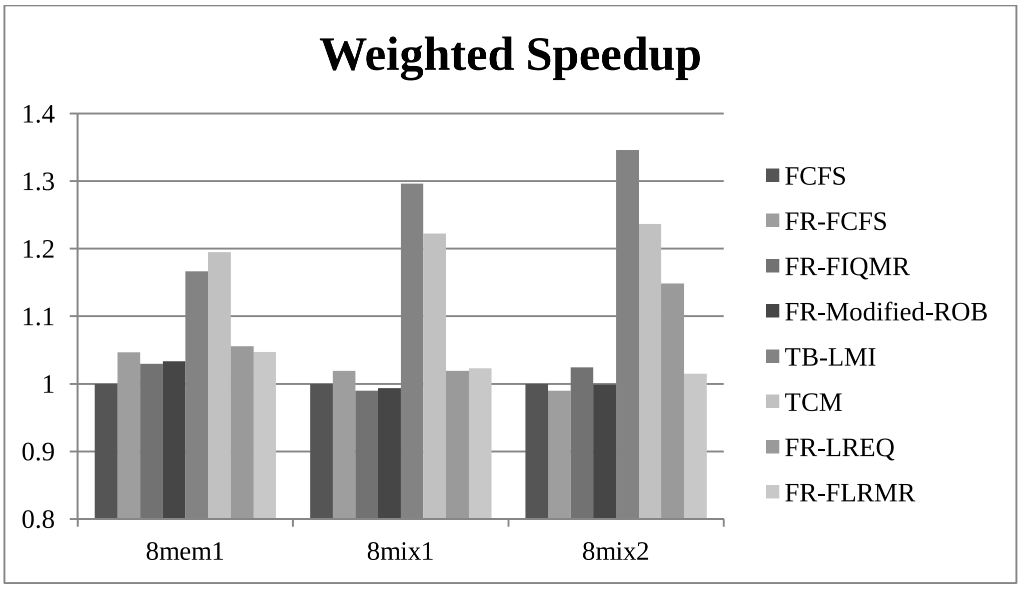


Figure 6.20: Weighted speedup 8-core \ 8 entries memory bank queue

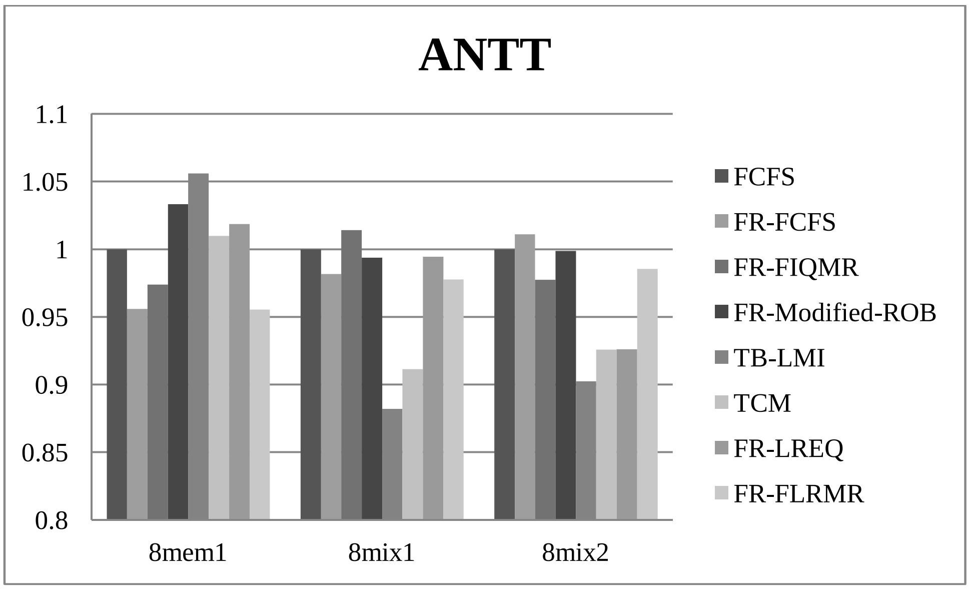


Figure 6.21: ANTT 8-core \ 8 entries memory bank queue

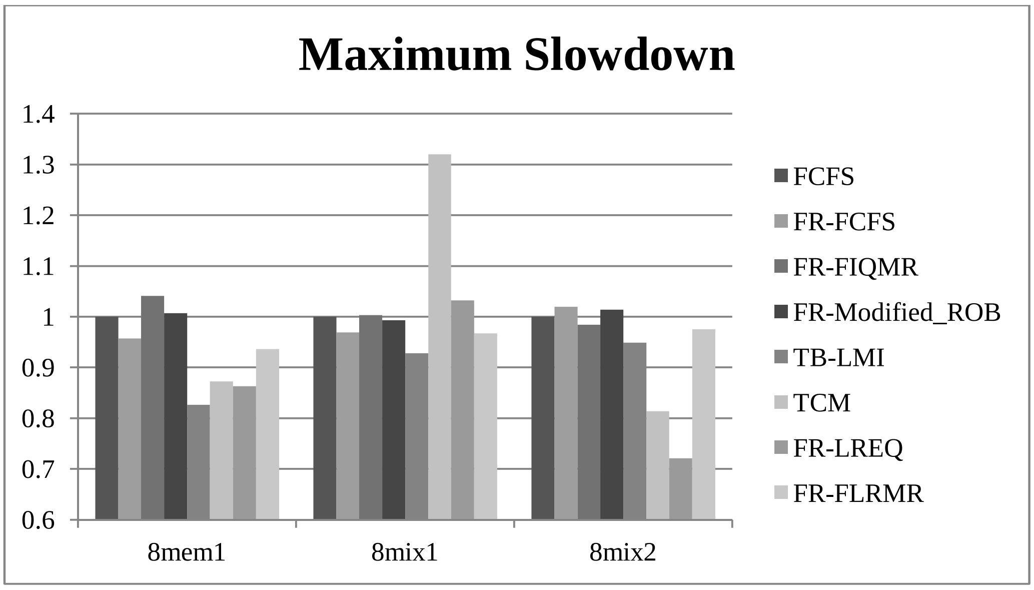


Figure 6.22: Maximum slowdown 8-core \ 8 entries memory bank queue

Figure 6.23, Figure 6.24, and Figure 6.25 show the weighted speedup, ANTT, and maximum slowdown when the memory bank queue has 16 entries. TB-LMI has the best performance in 8mix1 and 8mix2, but it showed a slight decrease compared to TCM in performance by 1.4% in case of 8mem1. TB-LMI compared to TCM shows a maximum weighted speedup by 8%, decrease in ANTT by 6.68%, and a decrease in maximum slowdown by 27% in case of 8mix1. TB-LMI compared to TCM shows an improvement in processor’s performance but fairness is decreased in case of 8mix2. TB-LMI compared to TCM shows an increase in maximum slowdown by 5.87% in case of 8mix2. This confirms our conclusion that was discussed in case memory bank queue has 8 entries where TB-LMI targets performance in case of 50% memory intensive workloads that has no high memory intensive benchmarks as in 8mix2.

