NEW REAL-TIME MEMORY CONTROLLER DESIGN FOR EMBEDDED MULTI-CORE SYSTEM

By

Ahmed Shafik Shafie Mohamed

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

2016

NEW REAL-TIME MEMORY CONTROLLER DESIGN FOR EMBEDDED MULTI-CORE SYSTEM

By
Ahmed Shafik Shafie Mohamed

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

Prof. Dr. Hossam A. H. Fahmy	Dr. Ali A. El-Moursy
……………………………….	……………………………….
Professor,	Associate professor,
Electronics and Communications Engineering ,	Computers and Systems department,
Faculty of Engineering, Cairo University	Electronics Research Institute

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2016

NEW REAL-TIME MEMORY CONTROLLER DESIGN FOR EMBEDDED MULTI-CORE SYSTEM

By
Ahmed Shafik

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

____
Prof. Dr. Hossam A. H. Fahmy, Thesis Main Advisor

____
Dr. Ali A. El-Moursy, Member

____
Prof. Dr., Mohamed Ryad El Ghonemy

____
Prof. Dr., Hazem Mahmoud Abbas
(Professor at Ain Shams University)

FACULTY OF ENGINEERING, CAIRO UNIVERSITY
GIZA, EGYPT
2016

Engineer's Name:	Ahmed Shafik Shafie Mohamed
Date of Birth:	
Nationality:	Egyptian
E-mail:	eng.ahmed.shafik@gmail.com
Phone:	
Address:	
Registration Date:	01/10/2010
Awarding Date	/ /2016
Degree:	Master of Science
Department:	Electronics and Communications Engineering
Supervisors:	Prof. Dr. Hossam A. H. Fahmy Dr. Ali A. El-Moursy
Examiners:	Prof. Dr. Hazem Mahmoud Abbas (External Examiner) Faculty of Engineering, Ain Shams university Prof. Dr. Mohamed Ryad El Ghonemy (Internal Examiner) Dr. Ali A. El-Moursy (Member) Prof. Dr. Dr. Hossam A. H. Fahmy (Thesis Main Advisor)

Title of Thesis:

New Real-time Memory Controller for Embedded Multi-Core System

Key Words:

Memory Controller, Real-time, Chip Multi-Processor, Embedded Systems

Summary:

Nowadays modern Chip Multi-Processors (CMPs) become more demanding because of their high performance especially in real-time embedded systems. On the other side, bounded latencies has become vital to guarantee high performance and fairness for applications running on CMPs cores. In modern embedded systems, CMPs has become more effective choice due to their low power and high performance. As application running on these processors use the shared resources such as: memory, the shared memory should provide high memory service rates to be able to serve multiple cores in an acceptable response time. In CMPs, real time applications are bound by the worst case estimated time to provide hard deadlines to real-time tasks (*WCET*). The modern systems use the Double Data Rate Dynamic RAM (DDR DRAM). Yet, constant *WCET* for the interfered threads cannot be guaranteed due to sequential serving between requests, so each request status depends on the previous and concurrent requests. Another reason is that DRAM access time has a high variation due to caching data in row buffers before reading or writing.

We propose a new memory controller that prioritizes and assigns defined quotas for cores within unified epoch time accompanied with fair round robin scheduling within cores it selves (*MCES*). *MCES* works on variety of generations of double data rate DRAM (DDR DRAM). *MCES* can differentiate between different types of requests as hard real-time request (*HRT*) and non-hard real-time requests (*NHRT*). Hence, *MCES* can run multimedia real-time applications and hard real-time applications. *MCES* is able to achieve an overall performance reached 35% for 4 cores system and an overall performance speedup of 16% for 8 cores system and the same level of power consumption compared to the last released memory controller design (*WCAD*).

Acknowledgements

Many people contributed both directly and indirectly to finish and submit dissertation. First of all, I would like to thank my mother and my father for their large and continuous support. They were my shelter, rocket launcher and everything to me in my graduate life.

I would thank my supervisors, Dr. Ali A. El-Moursy and prof. Dr.Hossam A. H. Fahmy for giving me this opportunity to work and learn from them the fundamentals to do appropriate research. They helped me to build my knowledge in computer architecture. They motivated me to perform serious research and were patient with me in making good progress in my research and publishing my ideas.

Dedication

This thesis is dedicated to my mother, my father, my wife, my baby Youssef and to those who scarified their lives trying to make our country better.

Table of Contents

List of Figures

List of Tables

List of Abbreviations and Definitions

CMC/CMP: Chip Multi-Core/Chip Multi-Processor

WCET: Worst Case Estimation Time

DDR DRAM: Double Data Rate Dynamic RAM

RAM: Random Access Memory

MCES: real-time Memory Controller for Embedded multi-core System

MC: Memory Controller

ACT: Activate command

PRE: Precharge Command

HRT: Hard Real-time Task

NHRT: Non-Hard Real-time Task

BL: Burst Length

T_{RCD}: ACT to READ/WRITE delay

T_{RL}: READ to Data Start

T_{WL}: WRITE to Data Start

T_{BUS}: Data bus transfer

T_{RP}: PRE to ACT Delay

T_{WR}: Data End of WRITE to PRE Delay

T_{RTP}: Read to PRE Delay

T_{RAS}: ACT to PRE Delay

T_{RC}: ACT-ACT (same bank)

T_{RRD}: ACT-ACT (different bank)

T_{FAW}: Four ACT Window

T_{RTW}: READ to WRITE Delay

T_{WTR}: WRITE to READ Delay

$T_{RFC:}$ Time required to refresh a row

$T_{REF:}$ REF period

OP: Open Page

CP: Closed Page

OoO: Out Of Order Cores

AMC: Analyzable Memory Controller

PREDATOR: A Predictable SDRAM Memory Controller

WCAD: Worst Case Analysis of DRAM Latency in Multi-Requestor Systems

RR: Round Robin

MOT: Memory Out Time

OR: Open Request

CR: Closed Request

T_{OR}: Open Request latency

T_{CR}: Closed Request Latency

WCOR: Worst Case Open Request

WCCR: Worst Case Closed Request

N_{OT}: Number of Open Tasks

N_{CT}: Number of Closed Tasks

CKE: Clock Enable

CS: Chip Select

RAS: Row Address Strobe

CAS: Column Address Strobe

WE: Write Enable

BAn: Bank Selection

FRFS: First Ready First Service

$IA^{3:}$ Interference Aware Allocation Algorithm

XCBA: Inter-Core Bus Arbiter

ICBA: Intra-Core Bus Arbiter

HLS: High Level Synthesis

BWPS: Bounding WCET Using SDRAM with Priority Scheduling

Abstract

Nowadays modern chip multi-processors (CMPs) become very attractive especially in real-time embedded systems because of their high performance and low per unit cost. On the other side, bounded latencies is vital to guarantee high performance and fairness for applications running on multicore processors. In modern embedded systems, Chip Multicore Processors (CMPs) has become more effective choice due to their low power and high performance. As application running on these processors use the shared resources such as: memory, the shared memory should provide high memory service rates to be able to serve multiple cores in an acceptable response time. In CMPs, real time applications are bounded by the Worst Case Estimated Time to provide hard deadlines to real-time tasks (*WCET*). The modern systems use the Double Data Rate Dynamic RAM (DDR DRAM) because they are able to transfer the data on the rising and falling edges of clock cycles which will provide double data rates than the normal DRAMS. Hence, DDR DRAM is the most compatible with the high demanding CMP processors. Yet, constant *WCET* for the interfered threads cannot be guaranteed due to sequential serving between requests, so each request status depends on the previous and concurrent requests. Another reason is that DRAM access time has a high variation due to caching data in row buffers before reading or writing.

In this thesis, we propose a new memory controller that prioritizes and assigns defined quotas for cores within unified epoch time accompanied with request ranking per core alongside with fair round robin scheduling among cores (*MCES*). Our approach works on variety of generations of double data rate DRAM (DDR DRAM). *MCES* can differentiate between two types of requests as hard real-time request (*HRT*) and non-hard real-time requests (*NHRT*). Hence, *MCES* can run multimedia real-time applications and hard real-time applications. *MCES* is able to achieve an overall performance speedup of 35% and for 4 cores system and an overall performance speedup of 16% for 8 cores system and the same level of power consumption compared to the last released memory controller design (*WCAD*).

Chapter 1. Introduction

1.1 Problem

In modern embedded systems, Chip Multicore Processors (CMPs) [1] has become more effective choice due to their low power, high performance and low per-unit cost. CMPs can be either homogenous or heterogeneous. Homogenous cores are identical cores. Heterogeneous is a set of cores which may differ in area, performance, power dissipated etc. As mentioned in [1], recent research in heterogeneous CMPs has identified significant advantages over homogeneous CMPs in terms of power and throughput and in addressing the effects of Amdahl's law on the performance of parallel applications.

As application running on these processors use the shared resources such as: caches, buses, buffers, queues, memory, etc... This shared memory should provide high memory service rates to be able to serve multiple cores in an acceptable response time. In CMPs, real time applications are bounded by the Worst Case Estimated Time in order to provide hard deadlines to real-time tasks (WCET). WCET is the maximum length of time the task could take to execute on a specific hardware platform. WCET is typically used in real-time systems, where understanding the worst case timing behavior of software is important for reliability or correct functional behavior [2].

Modern systems use the last released technology in memory fabrications associated with the best memory controller algorithms to exploit the high transfer rates and lower power as Double Data Rate Dynamic RAM (DDR DRAM) [3] [4].DDR DRAM transfers data on both the rising and falling edge of clock cycles. By using both edges of the clock, the data signals operate with the same limiting frequency, thereby doubling the data transmission rate. Although the constant WCET for the interfered threads, a technique to divide the executed program to two or more simultaneously running tasks, cannot be guaranteed for the following reasons[8]:

1- Each request delay, due to sequential serving, depends on the previous requests from the different cores (Inter-task Interference) or the same core (Intra-task interference).