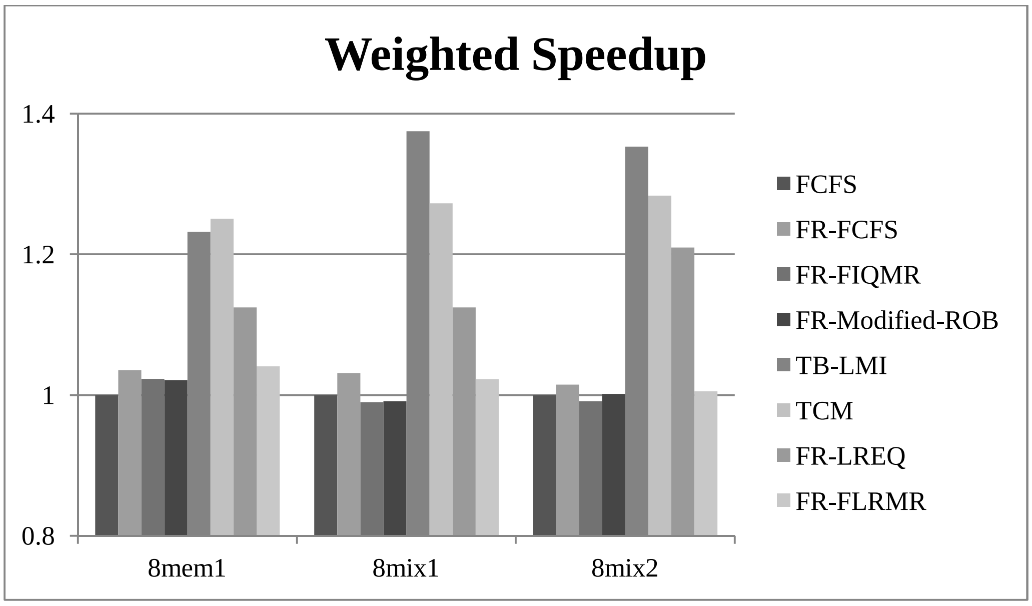


Figure 6.23: Weighted speedup 8-core \ 16 entries memory bank queue

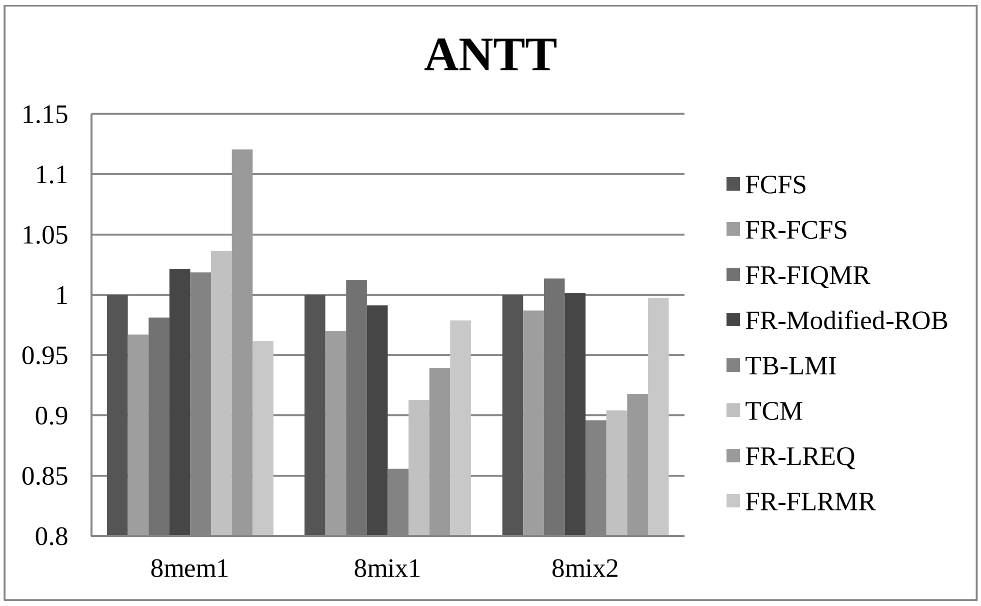


Figure 6.24: ANTT 8-core \ 16 entries memory bank queue

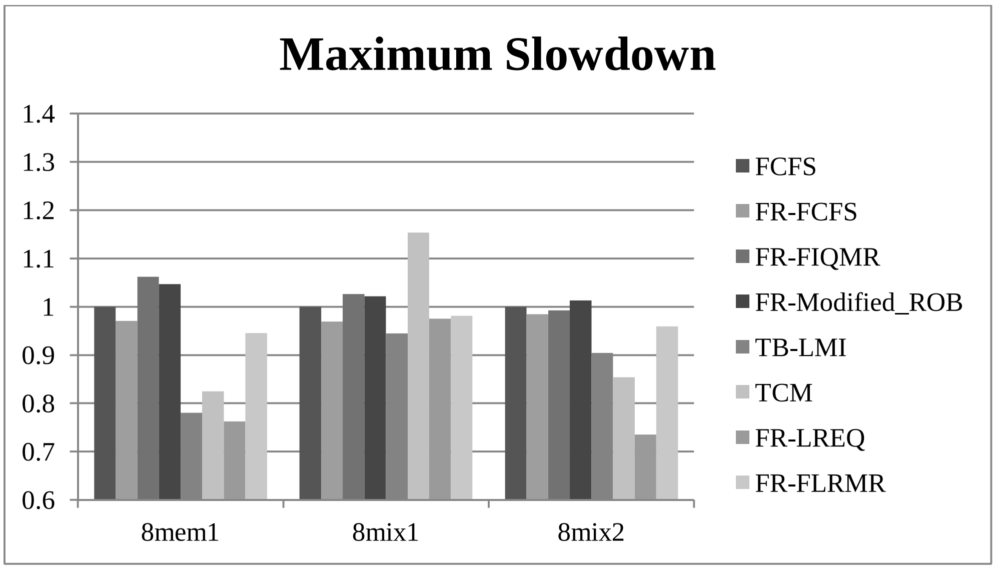


Figure 6.25: Maximum slowdown 8-core \ 16 entries memory bank queue

Figures 6.26 to Figure 6.31 show the results of memory scheduling algorithms in case of 24 entries and 32 entries memory bank queue. Memory scheduling algorithms achieve the same results that were discussed in 8 entries and 16 entries memory bank queue.