2- DRAM access time has high variation due to caching data in row buffers before reading or writing.

1.2 Motivation

To overcome these challenges, several work were proposed by [5] [6] [7] to provide tight and guaranteed upper bound memory latency timing for real time tasks when applied to DRAM devices as DDR2. The authors in [8] proved that working on higher speed DRAM devices as DDR3 and wider buses improves the performance of their algorithm compared to previous ones in [5][6][7] because they did not exploit the advantage of caching mechanism in DDR.

MCES, our approach, exploit the use of caching mechanism in DDR3 and parallelism in DRAM structure to reduce the interference between cores. We present:

1) Priority based dynamic scheduling for real time applications for multimedia and hard real-time applications.
2) Round robin fair scheduling within cores when the starvation flag is up.
3) Re-ordering between hard real-time tasks (*HRTs*) non-hard real-time tasks (*NHRTs*).
4) DRAM techniques as private banks and open page.

1.3 Thesis Organization

The thesis is organized as follows. Chapter 2 provides required background knowledge on how DRAM works. Chapter 3 compares our approach to related work in the field. Chapter 4 discusses memory controller algorithm and arbitrary rules. Chapter 5 discusses our used simulators and evaluation results are presented in chapter 6. Chapter 7 concludes the thesis and discusses the future work.

Chapter 2. Background

Embedded system is computer system with dedicated function running usually with real-time processing constraints. Compared to general purpose computer systems, embedded systems should consume lower power with smaller fabricated size and low cost per unit. Embedded systems are mainly used in military, medical, automotive and avionics applications due to the nature of integrating these systems in other systems as hand-hold systems such as mobile phones, automotive systems such as cars, avionics systems such as plans and spaceships and human disable assistant machines such as wheel chairs.

The main components of computer system are multi-core processor and memory chips. A multi-core processor chip is a single component with two or more independent cores. These cores can handle concurrent process in the system such as read and execute instructions as add, subtract, multiply, divide for the arithmetic operations ,move data , read and write for the memory operations and branch , conditionally branch for the control flow operation by passing them to caches. Multi-core processors may have two cores (dual-core), four cores (quad-core), six cores (hexa-core), eight cores (octa-core), ten cores (deca-core) or more. Multi-core processors can run multiple instructions in parallel to increase the overall speed and consequently whole machine performance. Parallel computing is a form of computation in which many instructions are carried out simultaneously and operating on the principle that large problems can often be divided into smaller ones, which are then executed at the same time.

In case the data or instruction request was missed in cache, it is considered a cache-miss and will be forwarded to memory array through Memory Controller (*MC)* located in memory device to serve the miss.

2.1 Memory Controller Architecture and Function

Memory Controller is designed to control the access to memory arrays among cores i.e. Memory controller is used to schedule the requests in a proficient manner to achieve the maximum benefits including performance and lowest request time. Memory controller contains the algorithm required to read/write data from/to DRAM memory and refresh the DRAM. If the data in the DRAM was not refreshed in constant times, these data will be lost as the charged capacitors leak their charge. Beside the memory controller, memory chip contains the memory array. Memory arrays are organized into *ranks* and only one rank can be accessed at a time [8] as shown in Figure 2.1. Each rank is divided into multiple *banks* which can be accessed in parallel, unless no collisions occurs on either command or data buses. Each bank consists of a *row-buffer* and an array of storage cells divided into *rows* and *columns*.

2.2 Real-time Memory Controller

Real-time Memory Controller is concerned with systems subjected to real-time constraints. These types of memory controllers must guarantee response for these systems within specific time or deadlines. These systems are classified into two types:

1- Hard Real-time Systems: These systems must operate within fixed deadlines and considered failing if these deadlines are not met such as anti-lock brakes and aircraft control systems.
2- Non Hard (Soft) Real-time Systems: These systems aim is to maximize the number of tasks to meet their deadlines. Soft real-time systems can miss some deadlines and cannot be considered failing but the overall performance will be degraded, connection and service maybe re-established and resumed. If too many tasks miss their deadlines as audio streaming, a violation of the Service Level Agreement (SLA) is encountered.

Real-time Memory Controller provides predictable response to core requests i.e. these types of memory controller need to issue the request based on real-time constraint, guaranteed real-time response within specific time constraints, which does not mean the fastest response.

There are many authors proposed algorithms and designs for memory controller. Some authors proposed real-time logic. Two main types are discussed in this chapter and will be discussed in more details later in chapter 3.

2.2.1 Fair based scheduling memory controller

This type of memory controllers focuses on fair scheduling between cores. Fair scheduling algorithm that gives each core equal/fair time slots to process its job and will not gain a new access until all the cores get their slots, and request ranking in core queues by re-reordering these requests based on the importance of request before issuing as per Figure 2.2. They fit hard real-time applications more as they do not require pre-requisites such as applying fixed priorities or minimum bandwidth before running [5].On the other side one of the main disadvantages of these fair scheduling techniques is being trivial approach to handle multiple tasks requirements without any privilege for highly risked tasks which can lead to degraded performance [9]. Instead, the AMC technique that is based on Round Robin concept (as explained in full details in section 3.1.1) requires neither knowing the bandwidth requirements, nor assigning a fixed priority to each task allowing AMC being applied to control based applications where the bandwidth requirements are not known. AMC is better suited for hard real-time applications, while CCSP arbitration is intended for streaming or multimedia real-time applications. [8]

2.2.2 Priority based scheduling memory controller

In this type[6][7], memory controllers run based on ranked cores through assigning priority among cores, and fair request queuing i.e. they treat all the requests form the same core on FCFS basis and rank requests dispatched from different cores. They are obviously used in memory intensive applications as multimedia real-time applications to handle different types of multimedia traffic as video conferencing.

2.3 Memory request process

Cores can only access the content of the row buffer and not the data in the memory array. Each request access part of the row by selecting some columns because the row size is large in modern DRAMs (multiple KB). In order to access a memory location, the row that contains the desired data needs to be loaded into the row buffer of the corresponding bank by an *Activate* (*ACT*) command as shown in Figure 2.3. If the controller needs to load a different row, the row buffer must first return back the old row to the mentioned bank in the array by a *Precharge* (*PRE*) command. By then, the new row can be loaded.

Each Load/Store command accesses data in *Burst Length* BL and the amount of data transferred will be ($BL \cdot W_{BUS}$), in which W_{BUS} is the width of the data bus. If the *BL* is 8 Bytes and W_{BUS} is 64 bits, the amount of data transferred for one request is 64 Bytes. Since DDR memory transfers data on the rise and the fall edges of clock, the amount of time for one transfer is *BL/*2 memory clock cycles i.e. four cycles, t_{BUS}, for a *BL* of eight.