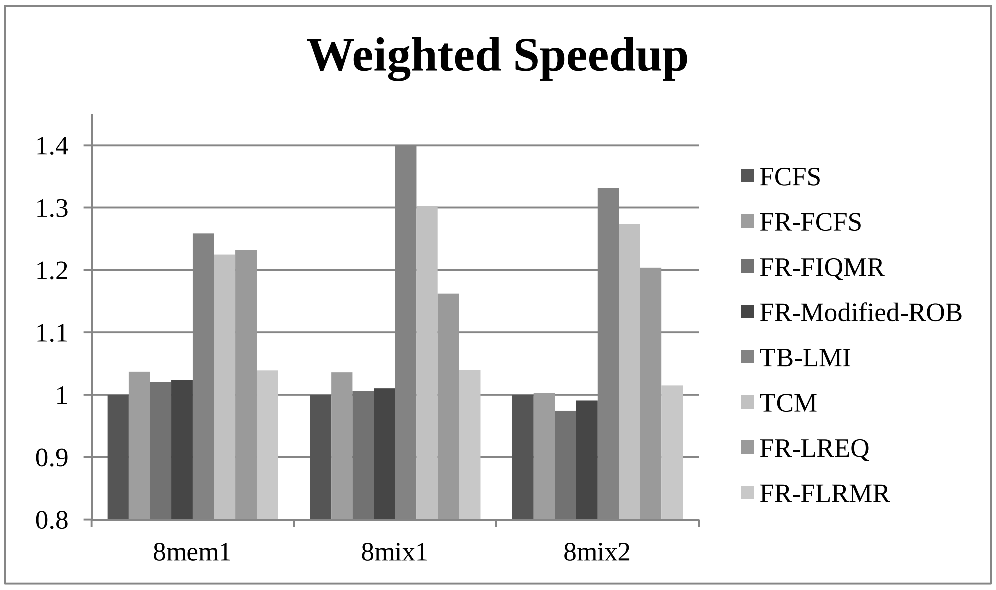


Figure 6.26: Weighted speedup 8-core \ 24 entries memory bank queue

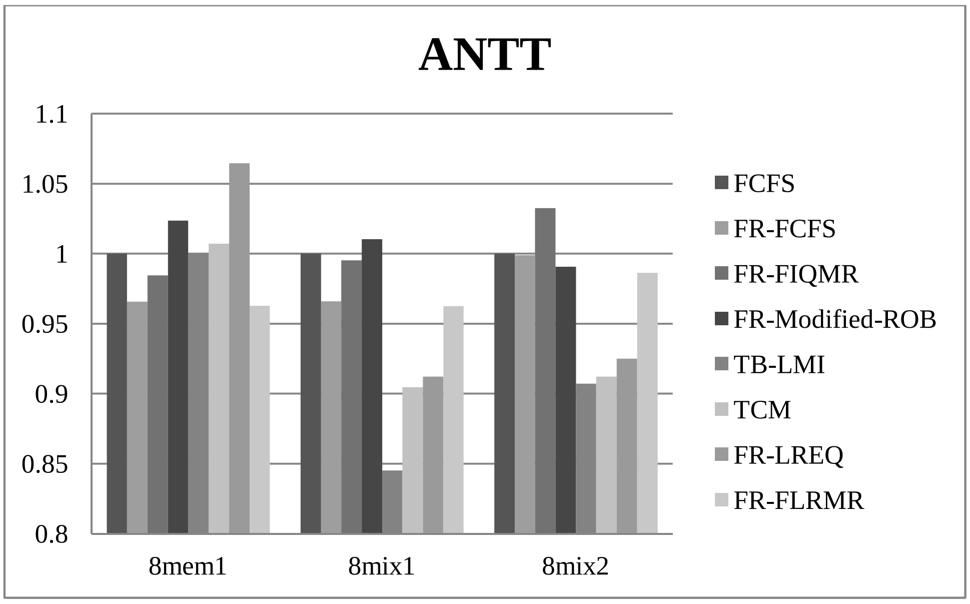


Figure 6.27: ANTT 8-core \ 24 entries memory bank queue

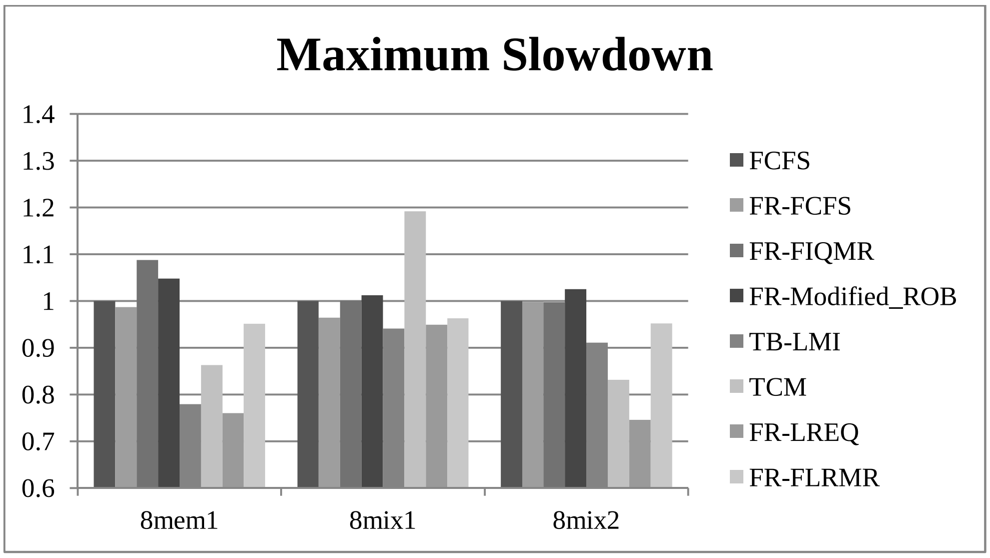


Figure 6.28: Maximum slowdown 8-core \ 24 entries memory bank queue

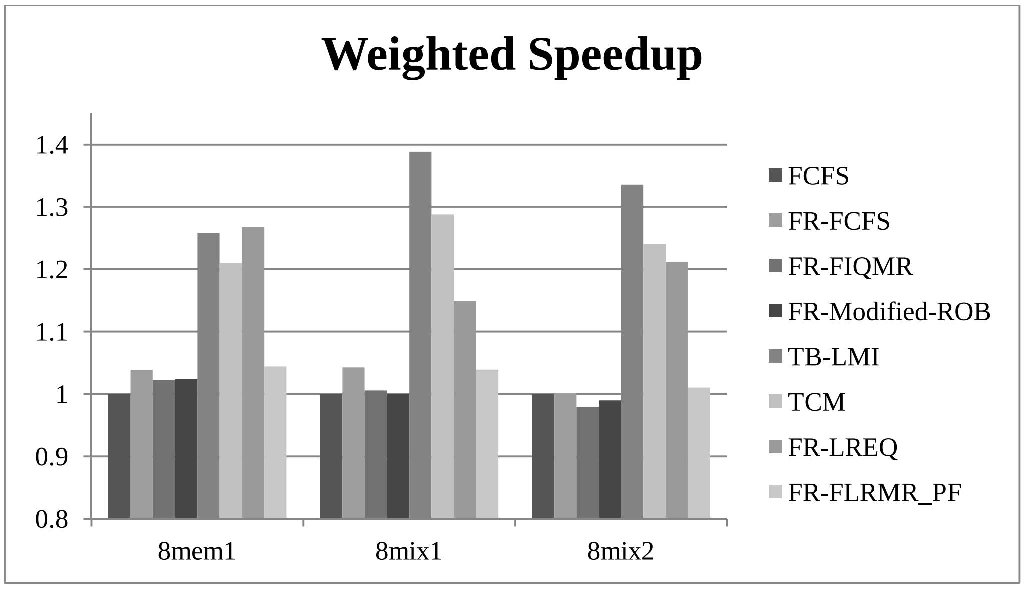


Figure 6.29: Weighted speedup 8-core \ 32 entries memory bank queue

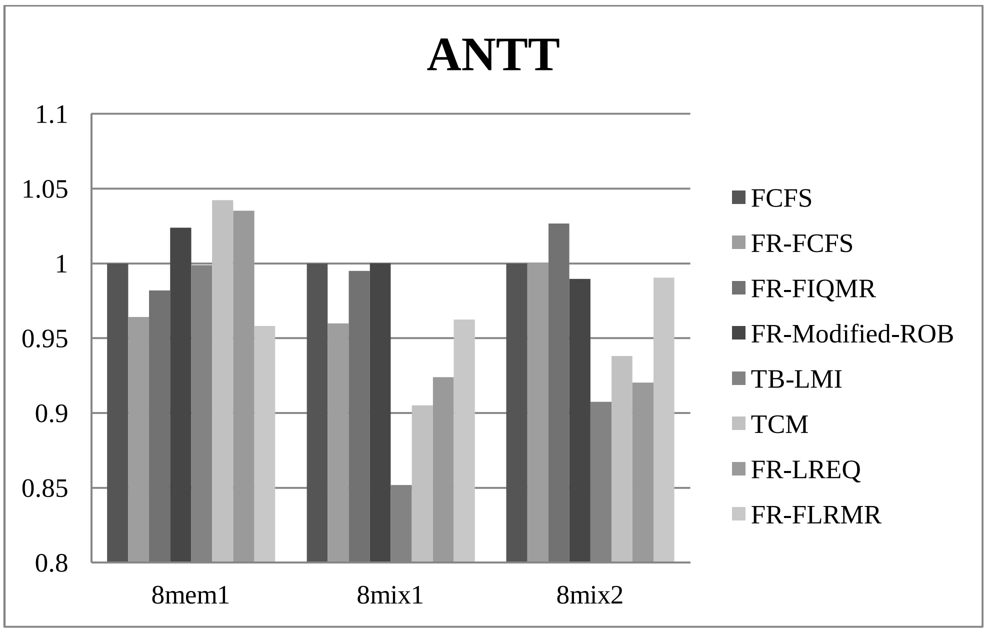


Figure 6.30: ANTT 8-core \ 32 entries memory bank queue

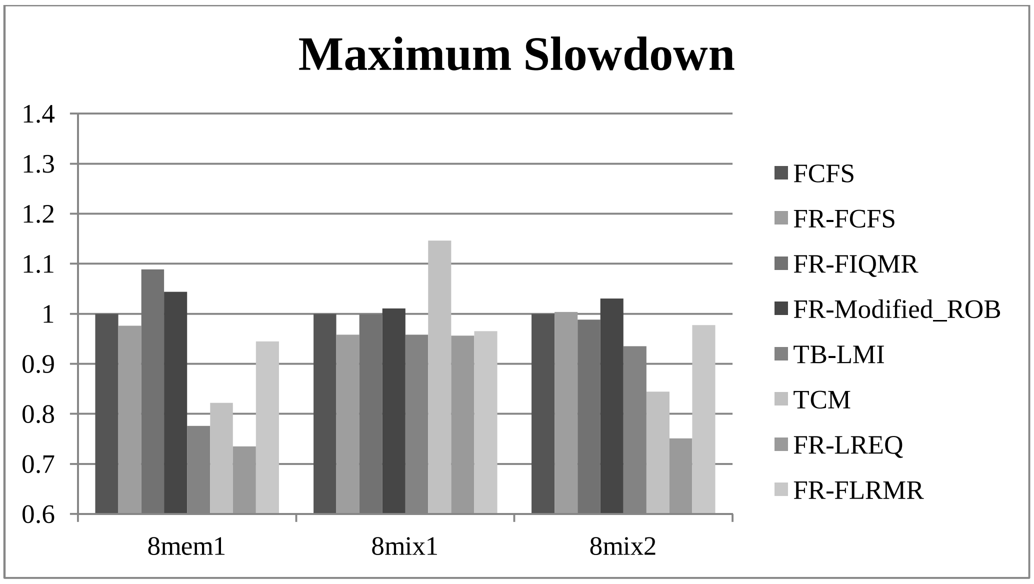


Figure 6.31: Maximum slowdown 8-core \ 32 entries memory bank queue

Figure 6.32, Figure 6.33, and Figure 6.34 show the average values of weighted speedup, ANTT and maximum slowdown respectively in case memory bank queue has different sizes. In 4-core, processor's performance and fairness are not affected when memory bank queue size changes. On average all memory scheduling algorithms performance and fairness were improved by 1% to 2 % when the memory bank queue size is increased from 8 entries to 16 entries. Memory scheduling algorithms showed no improvements in performance or fairness when the memory bank queue size is increased to 24 entries or 32 entries. In case memory bank queue has 16, 24, and 32 entries, TB-LMI and FR-LREQ have the best performance and fairness improvements. FR-FLRMR is in the third place.

In 8-core, it is observed that when the memory bank queue size increases, TB-LMI targets performance at the expense of fairness. In case memory bank queue has 8 entries. On average TB-LMI compared to TCM increased weighted speedup by 4.2% has same ANTT and decreased maximum slowdown by 11.7%. FR-LREQ compared to TCM decreased weighted speedup by 13.3%, increased ANTT by 3%, and decreased maximum slowdown by 12.8% than TCM. FR-FLRMR compared to TCM decreased weighted speedup by 18.4%, increased ANTT by 2.5%, and decreased maximum slowdown by 3.7%. TB-LMI has the best processor's performance and fairness improvement. TCM is in the second place. FR-LREQ and FR-FLRMR are in the third and fourth place respectively.