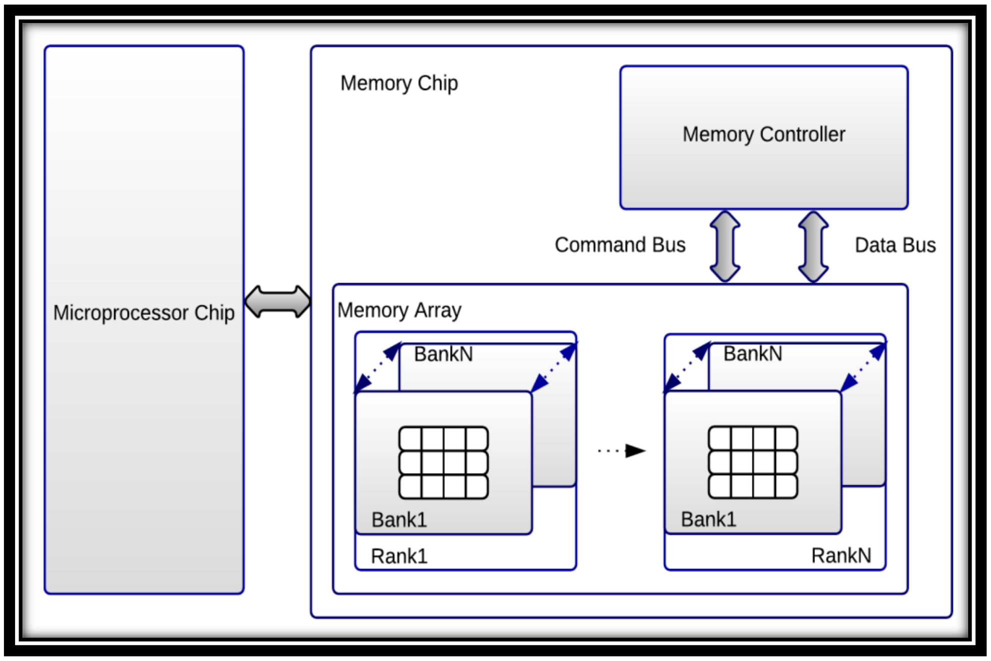


Figure 2.1 DDR DRAM Structure

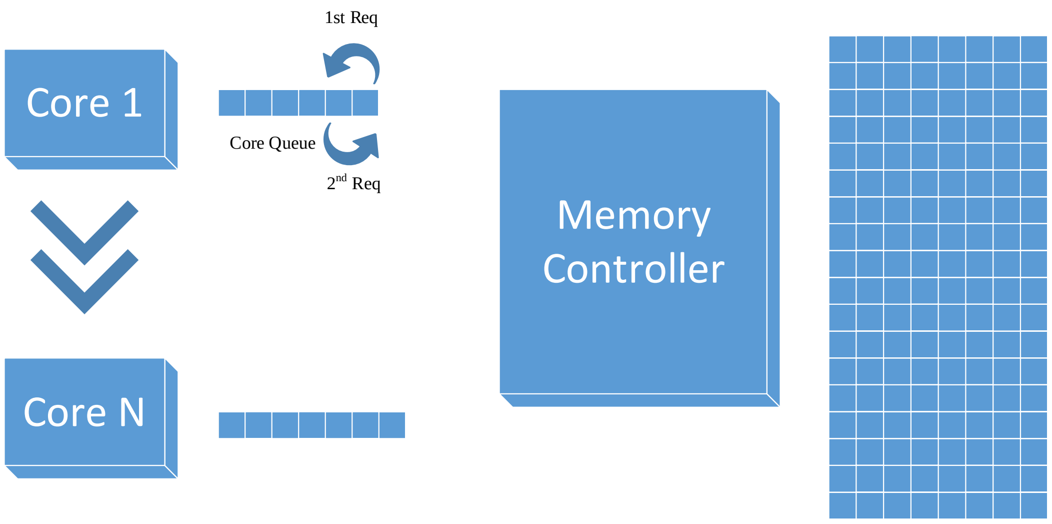


Figure 2.2 re-ordering requests in Memory Controller

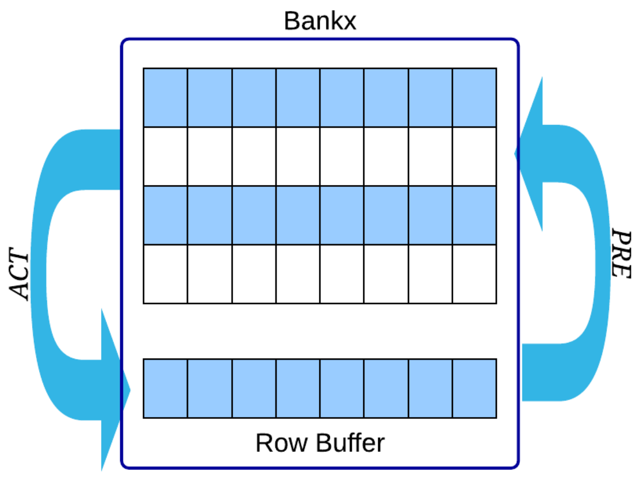


Figure 2.3 Row Buffer Loading

2.4 DDR Evolution

2.4.1 DRAM

The first version of these DRAM has an asynchronous interface with the system bus. This means that they will respond as quickly as possible to the control inputs. After that, the synchronous version on DRAM was invented and synchronized with the system bus. Both types are single data rate which means that they can accept one command and transfer one word of data per clock cycle. The Data buses ranges from 4 bits to 16 bits. All the commands occur on the rising edge of the clock cycle. Table 2.1 represents the control signals and the function of each signal. First of all, the memory needs the clock signal (CKE) enabled to perform any memory operation. In order to select a specific column to write or read the data, choose the bank, row, column by selecting BAn, RAS, CAS command bits respectively.

Table 2.1 DRAM Control Signals

Clock Signal	Function
CKE	All memory operations are continuous after this signal is high
CS	When this signal is high, this chip ignores all other inputs except the CKE
RAS	This is command bit used to determine the row selected
CAS	This command bit used to select the targeted column
WE	This bit distinguish the write command from the read commands
BAn	This bit/s determine the selected bank

2.2.4.2 DDRx DRAM

As the total request latency is limited and determined by DRAM access latency, so there was a need to increase bandwidth and decrease overall latency. Hence, a doubled data rate interface was developed. This interface can accept two reads and two writes on the rising and falling edges of the clock signal. On the other side, there were other modifications to decrease the consumed power. Table 2.2 shows the DDR DRAM rates [8].

Table 2.2 DRAM Specs

DRAM Standard Name	Memory Clock (MHz)	Cycle time (ns)	Peak transfer rate (MB/s)
DDR-200	100	10	1600
DDR-266	133.3	7.5	2133.3
DDR-333	166.67	6	2666.6
DDR-400A	200	5	3200
DDR-400B	200	5	3200
DDR-400C	200	5	3200

DDR2, as DDR, allows transmitting data on the rising and falling edges of the clock cycle. In addition, DDR2 operates on doubled bus speeds. This means that the DDR2 can operate on double speed of DDR which allow transferring four reads and four writes per clock cycle. DDR2 maximum size can reach 4 GB. Table 2.3 shows the DDR2 DRAM rates [10]

Table 2.3 DDR2 Specs

DRAM Standard Name	**Memory Clock (MHz)**	**Cycle time (ns)**	**Peak transfer rate (MB/s)**
DDR-400B	100	10	3200
DDR-400C	100	10	3200
DDR2-533B	133.3	7.5	4266.6
DDR2-533C	133.3	7.5	4266.6
DDR2-667C	166.6	6	5333.3
DDR2-667D	166.6	6	5333.3
DDR2-800C	200	5	6400
DDR2-800D	200	5	6400
DDR2-800E	200	5	6400
DDR2-1066E	266.6	3.75	8533.3
DDR2-1066F	266.6	3.75	8533.3

DDR3 continues to double the amount of data transferred to be eight times the speed i.e. DDR3 are able to transfer eight words (reads/writes) per clock cycle. DDR3 can support up to 4 ranks with 64 bits each to reach 64 GB. But for hardware limitations, most CPUs can handle from 4 GB to 16 GB. DDR3 consumes 30% fewer energy than DDR2 because of decreasing the supply voltage. Table 2.4 shows the DDR2 DRAM rates [11]

Table 2.4 DDR3 Specs

DRAM Standard Name	**Memory Clock (MHz)**	**Cycle time (ns)**	**Peak transfer rate (MB/s)**
DDR3-800D	100	10	6400
DDR3-800E	100	10	6400
DDR3-1066E	133.3	7.5	8533.3
DDR3-1066F	133.3	7.5	8533.3
DDR3-1066G	133.3	6	10666.6
DDR3-1333F	166.6	6	10666.6
DDR3-1333G	166.6	6	10666.6
DDR3-1333H	166.6	6	10666.6
DDR3-1333J	166.6	6	10666.6
DDR3-1600G	200	5	12800
DDR3-1600H	200	5	12800
DDR3-1600J	200	5	12800
DDR3-1600K	200	5	12800
DDR3-1866J	233.3	4.28	14933.3
DDR3-1866K	233.3	4.28	14933.3
DDR3-1866L	233.3	4.28	14933.3
DDR3-1866M	233.3	4.28	14933.3
DDR3-2133K	266.6	3.75	17066.6
DDR3-2133L	266.6	3.75	17066.6
DDR3-2133M	266.6	3.75	17066.6
DDR3-2133N	266.6	3.75	17066.6

DDR4 is the final DDR DRAM globally released. DDR4 operated on higher frequency ranges (800 - 1600) MHz and lower voltage reaches 1.2V

2.5 DDR Timing Constraints

DRAM device performs different operational tasks to handle requests. These tasks require timing constraints to be preserved through *MC*. All these timing constraints are defined by Joint Electron Device Engineering Council (JEDEC) standard [6]. This standard defines all DDR versions including DDR2, DDR3 and DDR4. Table 2.5 lists most of the used commands and their descriptions. Table 2.6 lists all values in memory cycles required for our analysis.

Table 2.5 DDR Timing Constraints description [11]

Timing Parameter	**Description**
T_{RCD}	This is RAS (Row Access Strobe) to CAS (Column Access Strobe) delay. Before issuing read/write request, the row must be first activated through activation command that is initiated by memory controller. Hence, the DDRAM moves the subjected row to the row buffer i.e. it is the clock cycles between the activation of a row read/write command to that row.
T_{RL}	This is read latency. It is the time required by the DRAM memory to start the data reading
T_{WL}	This is write latency. It is the time required by the DRAM memory to start the data writing
T_{BUS}	Amount of time required to transfer the data in the data bus channel
T_{RP}	This is RAS precharge after data transmission. If the memory controller receives another request targeting another row, the memory controller initiates a precharge command to return back the row from the row buffer. T_{RP} is the time required to precharge a row. i.e. it is the number of clock cycles elapsed between a row precharge and activation command in the same memory rank
T_{WR}	It is called write precharge delay or write recovery time. It represents the amount of time the memory spends after the completion of valid write operation and before the precharge command is issued.
T_{RP}	This is read to precharge delay. It is the number of cycles for a memory read command to the precharge command for the same memory rank.

T_{RAS}	This is activate to precharge latency. It is the time spent from the activation of a row till the precharge command.
T_{RC}	It is Row cycles. This is the cycles from the row activation passing by row precharging till the activation of another row. It is the summation of T_{RP} and T_{RAS}
T_{RRD}	This is activate to activate delay. It is the time elapsed between two row activate in different banks in the same rank.
T_{FAW}	This is the time window for four activation commands allowed simultaneously to the same rank. This is due to power limitation that feeds the charge pump on the chip [12].
T_{RTW}	Read Then Write delay. This is the time required to switch from read command to write command in the same rank.
T_{WTR}	Write Then Read Delay. It represents the delay required between the last valid write operation and the next read command.
T_{RFC}	This is the Refresh Cycle time. It represents the time measured from the refresh command (REF) to the first activate command.