In case memory bank queue has 16 entries. On average TB-LMI has maximized the weighted speedup by 4.05%, decreased ANTT by 3%, and decreased maximum slowdown by 8.08% than TCM. FR-LREQ has decreased the weighted speedup by 10.06%, increased ANTT by 4%, and decreased maximum slowdown by 13.5% than TCM. FR-FLRMR has decreased weighted speedup by 24%, increased ANTT by 3.08%, and increased maximum slowdown by 3.2% than TCM.

In case memory bank queue has 24 entries. TB-LMI has the best performance across all workloads. On average TB-LMI has maximized the speedup by 4.95%, decreased ANTT by

2.58%, and decreased maximum slowdown by 10.88% than TCM. FR-LREQ has decreased speedup by 5.66%, increased ANTT by 2.75%, and decreased maximum slowdown by 17.29% than TCM. FR-FLRMR has decreased weighted speedup by 22.88%, increased ANTT by 3.09%, and the same maximum slowdown when it is compared to TCM. On average, TB-LMI lies in the first place among memory scheduling algorithms from performance point of view but it is in the second place after FR-LREQ from fairness point of view. The same results are obtained when the memory bank queue size is increased to 32 entries.

In summary, TB-LMI and FR-LREQ have the same performance and fairness improvement in case of 4-core system across all memory bank queue sizes. FR-FLRMR is in the third place. In 8-core, on average TB-LMI has the highest performance when the memory bank queue has 8 entries. TCM is in the second place and FR-LREQ is in the third place. TB-LMI shares the best fairness with FR-LREQ when the memory bank queue has 8 entries. As the main memory queue size increases, TB-LMI increases processor's performance at the expense of fairness. On average TB-LMI compared to FR-LREQ has increased maximum slowdown by 5%, 5.7%, and 7.9% when main memory queue sizes are with 16, 24, and 32 entries respectively. FR-LREQ has the best fairness in case memory bank queue has 16, 24, 32 entries but it is in the third place from performance point of view after TB-LMI and TCM.