Table 2.6 DDR Timing constraints values

Parameters	**DDR2 – SG25E**	**DDR3_32M-SG15**	**DDR3-SG25E**	**DDR3_64M-SG15**
t_{RCD}	5	10	5	9
t_{RL}	5	10	5	8
t_{WL}	4	9	4	7
t_{BUS}	4	4	4	4
t_{RP}	5	10	5	9
t_{WR}	6	10	6	10
t_{RTP}	3	5	4	5
t_{RAS}	18	24	15	24
t_{RC}	23	34	20	34
t_{RRD}	3	4	4	4
t_{FAW}	14	20	16	20
t_{RTW}	5	6	6	5
t_{WTR}	3	5	4	5
t_{RFC}	51	107	64	107

To illustrate these time constraints, Figures 2.4 and 2.5 show load/store request scenarios to different banks. Figure 2.4 shows timing constraints related to load request. 1st request is targeting *bank 1*. Request 1 is a load and it consists of ACT and read commands. 2nd request is a store targeting *bank 2* and it consists of ACT and write commands. Note that the write command of 2nd request cannot be issued immediately once the t_{RCD} timing constraint of *bank 2* has been satisfied. This is because there is another timing constraint, t_{RTW} between read commands of 1st request and write command of 2nd request. Hence, the write command can only be issued after timing constraints for this request are satisfied. Similar constraints are shown for a store request targeting bank 1 and load request in *bank 2* in Figure 2.5.

We can note here:

1) The latency for a close request is longer than an open request (which is described later in section 2.6.2). There are long timing constraints involved with PRE and ACT commands, which are not needed for open requests.

2) Switching from load to store requests and vice-versa incurs a timing penalty. There is a constraint t_{RTW} between issuing a read command and a successive write command. On the other side, the t_{WTR} constraint applies between the end of the data transmission for a write request and any successive read request.

3) Different requests can access different banks in parallel. There is no constraint such as t_{RTW} and t_{WTR} between two successive reads or two successive writes to different banks. Also, PRE and ACT commands to different banks can be issued in parallel as long as the t_{RRD} and t_{FAW} constraints are met.

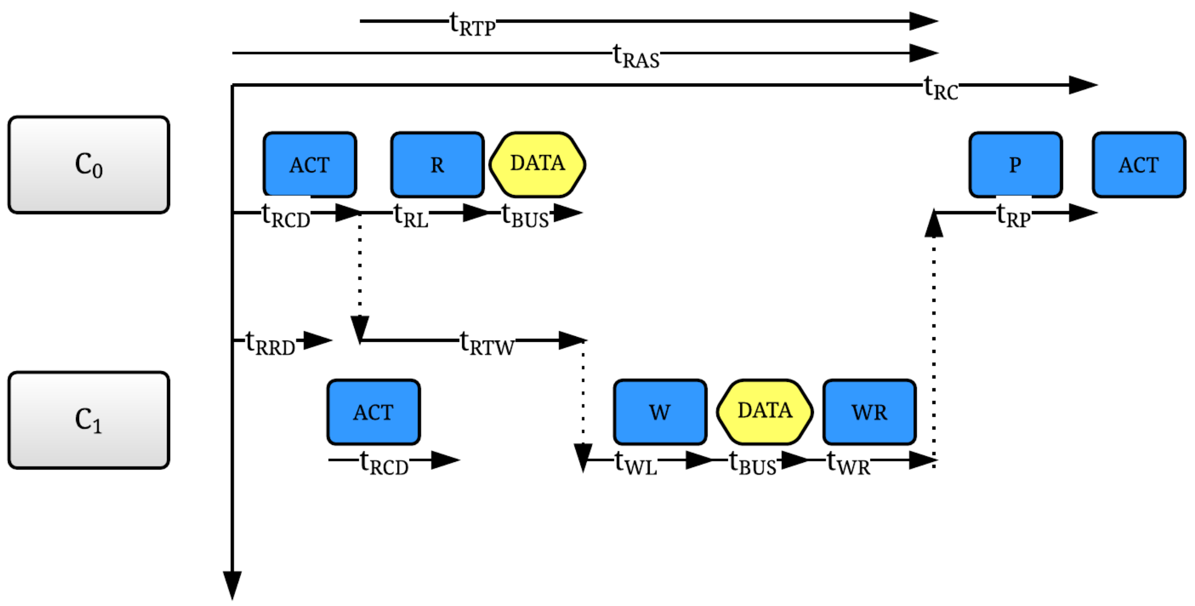


Figure 2.4 Load request Timing Constraints

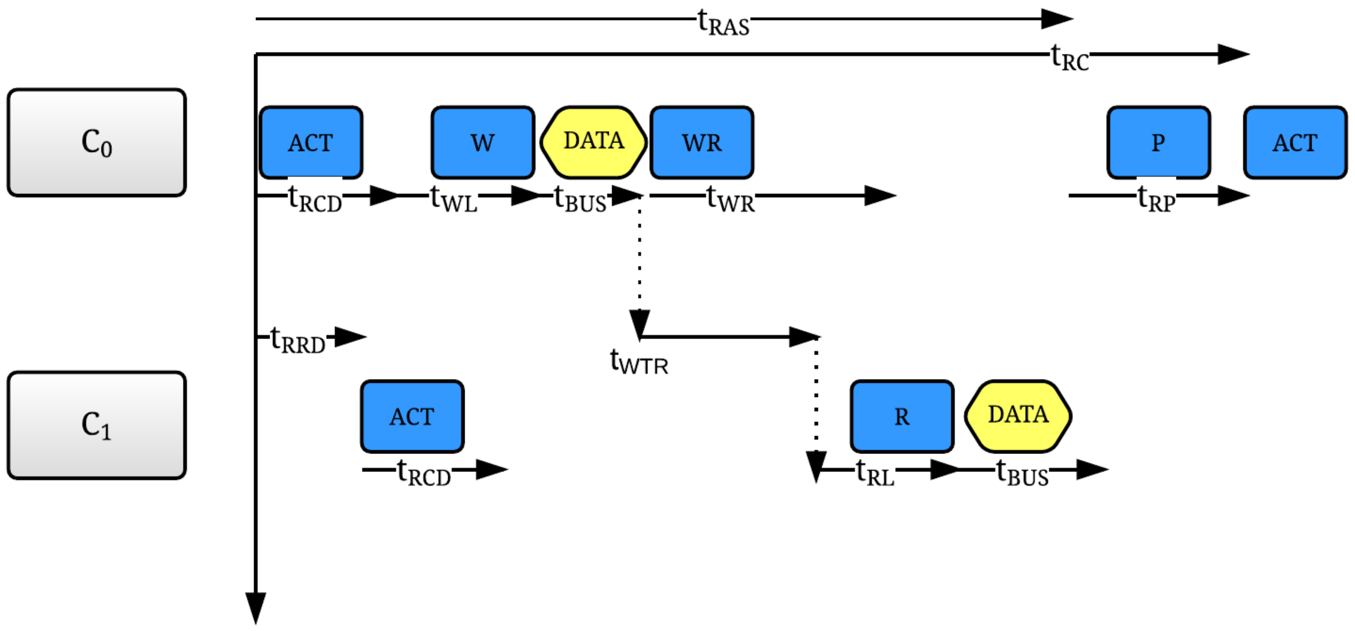


Figure 2.5 Store request timing constraints

2.6 DDR Requests Mapping and Row policy

2.6.1 Interleaved vs Private Banks

In general, any memory controller has two ways to map data to the banks: *Interleaved* Banks and *Private* Banks [8]. *Interleaved* banks allow each core to access all banks in parallel. Meanwhile, the other cores can access all banks concurrently, which will lead to interference among the cores. The amount of data transferred in one request is $BL.W_{BUS}.NUM\_BANKS$. E.g. if BL is 8 bytes and W_{BUS} is 64 bits, then the amount of data transferred is 256 bytes as shown in Figure 2.6. The other request map, *Private* Banks map one bank or more to a core so that we can eliminate the interference across the cores as shown in Figure 2.7. For our approach, we will adopt the private bank methodology.

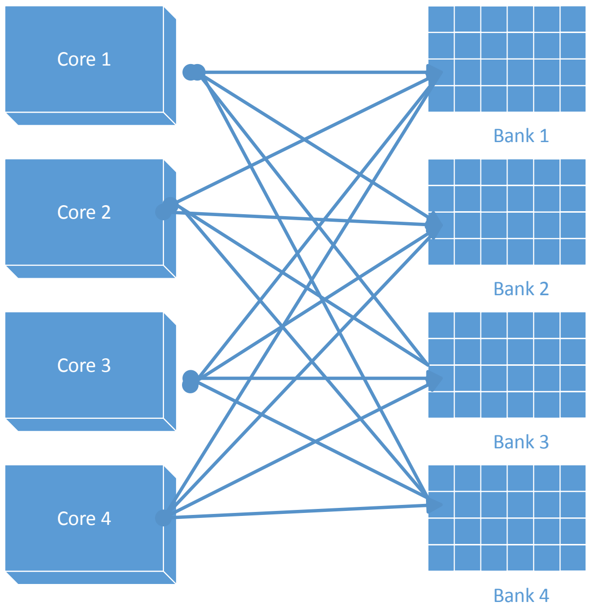


Figure 2.6 Interleaved banks for 4 banks DDR DRAM

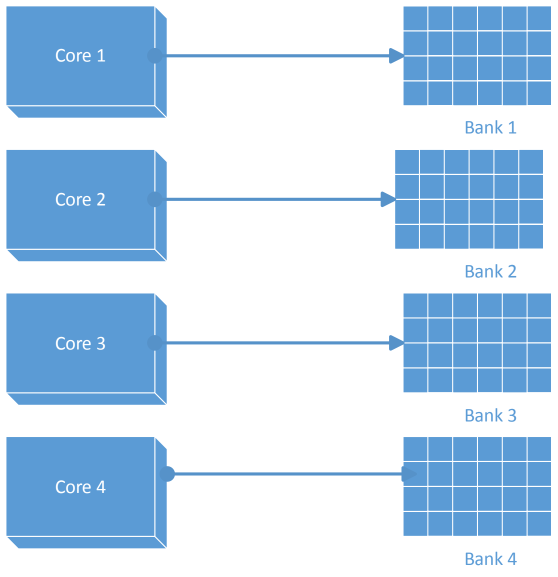


Figure 2.7 Private Banks for 4 banks DDR DRAM

2.6.2 Close Page vs Open Page Policies

Regarding the row policies, there are two row buffer policies: *Open Page* (*OP)* and *Close Page (CP)* policies [8]. For the *OP* Policy, the request consists of a *read* or a *write* command. These commands are carried out immediately, if there is no other request uses this row and the mentioned row is already cached in the row buffer, or the refresh period does not take place. Hence, total latency decreases. On contrary, if the above mentioned conditions are not satisfied, then the row buffer is called a *row miss*. By then, the row must first be written back to the array by a PRE command. After that, an ACT command loads the desired row into the row buffer and *read/write* commands can be issued to access the data.

On the other side, the *CP* Policy will auto-precharge the row after the *read/write* command is issued. *CP* can be considered an advantage while pipelining the requests through *interleaved* banks because all the read and write commands will run in parallel and their interference can be reduced using this policy [5]. The drawback for *CP*, while using *private* banks, is that the total latency will be increased for all tasks issued by *MC* especially in case *row hit* is zero. Hence our approach, *MCES*, uses OP to improve the cumulative latency by exploiting the DRAM *row hit* cases.

2.7 In-order vs Out Of Order Cores

MCES uses Out Of Order (*OoO*) [13] cores that are capable of dispatching multiple instructions to set of queues per cycle. *OoO* cores avoid the stall that occurs in the in-order core when the instruction is not completely ready to be processed due to missing data. *OoO* processors fill these "slots" in time with other instructions that are ready, then re-order the results at the end to make it appear that the instructions were processed as normal as shown in Figure 2.8. However, *MC* will serve one request each memory cycle due to the shared resources. The other type of cores is in-order core where instructions are fetched, executed in generated order. If a request stalls, all requests queued in buffers will stall waiting for the mentioned request to finish as shown in Figure 2.9. It is worth mentioning here that the focus is not on modelling cores. Hence, we will not study here the utilization of cores or its contribution in calculating WCET. Hence, our focus is on the design of *MC*.

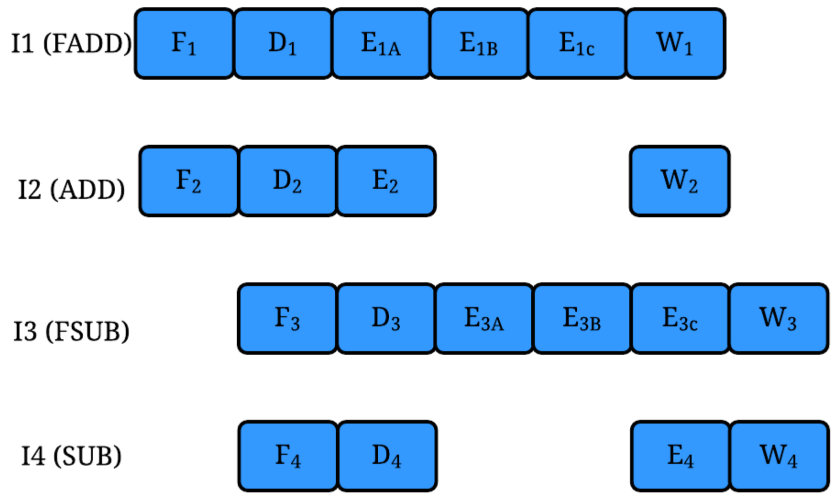


Figure 2.8 OoO execution pipeline

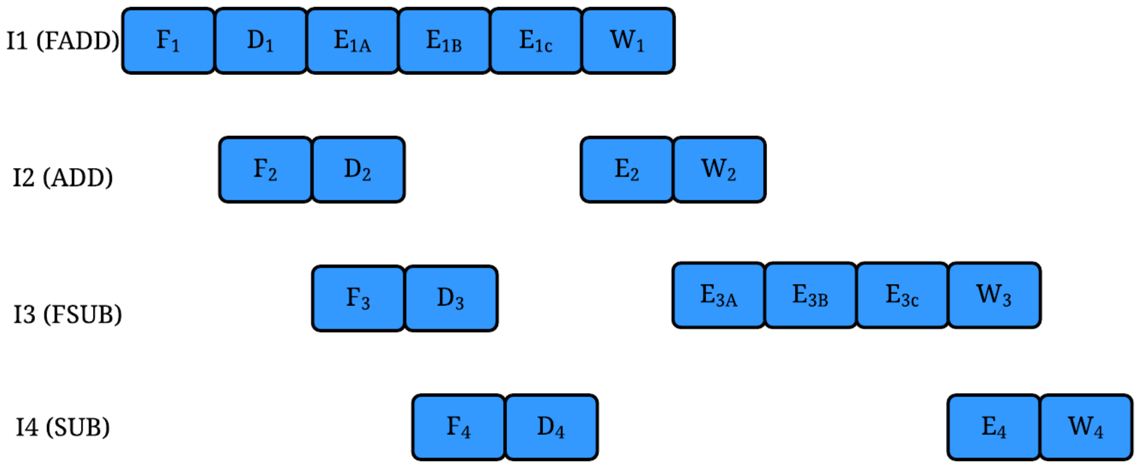


Figure 2.9 In-order core pipeline

Chapter 3. Related Work

In this chapter, we will discuss the related work. Similar memory controllers have been proposed.

3.1 Fair Scheduling Memory Controllers

This category as we mentioned before in chapter 2 fairly schedules the cores' requests. Hence, all requests arrived from different cores and reached the final queue will be scheduled on fairly basis either using Round Robin or First Come First Serve (FCFS). On the other hand, memory controllers apply ranking on requests during queuing in buffers before scheduling.

3.1.1 Analyzable memory controller (*AMC*)

AMC [5] is memory controller that provides an upper-bound latency for hard and non-hard memory requests in a multi-core system using bank interleaving with closed page policy alongside with fair round-robin arbiter. *AMC* prioritize HRT requests over NHRT requests. If all the requests are HRT, so the AMC schedules them based on fair RR to prevent any timing anomalies from happening to the tasks. *AMC* uses one queue per core to isolate inter-task interference from intra-task interference. Hence, the maximum latency a request can suffer from will depend on the number of cores. *AMC* fits better than the priority scheduling memory controllers for hard real-time applications as their solution can run on any hard real-time applications without defining parameters as priorities, minimum bandwidth. On the other hand, *AMC* RR policy with one request per core will not satisfy the different core requirements. Hence, if more than one request is assigned per core, WCET will be degraded severely. Also, assuming fixed Read/Write or Write/Read maximum latency cannot produce precise latencies as we will show later in 4.3.3.

3.1.2 Worst Case Analysis of DRAM Latency (*WCAD*)

WCAD [8] controller extends the AMC proposal, yet via using *Private* Banks and *OP* policy. *WCAD* uses a queue for each core and enqueues only one request per core to a global FIFO queue. The arbitration rule of the global queue is based on issuing one request only from each core based on round robin (RR) arbitration and not to enqueue any other request until the data of the previous request is transmitted. Yet the drawbacks here in this proposal*:*

1- *WCAD, as AMC,* assumes equal importance of applications as it does not consider prioritization in scheduling regardless how much the applications are memory intensive. This means that using memory intensive applications will lead to degradation in performance.
2- *WCAD* cannot distinguish the severity of tasks as hard and non-hard real-time tasks.
3- *WCAD* focuses on in-order cores although their algorithm performance degrades while using *OoO* cores as we will show later in chapter 6.

3.2 Priority Scheduling Memory controllers

Priority Memory controllers use offline fixed assigned priorities to cores based on offline analysis for the application running on these cores and bandwidths needed to accomplish their tasks. Yet, they do not apply request ranking in buffers before scheduling these tasks.

3.2.1 A Predictable SDRAM Memory Controller (*PREDATOR*)

Predator [6] ,as *AMC*, uses bank interleaving with closed page policy yet assigns fixed priority to requests to guarantee minimum bandwidth and consequently maximum bounded latency to the cores requests. This is accomplished by defining two steps approach to predict DDR2 access. First, Predator defines statically read and write requests to SDRAM memory and this will determine the lower net bandwidth required. Then, scheduling using Credit Controlled Static Priority (*CCSP*) which is composed of rate regulator and a scheduler. The rate regulator is responsible of assigning bandwidth to each core. After that the static priority scheduler is responsible to provide maximum latency bound for the requests. This algorithm becomes more effective for multimedia real-time applications. On the other hand, PREDATOR cannot be used on hard real time applications as *WCET* bounded calculations cannot be estimated if cores received hard real-time requests.

3.2.2 Bounding WCET Using SDRAM with Priority Scheduling (BWPS)

This algorithm BWPS [7] extends *Predator* to work with priority-based scheduling (*PBS*) arbiter to provide WCET, Observed Execution Time (OET) and the Best Case Execution Time (BCET) analysis of the application running on the proposed architecture. The use of OET and BCET to evaluate the precision of the algorithm results and the variability of the *PBS* arbitration scheme. *PBS* assigns priority and budget to cores at design time. Cores with high priority has the low budget and cores with low priority gains high budget. Pros and cons are to be discussed in section 3.4.

3.2.3 SDRAM controller Traffic with complex QOS requirements

Authors in [14], proposed i*nterleaved* bank memory controller alongside with round robin arbiter to provide maximum latency for cores. Authors suggested 2-stage scheduling algorithm and static priorities assigned to achieve the required QOS which includes guaranteed minimum throughput at guaranteed maximum latency and smallest possible latency. Hence, cores are assigned high or standard priorities. After each high priority request is issued, it is followed by standard priority request to prevent starvation. This algorithm did not exploit the core priorities well and shared the same issues of interleaved banks and round robin as mentioned above.

3.3 Non Real-time Memory controllers

The below subsection discusses other algorithms for different designs.