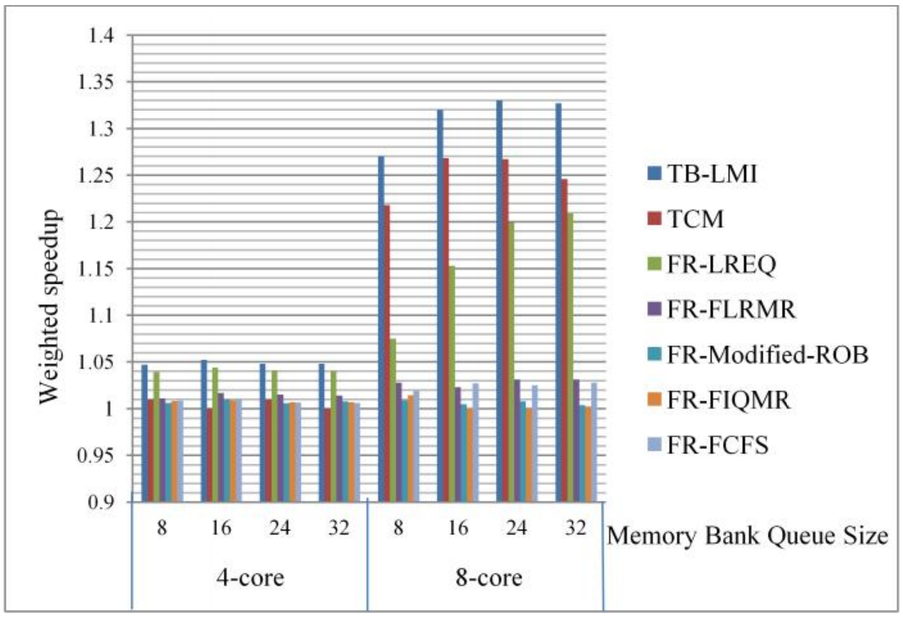


Figure 6.32: Average Weighted speedup \ different memory bank queue sizes

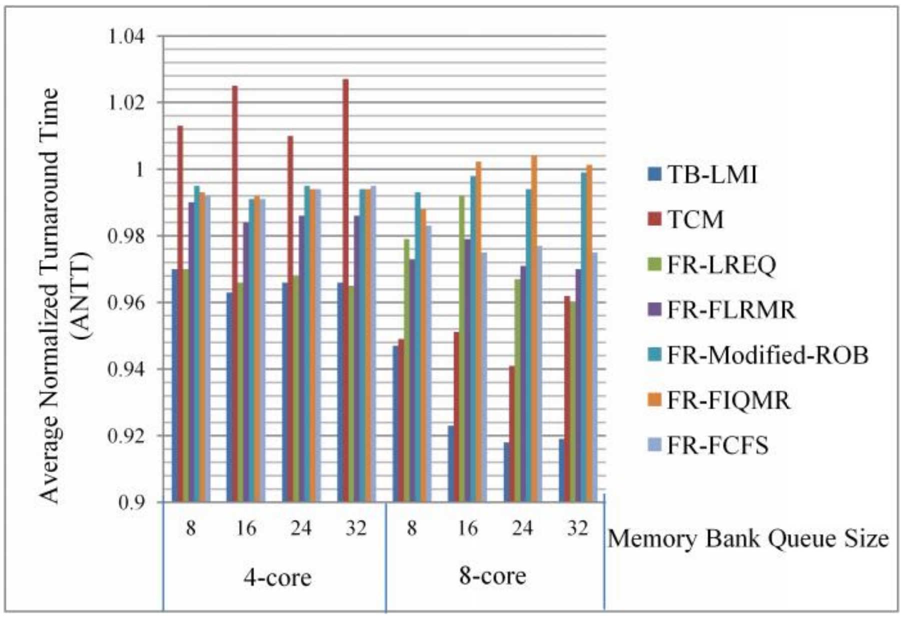


Figure 6.33: Average ANTT \ different memory bank queue sizes

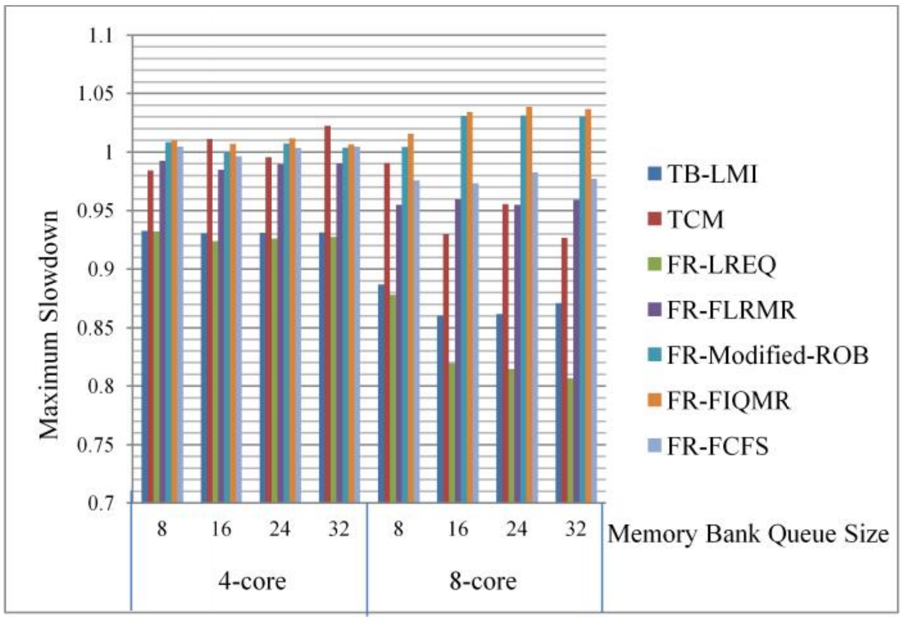


Figure 6.34: Average Maximum slowdown \ different memory bank queue sizes

Chapter 7. Conclusion and Future work

In this chapter we summarize the work done in this thesis. Also we write our plan for the future work.

7.1 Conclusion

We presented the new memory scheduling Time-Based Least Memory Intensive scheduling (TB-LMI) which provides the highest system throughput and the best fairness. TB-LMI changes threads priority every a pre-defined quantum (SQ) to improve fairness. Also, TB-LMI prioritizes row buffer hit requests over any waiting request to improve throughput and prevent starvation of memory intensive threads within a schedule quantum (improve fairness).

TB-LMI with memory bank queue size of 8 entries has the best fairness and throughput compared to the best recently proposed memory scheduling algorithms across all workloads. On average TB-LMI has the best weighted speedup by 4.22% than TCM (best performance improvement scheduling from other scheduling algorithms) with no changes at ANTT, and has the best fairness as FR-LREQ with an improvement by 7.74% than FR-FLRMR PF (second best fairness scheduling of all scheduling algorithms). Unlike FR-LREQ, TB-LMI targets processor's at the expense of processor's performance in case of 100% memory intensive workloads. TB-LMI improves both processor's performance and fairness in case of 50% memory non-intensive workloads. TB-LMI fairness was decreased compared to FR-LREQ but its throughput was noticeably increased when memory bank queue sizes has 16, 24, and 32 entries. TB-LMI is an effective memory scheduling for both low-count and high count cores in multi-core systems.

TB-LMI only requires hardware to support profiling of running applications/threads. TB-LMI does not require extra ALU to calculate priorities of waiting requests in the memory bank queue. Calculations can be performed on any core with an idle ALU when the main memory bank is busy. TB-LMI extra hardware depends on the number of memory controller used and the number of cores. Beyond first ready bit, and added bits to memory request address, 728 bits and 196 bits are required for 8-core and 4-core system respectively. The area of the added hardware storage and logic to implement TB-LMI are negligible compared to TCM which requires approximately 1168 bits and 584 bits in case of 8-core and 4-core system respectively.

7.2 Future work

In this section we propose our suggestion for future work in order to improve speedup, fairness, and scalability of TB-LMI

- **Dynamically set SQ**

Setting the SQ dynamically is considered one of the main tasks to improve scalability of TB-LMI. The idea is to change SQ value while the processor is working in order to improve throughput, fairness, or both.

- **L2 – Main memory connections and connectivity**

TB-LMI was tested on 4-banks main memory and each bank has its own controller. L2 is connected to all bank memory controllers. How will TB-LMI perform when 4-bank main memory has only two controllers? How will TB-LMI perform when L2 is only connected to only 1 controller? How will TB-LMI perform in case a multi-core processor is connected to 8-bank main memory?

Sensitivity analysis of TB-LMI must be performed to show the effect of these parameters. In addition, a parameter that represents the number of main memory banks may be added. TB-LMI uses this parameter in order to perform better multi-core processor throughput and better fairness between running applications/threads.

- **Power Usage and Energy Consumption**

Power and energy analysis need to be performed for different scheduling. TB-LMI favor decreases if it uses more power and consumes more energy than other scheduling even if it has the best speedup and fairness. Also, applications/threads profiling storage must be decreased or different applications/threads profiling are combined so that energy consumption and power usage are decreased.

- **Multithreaded processors and other benchmarks**

TB-LMI was tested on multi-core processors and SPEC CPU2006 which reflects current working applications. Multithreaded benchmarks such as PARSEC benchmarks need to be simulated. PARSEC benchmarks need to be used in evaluation as it reflect the next generation shared memory programs for CMP [37]. Also, Mediabench benchmarks need to be used in evaluation as it stands for multimedia benchmarks. TB-LMI must target these benchmarks and show competitive results against other memory scheduling algorithms.

References

1. Marr, D.T., Binns, F., Hill, D.L., Hinton, G., Koufaty, D.A., Miller, J.A., and Upton, M., “Hyper-Threading Technology Architecture and Microarchitecture”, Intel Technology Journal Q1, 2002.

2. Anderson, D.W., Sparacio, F.J., and Tomasulo, R.M., “The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling”, IBM Journal of Research and Development, vol. 11, pp. 8-24, January 1967.

3. Qayum, M.A., “Heterogeneous Chip Multiprocessors: A Survey”, ECEN 6253: Advanced Topics in Computer architecture.

4. Blake, G., Dreslinski, R.G., Mudge, T., “A survey of multicore processors”, Signal Processing Magazine, IEEE , vol.26, no.6, pp.26,37, November 2009 doi: 10.1109/MSP.2009.934110

5. Kumar, D., Chauhan A., “Multi-Core Processors and Where We’re Headed”, International Journal of Innovation Research in Technology. Volume 1 Issue 5, ISSN : 2349-6002

6. http://www.valvesoftware.com/

7. http://www.crytek.com/

8. Smith, A.J., Line (Block) Size Choice for CPU Caches, IEEE Trans. on Computers, C-36, September 1987.

9. Kessler, R., McLellan, E., and Webb, D., “The Alpha 21264 microprocessor architecture”, International Conference on Computer Design, October 1998.

10. Gwennap, L., “Digital Leads the Pack with the 21164”, Microprocessor Report, pp. 1,6-10, Sept. 12, 1994.

11. Kroft, D., “Lockup-Free Instruction Fetch/Prefetch Cache Organization”, In the Proceedings of the 8th International Symposium on Computer Architecture, May 1981.

12. ITRS, International Technology Roadmap for Semiconductors, 2008 Update. http://www.itrs.net/Links/2008ITRS/Update/ 2008Tables_FOCUS_B.xls.

13. Mogul, J.C., Argollo, E., Shah, M., Faraboschi, P., “Operating system support for NVM+DRAM hybrid main memory”, Hot Topics in Operating Systems (HotOS) (2009).

14. Koopman, P., "Main Memory Architecture", October 1998, Carnegie Mellon, pp. 3, 6.

15. Robinson, T., and Zuravleff, W.K., “Controller for a synchronous DRAM that maximizes throughput by allowing memory requests and commands to be issued out of order”, U.S. Patent No. 5,630,096. 13 May 1997.

16. Mathew, B.K., McKee, S.A., Carter, J.B., and Davis, A., “Design of a parallel vector access unit for SDRAM memory systems”, in Proceedings of the Sixth International Symposium on High-Performance Computer Architecture, pp. 39 –48, Jan 2000.

17. Moyar, S.A., “Access ordering and effective memory bandwidth”, Technical Report TR CS-93-18, April 1993.

18. McKee, S.A., and Wulf, W.A., “Access ordering and memory-conscious cache utilization”, in Proceedings of the First International Symposium on High-Performance Computer Architecture, pp. 253 –262, Jan 1995.

19. Hong, S.I., McKee, S.A., Salinas, M.H., Klenke, R.H., Aylor, J.H., and Wulf, W.A., “Access order and effective bandwidth for streams on a direct rambus memory”, in Proceedings of the Fifth International Symposium on High-Performance Computer Architecture, pp. 80 –89, Jan 1999.

20. Zhu, Z., and Zhang, Z., “A performance comparison of DRAM memory system optimizations for SMT processors”, in Proceedings of the 11th International Symposium on High-Performance Computer Architecture, pp. 213 –224, 2005.

21. Rixner, S., “Memory controller optimizations for web servers”, Microarchitecture, 2004. MICRO-37 2004. 37th International Symposium on , vol., no., pp.355,366, 04-08 Dec. 2004.

22. Rixner, S., Dally, W.J., Kapasi, U.J., Mattson, P., and Owens, J.D., “Memory access scheduling”, Computer Architecture, International Symposium on, pp. 128-128. ACM Press, 2000.

23. Lakshminarayana, N.B., Lee, J., and Kim, H., “Age based scheduling for asymmetric multiprocessors”, in Proceedings of Supercomputing: the Conference on High Performance Computing Networking, Storage and Analysis (SC), Nov.2009.

24. El-Reedy, W., El-Moursy, A., and Fahmy, H.A.H., ”High performance memory requests scheduling technique for multicore processors”, High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS), 2012 IEEE 14th International Conference on , vol., no., pp.127,134, 25-27 June 2012.

25. Zheng, H., Lin, J., Zhang, Z., and Zhu, Z., “Memory access scheduling schemes for systems with multi-core processors”, The 37th International Conference on Parallel Processing, 2008.

26. Kim, Y., Papamichael, M., Mutlu, O., and Harchol-Balter, M., “Thread Cluster Memory Scheduling: Exploiting Differences in Memory Access Behavior”, Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACMInternational Symposium on, vol., no., pp.65,76, 4-8 Dec. 2010.

27. Mutlu, O., and Moscibroda, T., “Parallelism-Aware Batch Scheduling: Enhancing both Performance and Fairness of Shared DRAM Systems”, Computer Architecture, 2008. ISCA ’08. 35th International Symposium on, vol. 36, no. 3, pp.63,74, 21-25 June 2008.

28. Kim, Y., Han, D., Mutlu, O., and Harchol-Balter, M., “Atlas: A scalable and high-performance scheduling algorithm for multiple memory controllers”, the 16th International Symposium on High-Performance Computer Architecture, 2010.

29. Yadav, R.K., Mishra, A.K., Prakash, N., Sharma, H., “An Improved Round Robin Scheduling Algorithm for CPU Scheduling”, (IJCSE) International Journal on Computer Science and Engineering Vol. 02, No. 04, 1064-1066, 2010.

30. Mutlu, O., and Moscibroda, T., “Stall-time fair memory access scheduling for chip multiprocessors”, in Proceedings of the 40th International Symposium on Microarchitecture, pp. 208–222, Dec. 2007.

31. http://web.eecs.umich.edu/~taustin/

32. Burger, D., and Austin, T., “The simplescalar toolset, version 2.0.”, Technical Report TR-97-1342, June 1997.

33. El-Moursy, A., Garg, R., Albonesi, D.H., and Dwarkadas, S., “Partitioning multi-threaded processors with a large number of threads”, in International Symposium on Performance Analysis of Systems and Software, pp. 112 – 123, March 2005.

34. Ubal, R., Jang, B., Mistry, P., Schaa, D., and Kaeli, D., Multi2Sim, “A simulation framework for CPU-GPU computing”, in Proceedings of the 21st international conference on Parallel architectures and compilation techniques, pp. 335-344. ACM, 2012.

35. https://www.multi2sim.org/

36. Eyerman, S., and Eeckhout, L., “Restating the Case for Weighted-IPC Metrics to Evaluate Multiprogram Workload Performance”, IEEE Computer Architecture Letters 99, no. 2 (2013): 1

37. http://parsec.cs.princeton.edu/

38. http://euler.slu.edu/~fritts/mediabench/.

39. Lee, C., Potkonjak, M., Mangione-Smith, W.H., "MediaBench: a tool for evaluating and synthesizing multimedia and communications systems," Microarchitecture, 1997.

Proceedings., 3th Annual IEEE/ACM International Symposium on , vol., no., pp.330,335,1-3 Dec 1997doi: 10.1109/MICRO.1997.645830

40. http://www.spec.org/cpu2006/.

Appendix A. Benchmarks

There are several benchmarks that can be used in our evaluation. We are interested in the benchmarks that are available in the Multi2Sim website [35].

A.1 PARSEC

PARSEC [27] stands for Princeton Application Repository for Shared-Memory Computers. PARSEC is a benchmark suit composed of multithreaded programs. The suite focuses on emerging workloads and was designed to be representative of next-generation shared-memory programs for chip-multiprocessors. PARSEC consists of 13 benchmarks. Table A.1 summarizes the PARSEC benchmarks.

A.2 Mediabench

Mediabench benchmarks targets multimedia and communication applications. It contains video processing algorithms, audio encoding, and image compressing applications. Table A.2 summarizes benchmarks introduced in Mediabench. For more information refer to [38] and [39].

A.3 SPEC CPU 2006

SPEC benchmarks suite is designed to provide a comparative measure of compute-intensive performance across the widest practical range of hardware using workloads developed from real user applications. SPEC2006 benchmark suite includes the SPECint benchmarks (integer benchmarks) and the SPECfp benchmarks (floating point benchmarks). SPECint 2006 benchmark contains 12 different benchmarks. SPECfp 2006 benchmark contains 19 different benchmarks. Table A.3 and Table A.4 summarize the SPECint 2006 and SPECfp 2006 benchmarks respectively. For more information refer to [40].