3.3.1 Predictable Memory Controller Performance in Many-Core CMPs

Author as in [15], applied multiple memory controller designs to multi-core system using a mesh or torus topologies for interconnection to show how memory controller location and routing algorithm used can improve the latency and bandwidth characteristics regardless of the processor used. It is worth to mention that this type can fit Big Data or cloud computing solutions because of their huge and powerful amount of data but not suitable for embedded systems which need to achieve compromising levels of high performance and low power consumption.

3.4 WCET enhancement techniques

The next part will discuss other ways to analyze other designs to enhance WCET estimation and calculation.

3.4.1 Hardware WCET Analysis of Hard Real-Time Multicore Systems

Authors in [16] discusses hardware shared resources as bus and cache interference. The authors propose bus arbiter to control bus interference when computing WCET and be able to compute the maximum bounded latency a bus thread can suffer from. The proposal splits the bus arbiter into 2 bus components. Inter-Core Bus Arbiter (XCBA) that schedules requests from different cores, and multi Intra-Core Bus Arbiters (ICBAs) that schedules requests from the same core. The *NHRT* requests are send to the corresponding core's ICBA to select the next memory request to be sent to *XCBA* based on *FRFS*. This *FRFS* will issue all requests out of order that targets different banks to increase the overall performance. Regarding *HRT* requests, the arbiter issue the requests based on FIFO to provide any timing anomalies. In mixed overload, if *HRT* request needs the bus and it is being used by *NHRT* request, so the arbiter will serve the *HRT* request immediately to increase any extra delay can be added to its execution time. The disadvantage is that the arbiter will retry to issue the *NHRT* request next which consumes more energy.

3.4.2 Interference Aware Allocation Algorithm (IA^3) for multicore hard real-time systems

The authors in [17] proposes IA^3 which is a new offline algorithm that uses set of WCET estimations mapped to all execution environment for the task to run. Execution environment is the resources the task will be assigned to as the cores, scheduling algorithm, and bus arbitration policy and cache partitions. Their algorithm introduces 2 concepts: WCET matrix and WCET sensitivity. The WCET matrix is n-dimensional vector for *HRT* task where each dimension determines the different execution environments parameters that may affect the WCET for this *HRT* task i.e. WCET matrix for a specific task is the collection of WCET estimations for this task running on a processor under different execution environments.

WCET sensitivity allows the IA^3 to be aware of the impact of changing the execution environment on the WCET estimation. These two concepts will allow the IA^3 algorithm to define the allocated resources (number of cores, cache partitions assigned to each core, etc..) assigned to each task to run efficiently as being used in avionics and space systems. WCET sensitivity allows tasks with higher demand to be allocated first.

IA^3 algorithm can run in 2 different modes:

1- Common mode where the same exaction environment will be applied on all cores.
2- WCET Sensitivity mode, where different execution environments will be applied on different cores.

The main drawbacks of this algorithm is that IA^3 is off-line algorithm and needs high computation to fulfill the matrices before running. It can be used in high risk applications as military or space systems.

3.4 Summary of Scheduling algorithms

To summarize all the above discussed memory controller algorithms

1- Algorithms in [5], [6], [7], and [14] used *interleaved* banks so there can be interference between cores with C*P* policy which affects total latency by allowing auto (*ACT+PRE*) commands and pretending the *row hit* equals zero.

2- As mentioned in [8], these algorithms exploit the caching mechanism in DRAM devices. As modern DRAM devices become faster, performance of these *MCs* degrade severely because the access time between cached devices and non-cached in DRAM is growing.
3- *WCAD* [8], uses fair FIFO scheduling and in-order cores so by applying intensive applications on multiple cores, i.e. core receives more than one request per cycle, and the performance degrades severely.
4- Other memory controllers [15] is used for cloud computing applications and cannot be implemented here because it depends on very high numbers of cores and routing techniques cannot be implemented in embedded designs
5- *WCET* enhancement techniques are using other layers as hardware layer by implementing bus arbiters and matrix algorithm to define the best paths to consume the better WCET. These techniques can help in future implementations.

After discussing all the above memory controllers, it is observed that each scheduling type is targeting one category of applications and we believe that by taking the advantages of those techniques in one memory controller can achieve better performance results and be applied on different application categories.

We are interested in memory controller design for embedded multi-core systems to optimize shared resources allocation. Hence, *MCES,* our approach, is memory controller that aims to apply request ranking within the same core and core prioritization among cores by exploiting the two methodologies, namely *Private* Banks and *OP* policy. MCES should eliminate row interferences from different cores, since each core can only access its own memory banks and its allocated private buffer. MCES compares the achieved results with WCAD as they are the most recent work and the results achieved by their algorithm are better than any previous algorithm.

Chapter 4. Real-time Memory Controller for Embedded Systems (MCES)

4.1 System Architecture

In this chapter, we will discuss our algorithm for memory controller design. The architecture of our system is shown in Figure 4.1. We applied our algorithm using OoO cores. Each core has separate L1 instruction and data caches, followed by L2 unified cache. *MCES* is priority-based memory controller as it takes into account the effect of multimedia real-time applications and request ranking required for hard real-time applications. MCES Enables fair scheduling using round robin (RR) arbiter within cores when the starvation flag is up. Each core stores the memory requests (*read/write*) in private buffers based on *FCFS* method, unless otherwise there are different types of requests arrive in the private buffer in an unordered way i.e. HRT arrived the private buffer after NHRT request. These memory requests are then enqueued to the global queue based on the quota/budget assigned to each core. Our queuing and reordering flowchart to re-order the requests in private buffers is shown in Figure 4.2 and scheduling algorithm for requests after reaching Global Queue is found in Figure 4.3. From the figure, the MCES checks the priority of the requests and their timing constraints. If the request meets these criteria, the request will be issued to memory. Otherwise, starvation flag decrements and check the lower priority requests to serve if the starvation flag reaches zero and still has remaining quota to issue. Each memory cycle MCES checks the Memory Time Out (MOT) parameter to refresh quotas of all cores if the MOT becomes zero.