Table A.1: PARSEC Benchmarks

Benchmarks	Description
blackscholes	Option pricing with Black-Scholes Partial Differential Equation (PDE)
bodytrack	Body tracking of a person
canneal	Simulated cache-aware annealing to optimize routing cost of a chip design
dedup	Next-generation compression with data deduplication
facesim	Simulates the motions of a human face
ferret	Content similarity search server
fluidanimate	Fluid dynamics for animation purposes with Smoothed Particle Hydrodynamics (SPH) method
freqmine	Frequent itemset mining
raytrace	Real-time raytracing
streamcluster	Online clustering of an input stream
swaptions	Pricing of a portfolio of swaptions
vips	Image processing
X264	H.264 video encoding

Table A.2: Mediabench Benchmarks

Benchmark	Description
JPEG	Implement JPEG image compression and decompression for full-color and gray-scale images.
MPEG	Standard for high quality digital video transmission. Used by mpeg2enc and mpeg2dec.
GSM	Implementation of the European GSM 06.10 provisional standard for full rate speech transcoding.
PGP	Use message digests to form signatures.
Pegwit	A program for performing public key encryption and authentication.
Gostscript	Interpreter for the PostScript language.
Mesa	3-D graphics library clone of OpenGL.
SPHERE	A set of library functions and command-level programs which can be used to read and modify NIST-formatted speech waveform files.
RESTA	A program for speech recognition.
EPIC	An experimental image data compression utility.
ADPCM	Adaptive differential pulse code modulation.
G.721	The files in this package comprise ANSI-C language reference implementations of the CCITT G.711, G.721 and G.723 voice compressions.

Table A.3: SPECint CPU2006 Benchmarks

Benchmark	Language	Application area	Description
400.perlbench	C	Programming Language	Derived from Perl V5.8.7. The workload includes SpamAssassin, MHonArc (an email indexer), and specdiff (SPEC's tool that checks benchmark outputs).
401.bzip2	C	compression	Julian Seward's bzip2 version 1.0.3, modified to do most work in memory, rather than doing I/O.
403.gcc	C	Compiler	Based on gcc Version 3.2, generates code for Opteron.
429.mcf	C	Combinatorial Optimization	Vehicle scheduling. Uses a network simplex algorithm (which is also used in commercial products) to schedule public transport.
445.gobmk	C	Artificial Intelligence: Go	Plays the game of Go, a simply described but deeply complex game.
456.hmmer	C	Search Gene Sequence	Protein sequence analysis using profile hidden Markov models (profile HMMs)
458.sjeng	C	Artificial Intelligence: chess	A highly-ranked chess program that also plays several chess variants.
462.libquantum	C	Physics / Quantum Computing	Simulates a quantum computer, running Shor's polynomial-time factorization algorithm.
464.h264ref	C	Video Compression	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2
471.omnetpp	C++	Discrete Event Simulation	Uses the OMNet++ discrete event simulator to model a large Ethernet campus network.
473.astar	C++	Path-finding Algorithms	Pathfinding library for 2D maps, including the well known A* algorithm.
483.xalancbmk	C++	XML Processing	A modified version of Xalan-C++, which transforms XML documents to other document types.

Table A.4: SPECfp CPU2006 Benchmarks

Benchmark	Language	Application area	Description
410.bwaves	Fortran	Fluid Dynamics	Computes 3D transonic transient laminar viscous flow.
416.gamess	Fortran	Quantum Chemistry.	Gamess implements a wide range of quantum chemical computations. For the SPEC workload, self-consistent field calculations are performed using the Restricted Hartree Fock method, Restricted open-shell Hartree-Fock, and Multi-Configuration Self-Consistent Field
433.milc	C	Physics / Quantum Chromodynamics	A gauge field generating program for lattice gauge theory programs with dynamical quarks.
434.zeusmp	Fortran	Physics / CFD	ZEUS-MP is a computational fluid dynamics code developed at the Laboratory for Computational Astrophysics (NCSA, University of Illinois at Urbana-Champaign) for the simulation of astrophysical phenomena.
435.gromacs	C, Fortran	Biochemistry / Molecular Dynamics	Molecular dynamics, i.e. simulate Newtonian equations of motion for hundreds to millions of particles. The test case simulates protein Lysozyme in a solution.
436.cactusADM	C, Fortran	Physics / General Relativity	Solves the Einstein evolution equations using a staggered-leapfrog numerical method
437.leslie3d	Fortran	Fluid Dynamics	Computational Fluid Dynamics (CFD) using Large-Eddy Simulations with Linear-Eddy Model in 3D. Uses the MacCormack Predictor-Corrector time integration scheme.
444.namd	C++	Biology / Molecular Dynamics	Simulates large biomolecular systems. The test case has 92,224 atoms of apolipoprotein A-I.
447.dealII	C++	Finite Element Analysis	It is a C++ program library targeted at adaptive finite elements and error estimation. The testcase solves a Helmholtz-type equation with non-constant coefficients.
450.soplex	C++	Linear Programming, Optimization	Solves a linear program using a simplex algorithm and sparse linear algebra. Test cases include railroad planning and military airlift models.

453.povray	C++	Image Ray-tracing	Image rendering. The testcase is a 1280x1024 anti-aliased image of a landscape with some abstract objects with textures using a Perlin noise function.
454.calculix	C, Fortran	Structural Mechanics	Finite element code for linear and nonlinear 3D structural applications. Uses the SPOOLES solver library.
459.GemsFDTD	Fortran	Computational Electromagnetics	Solves the Maxwell equations in 3D using the finite-difference time-domain (FDTD) method.
465.tonto	Fortran	Quantum Chemistry	An open source quantum chemistry package, using an object-oriented design in Fortran 95. The test case places a constraint on a molecular Hartree-Fock wavefunction calculation to better match experimental X-ray diffraction data.
470.lbm	C	Fluid Dynamics	Implements the "Lattice-Boltzmann Method" to simulate incompressible fluids in 3D
481.wrf	C, Fortran	Weather	Weather modeling from scales of meters to thousands of kilometers. The test case is from a 30km area over 2 days.
482.sphinx3	C	Speech recognition	A widely-known speech recognition system from Carnegie Mellon University

Table A.5: Single core processor parameters

Processor pipeline	2.33 GHz, x86 processor, 64-entry fetch queue size, 32-entry re-order buffer, 40-entry issue queue size, and 20-entry load-store queue size
Fast-forwarding	1 billion instructions
Detailed simulation	25 million instructions
L1 cache	64 Kbytes, 2-way set associative, 2 cycles latency, Least recently used replacement policy, 2 ports
L2 cache	4 Mbytes, 4-way set associative, 11 cycles latency, Least recently used replacement policy, 4 ports, 64-byte block size
DRAM Chip parameters	4-banks, 256-Bytes row-buffer per bank, 1 controller per bank, main memory queue with 8 entries
Round-trip L2 miss latency	For a 64-byte cache line, row-buffer hit: 46ns (108 cycles), closed: 60ns (140 cycles), conflict: 92ns (216 cycles)

SPEC2006 benchmarks were classified into memory intensive benchmarks and memory non-intensive benchmarks. SPEC 2006 benchmarks were simulated alone in a multi-core processor or simulated in a single core processor with the parameters shown in Table A.5. MPKI for each benchmark was calculated. Benchmarks with MPKI less than 1 are classified as memory non-intensive benchmarks; others are classified as memory intensive benchmarks. Table A.6 shows the results of SPEC 2006 classification.