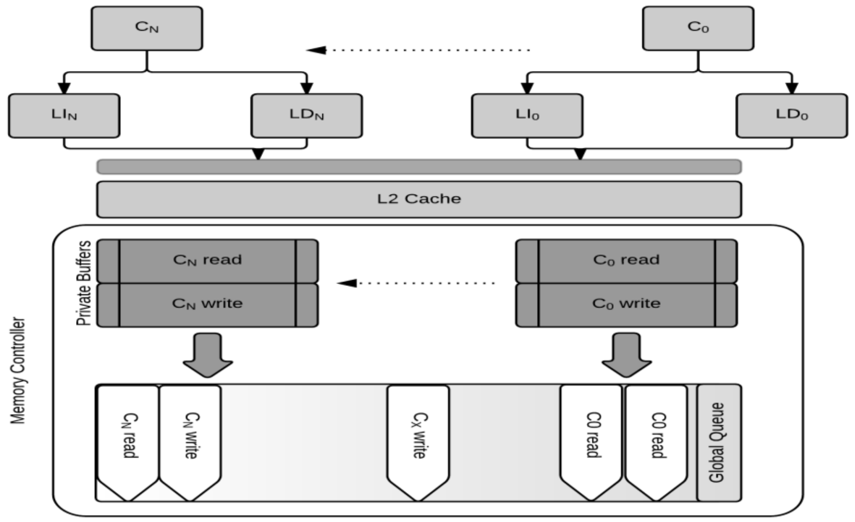


Figure 4.1 MCES architecture

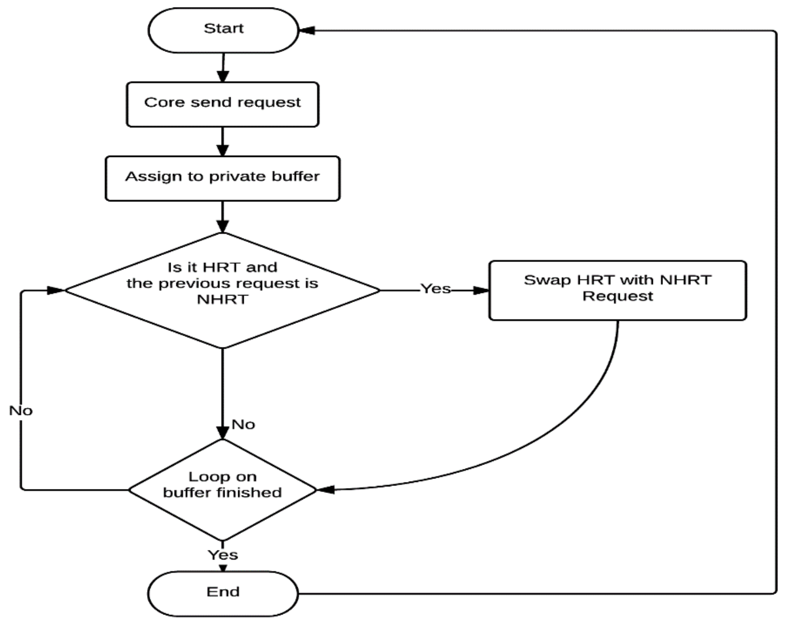


Figure 4.2 Queuing and Re-ordering

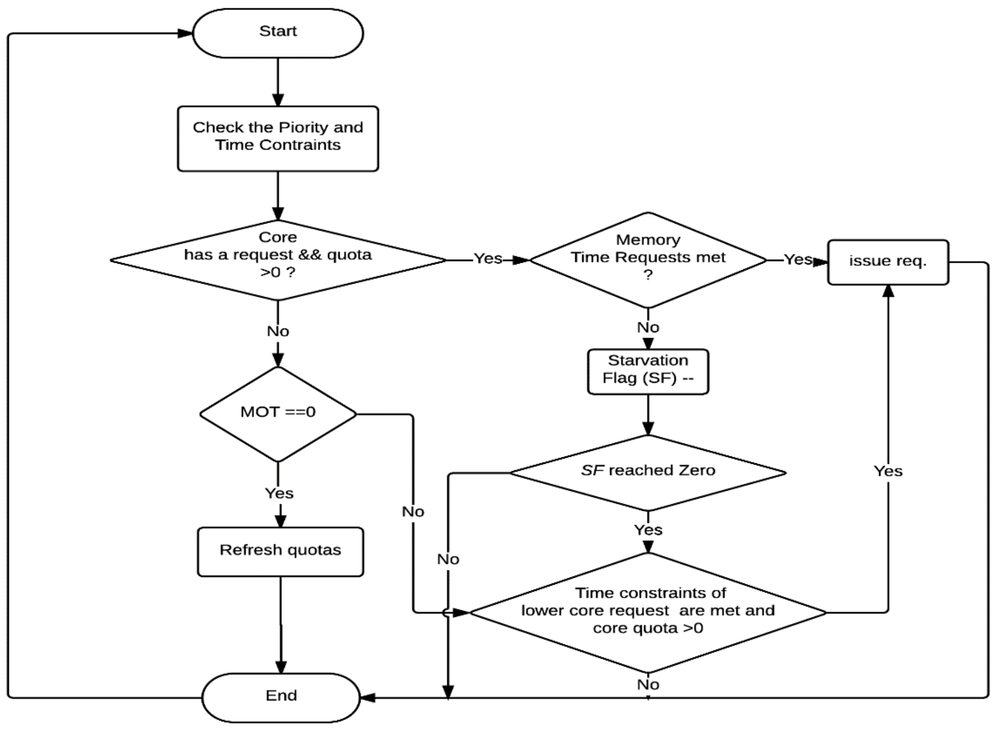

Start
Check the Piority and Time Contraints
Core has a request && quota >0 ?
Yes
Memory Time Requests met ?
Yes
issue req.
No
No
Starvation Flag (SF) --
MOT ==0
SF reached Zero
Yes
Yes
No
Yes
No
Refresh quotas
Time constraints of lower core request are met and core quota >0
No
End

Figure 4.3 Scheduling Flow chart

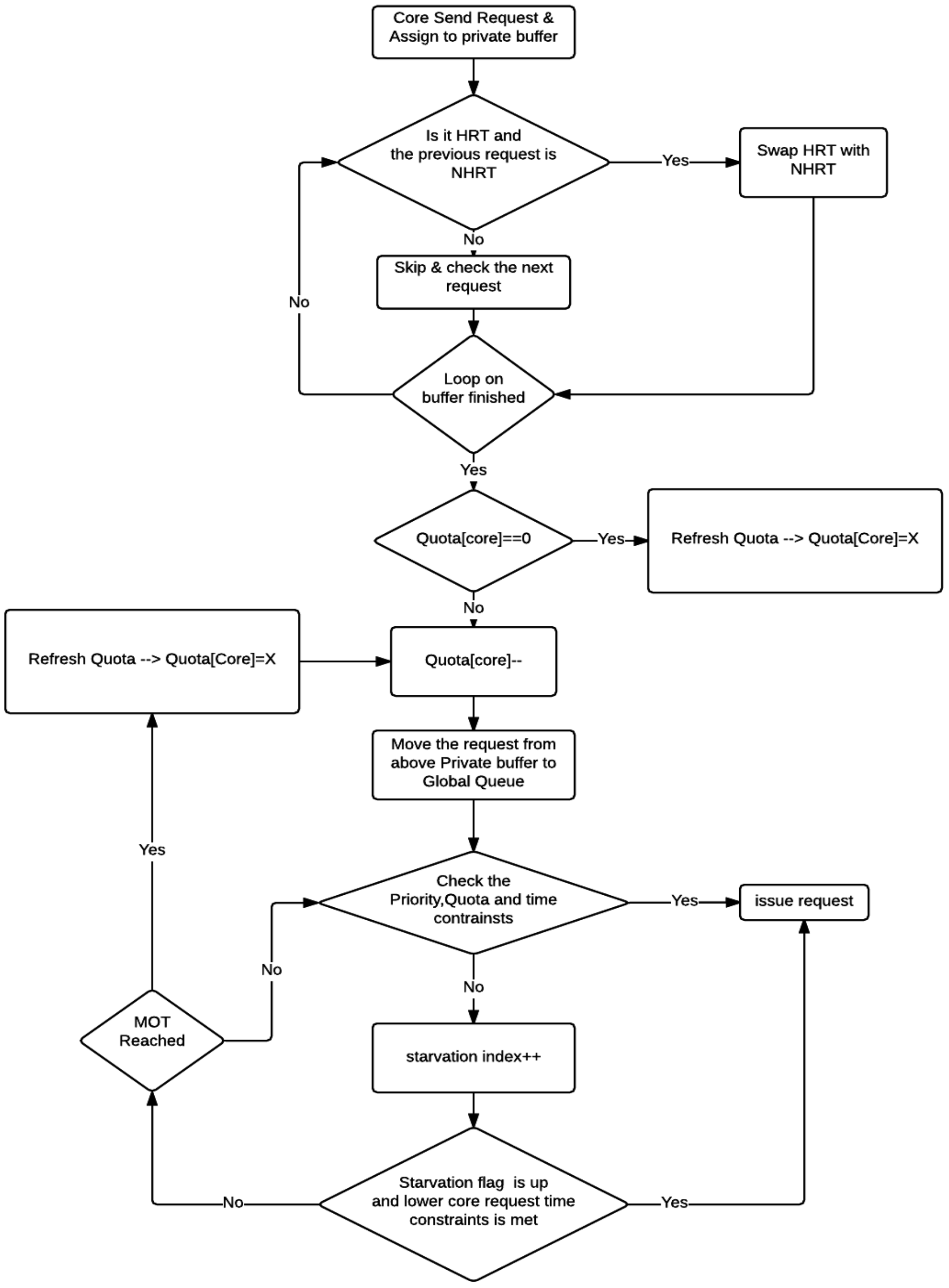

Core Send Request & Assign to private buffer
Is it HRT and the previous request is NHRT
Yes
Swap HRT with NHRT
No
Skip & check the next request
No
Loop on buffer finished
Yes
Quota[core]==0
Yes
Refresh Quota --> Quota[Core]=X
No
Refresh Quota --> Quota[Core]=X
Quota[core]--
Move the request from above Private buffer to Global Queue
Yes
Check the Priority,Quota and time contrainsts
Yes
issue request
No
No
MOT Reached
starvation index++
Starvation flag is up and lower core request time constraints is met
No
Yes

Figure 4.4 MCES Algorithm flow chart

4.2 Arbitration Rules

Memory requests are enqueued to global queue and scheduled based on the below arbitration rules as follow:

1- Each core will have priority given to the core through OS, hence a corresponding quota for the number of requests allowed to be queued in the global queue. The core with high priority will gain high quota and so on.

2- Each core will be able to dispatch only requests within the assigned quota. Hence, the size of the global queue will be limited to the summation of the quotas of all cores.

3- The core consumes its quota after issuing its requests from the global queue to memory. The core will not be able to issue any other request until the epoch period is passed or the global queue issues all the requests. This epoch is called M*emory Request Timeout (MOT).*

4- *A)* The requests issued per cores from the global queue to memory in a RR fashion.
B) At the start of every memory cycle, the queue is scanned and request is issued from the highest priority core hence consequently the lower priority core, unless otherwise the starvation flag is up. This flag switches between cores prevent request of low priority cores from starvation.

5- Request ranking is also considered in ordering the requests within each core. The core prioritizes *HRT* over *NHRT* through re-ordering the requests in the private buffer. Hence, each core has two private buffers based on the type of requests (*read/write*) to apply this re-ordering. Also the type of the request is watched for each core dispatches the read requests first followed by the write requests.

6- To eliminate the inter-task interference, every core has its private bank and its own private buffer.

7- Another level of prioritization considers the similar type of requests (*read/write*) regardless of the core priority. This rule forces the controller to issue the similar requests even it was from lower priority cores before the high priority core request/s.

To clarify Rule-5, we serve every *read/write* request based on its time constraint. Also we focus to serve the hard real-time tasks (*HRTs*) to improve the overall WCET. Hence, we apply the re-order technique in the private buffers, so that the *HRT* request will be issued first. If all the requests reached the private buffer are *HRTs*, then they will be served on FCFS basis .In Figure 4.5, the state of the private buffer is at the initial time

t. Let us consider the following scenario. One hard real-time read task (R_H) arrived the private buffer after two non-hard real-time read tasks (R_N). Without re-ordering, we can figure out that the *HRT* request will be served at t` which increases the latency of this request although after re-ordering in Figure 4.6, it will be served first at t.