Table A.6: Benchmarks MPKI

Benchmarks	L2 Misses	Instructions	MPKI
429.mcf	39257838	1752604973	22.399707
470.lbm	25794948	2644797371	9.7530905
462.libquantum	10988941	1571729331	6.9916243
410.bwaves	14608599	2357468772	6.1967307
483.xalancbmk	7216122	1876740841	3.8450285
471.omnetpp	3704776	1547116146	2.3946334
401.bzip2	3767772	1857054462	2.02889688
436.cactusADM	4561656	2968216881	1.53683379
403.gcc	1723496	1859262932	0.92697809
456.hmmer	748579	1758863030	0.42560392
473.astar	1298070	1847756712	0.70251132
464.h264ref	10226	3691915767	0.00276984

الملخص

تعتبر الذاكرة الرئيسية هي مورد مشترك بين عدة تطبيقات في نظام الرقاقة متعددة الأنوية الحديثة. يتم تطوير خوارزميات جدولة الذاكرة لحل الخلاف من خلال التحكم في الطلبات المرسلة للذاكرة من التطبيقات المتنافسة للوصول إلي أداء عالي ، مضمون و الإنصاف بين التطبيقات في أنظمة متعددة النواة. وهذا يؤكد أهمية جدولة طلبات الذاكرة للإستفاده من عرض النطاق الترددي للذاكرة بكفاءة. على الرغم من أن تقنيات جدولة طلبات الذاكرة قد اقترحت مؤخرا لتحسين الأداء، و لكن معظمها تغاضي عن الإنصاف بين التطبيقات.

نقدم في هذه الرسالة نظام جدولة جديدة و هي جدولة قائمة علي الوقت الأقل طلبات للذاكرة الرئيسية. الفكرة الرئيسية لنظام الجدولة الجديد هي تحديد الأولوية للتطبيقات وفقا لطلباتهم إلي الذاكرة الرئيسية تحسب كل فترة زمنية محدده سابقا لتحسين الأداء و الإنصاف بين التطبيقات. نقيم الجدولة القائمة علي الوقت الأقل طلبات للذاكرة الرئيسية من خلال مجموعة متنوعة من أعباء العمل مع أحجام مختلفة من قائمة الإنتظار في وحدات تحكم الذاكرة الرئيسية ومقارنة أدائها مع ستة من خوارزميات الجدولة المقترحة سابقاً. الجدولة قائمة علي الوقت الأقل طلبات للذاكرة الرئيسية يحقق أفضل أداء وإنصاف بين التطبيقات في نظام الرقاقة متعددة الأنوية الحديثة. الخوارزميات المقترحة سابقا هي جدولة الجاهز أولاً الأول يأتي أولاً يخدم ، جدولة الجاهز أولاً عدالة الأقل طلبات للذاكرة الرئيسية ذات صلة ، جدولة الجاهز أولاً عدالة الإصدارات في قائمة الإنتظار ذات صلة ، جدولة الجاهز أولاً إعادة ترتيب سجل التخزين المؤقت على أساس بناء المعدل ، جدولة الجاهز أولاً الأقل طلبات للذاكرة الرئيسية و جدولة تقسيم التطبيقات إلي كتل. أظهر جدولة تقسيم التطبيقات إلي كتل ، جدولة الجاهز أولاً الأقل الطلبات للذاكرة الرئيسية ، و جدولة الجاهز أولا عدالة الأقل طلبات للذاكرة الرئيسية ذات صلة نتائج تنافسية ضد الجدولة الجديدة القائمة علي الوقت الأقل طلبات للذاكرة الرئيسية. في نظم الرقاقة ذات الثماني نواة، الجدولة القائمة علي الوقت الأقل طلبات للذاكرة الرئيسية يحسن إنتاجية النظام والإنصاف بين التطبيقات بنسبة ٤,٢٢٪ و ١١,٧٪ في المتوسط على التوالي مقارنة مع جدولة تقسيم التطبيقات إلي كتل الذي يوفر أفضل أداء وإنصاف بين التطبيقات من بين الخورزمات المقترحة سابقاً.

مهندس: عمرو صالح أبوبكر خليل الحلو

تاريخ الميلاد: ١٩٨٦/٦/٢٢

الجنسية: مصري

تاريخ التسجيل: ٢٠١٠/١٠/١

تاريخ المنح:

الدرجة: ماجستير العلوم

القسم: هندسة الإلكترونيات و الإتصالات الكهربية

المشرفون:

أ.م.د حسام علي حسن فهمي

د. علي علي المرسي (باحث بمعهد بحوث الإلكترونيات)

الممتحنون:

أ.م.د حسام علي حسن فهمي (المشرف)

أ.د. أمين محمد نصار (الممتحن الداخلي)

أ.د السيد مصطفي سعد (الممتحن الخارجي)

عنوان البحث:

نظام جدولة زمني مدرك للعدالة بين التطبيقات لطلبات الذاكرة في المعالجات متعددة الأنوية.

الكلمات الدالة:

متعددة الأنوية، التحكم في الذاكرة، موارد مشتركة، تداخل الطلبات في الذاكرة

ملخص البحث:

تعتبر الذاكرة الرئيسية هي مورد مشترك بين عدة تطبيقات في نظام الرقاقة متعددة الأنوية الحديثة. يتم تطوير خوارزميات جدولة الذاكرة لحل الخلاف من خلال التحكم في الطلبات المرسلة للذاكرة من التطبيقات المتنافسة، مما يؤدي إلى ارتفاع إنتاجية الرقاقة مع مراعاة نظام العدالة بين التطبيقات المتنافسة. نظام جدولة زمني مدرك للعدالة بين التطبيقات لطلبات الذاكرة هو نظام جدولة جديد لتحسين الإنتاجية والعدالة للمعالجات متعددة الأنوية. تم مقارنة نظام الجدولة الجديد مع عدة جدولة للذاكرة المقترحة سابقا وأظهر النظام الجديد نتائج تنافسية.

نظام جدولة زمني مدرك للعدالة بين التطبيقات لطلبات الذاكرة في المعالجات متعددة الأنوية

إعداد

عمرو صالح ابو بكر خليل الحلو

رسالة مقدمة الي كلية الهندسة – جامعة القاهرة

كجزء من متطلبات الحصول علي درجة ماجستير العلوم

في

هندسة الإلكترونيات و الإتصالات الكهربية

يعتمد من لجنة الممتحنين:

الأستاذ المساعد: حسام علي حسن فهمي	المشرف
الأستاذ الدكتور: أمين محمد نصار	الممتحن الداخلي
الأستاذ الدكتور: السيد مصطفي سعد (أستاذ دكتور بهندسة حلوان)	الممتحن الخارجي

كلية الهندسة – جامعة القاهرة

الجيزة – جمهورية مصر العربية

٢٠١٥

نظام جدولة زمني مدرك للعدالة بين التطبيقات لطلبات الذاكرة في المعالجات متعددة الأنوية

إعداد

عمرو صالح ابو بكر خليل الحلو

رسالة مقدمة الي كلية الهندسة – جامعة القاهرة
كجزء من متطلبات الحصول علي درجة ماجستير العلوم
في
هندسة الإلكترونيات و الإتصالات الكهربية

تحت إشراف

أ.م.د حسام علي حسن فهمي
قسم هندسة الإلكترونيات و الإتصالات الكهربية

د. علي علي المرسي
معهد بحوث الإلكترونيات

كلية الهندسة – جامعة القاهرة

الجيزة – جمهورية مصر العربية

٢٠١٥

نظام جدولة زمني مدرك للعدالة بين التطبيقات لطلبات الذاكرة في المعالجات متعددة الأنوية

إعداد

عمرو صالح ابو بكر خليل الحلو

رسالة مقدمة الي كلية الهندسة – جامعة القاهرة

كجزء من متطلبات الحصول علي درجة ماجستير العلوم

في

هندسة الإلكترونيات و الإتصالات الكهربية

كلية الهندسة – جامعة القاهرة

الجيزة – جمهورية مصر العربية

٢٠١٥