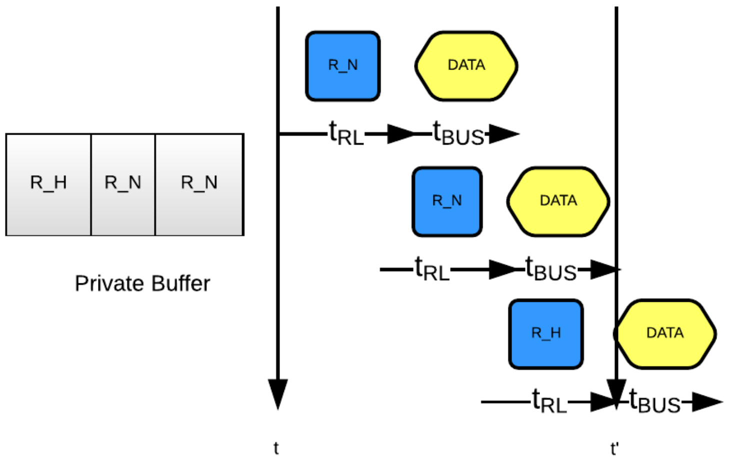


Figure 4.5 FCFS Queuing

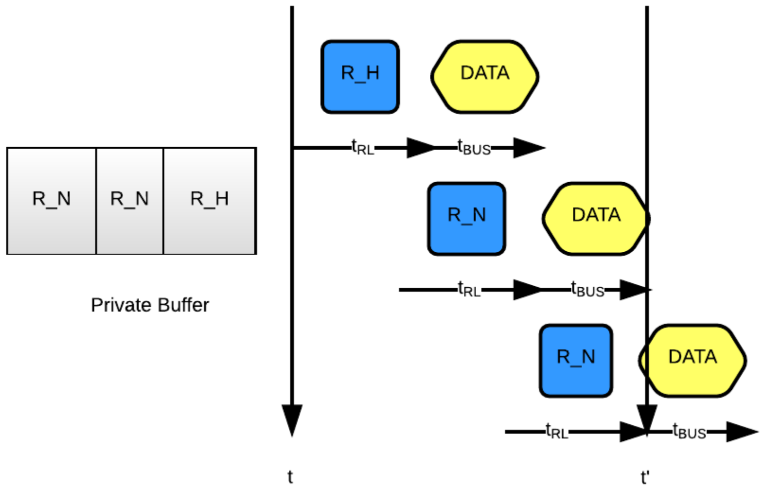


Figure 4.7 MCES Queuing

Rule-7 determines our algorithm technique that improves the overall performance. We derive a new way to prioritize requests based on similar request types, regardless of the priorities of the cores. Let us consider the following scenario for four cores (C_0, C_1, C_2, and C_3) in Figure 4.7 for *FCFS* scheduling:

1- The global queue received 4 simultaneous requests at time t, *read* from C_0, *write* request from C_1 and *read* requests from C_2 & C_3.

2- The queue will issue the *read* request from C_0 which has the highest priority (R_0). After that the *write* request (W_1) will be issued at t` after applying the time constraints *Read Then Write* (t_{RTW}).

3- Although C_2 & C_3 *read* requests (R_2 & R_3) are waiting the previous C_0 and C_1 requests to be served, but they will be issued at the end since they are the lower priority cores and will be served after time delay *Write Then Read* (t_{WTR}).

4- In *MCES*, Figure 4.8, the arbiter will switch to C_2 & C_3 after serving the *read* request from C_0 (R_0). Hence, R_2 & R_3 will be served after R_0 and finally W_1 will be served next at t`.

5- As per Figure 4.8, the re-ordered requests leads to a compromise between cores by shrinking the time required to issue all 4 requests. Hence, the overall performance increases.

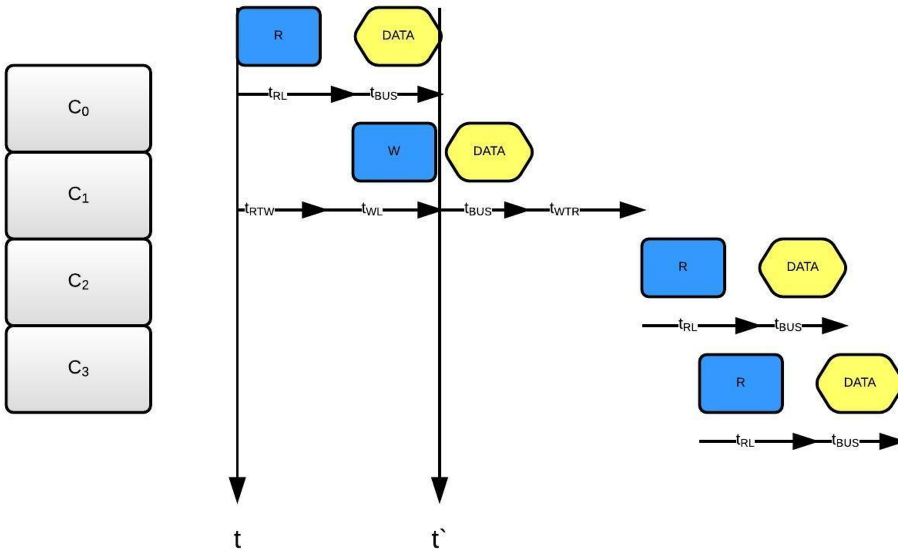


Figure 4.7 FCFS Scheduling

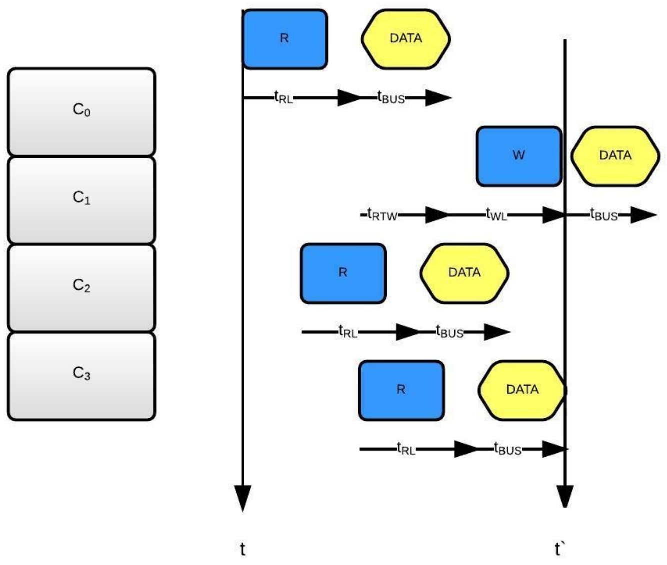


Figure 4.8 MCES Scheduling

4.3 Hardware / Area Overhead analysis

In this section, we will discuss the hardware needed to implement the MCES algorithm. As shared memory resources is limited between higher level memories as caches and lower level memory which is the DRAM. Hence, MCES needs a register to hold the bits of the Core ID which help to detect the Core's request sent from cache. This register is called Core Identifier Register (CIR) and it will be of max four bits in case of 16 cores , three bits in case of eight cores, two bits in case of four cores and one bit only in case of two cores. After that the data transactions (read/write) that should be transferred to private buffers as shown in Figure 4.1. Hence, the private buffers size per core is the summation of number of requests that can be held by the private buffers multiplied by the size of a single request $N_{private\ buffers} * \sum_{i=0}^{i=N} request * \ log_2 request$. In order to apply the re-ordering rule in private buffers, temporary register is needed to hold the replaced transaction data till the replacement is completed called *Hard to Non-hard Register* (HNR). HNR register size is one transaction request. Finally the global queue GQ size which will hold all the data ready to be issue. Hence, GQ size will depend on summation of the total quotas assigned to cores $\sum_{core=0}^{core=N} Quota_{core}$. Table 4.1 shows the total size in bytes required by MCES to be implemented for 8 cores. Each core has 2 private buffers for total 16 buffers. Each buffer holds 128 requests and each request is 64 bytes because data bus channel is 64 bits and we use BL equal to eight so bytes Per Transaction = (JEDEC_DATA_BUS_BITS*BL)/8 . Hence, total private buffer size is 2048000 bytes. To determine GQ size, we need to collect the core quota for each core. Let's assume {C_0,C_1,C_2,C_3,C_4,C_5,C_6,C_7} quotas are {9,8,7,6,5,4,3,2},hence GQ size is

2816 bytes with total size equal to 132 KB. On the other side, WCAD MC total size for eight cores is 129 KB

Table 4.1 MCES Estimated hardware size

Core Number	Total private buffer number	Transaction size	Private buffer size (KB)	HNR Register size (Bytes)	CIR Size (Bytes)	Global Queue Size (KB)	Total size (KB)	WCAD MC Size (KB)
4	8	64	64	64	1	1	66	65
8	16	64	128	64	1	3	132	129
16	32	64	256	64	1	9	266	257

4.4 WCET Analytical Model

In this section, we are going to study the WCET for requests under analysis. For simplicity, let's assume each core has one request in its private buffer and this request is forwarded to global queue. We consider all the requests arrived at the same time t_0 as a worst case conflict among the requests to show the longest waiting time in the queue. Let's assume the lowest priority core reached global queue below t_0 by delta. This request will interfere with other requests with higher priorities, so we calculate the *WCET* for this request. We are running our *WCET* calculation on four core numbers (C_N=4) based on the following specifications. Priority of cores will be assigned decreasingly ($C_0 \rightarrow C_3$) = (highest priority$\rightarrow$lowest priority). Hence, based on the core priority distribution, quotas will be assigned decreasingly ($C_0 \rightarrow C_3$) = (highest quota$\rightarrow$lowest quota). This calculation will deal with the two different kinds of commands. First we will check the open request (*OR*). *OR* will deal with requests that targets open rows which are already loaded in the row buffer. After that, we will check closed request (*CR*). *CR* command ensures all requests from cores are targeting different rows, hence continuous *ACT* and *PRE* timing constraints will be added.

4.3.1 Open Request (*OR*)

For open requests, all rows are already open, and loaded to banks' row buffer. To calculate Open Request latency (T_{OR}), we will depend on the previously and concurrently issued requests of the cores under analysis. If the previous and current requests were of the same type (*read/write*), then T_{OR} will be zero because no timing constraints will be applied here. If the previous and current requests were interleaved, i.e. *read* request followed by *write* requests or vice versa, then the latencies should be considered due to the command line timing constraints for the memory DRAM IO gates which consumes time to switch from read command to write (t_{RTW}) and from write command to read (t_{WTR}). We assume that all requests reached global queue at the same time t_0 and the lowest priority core send its request before t_0 by delta. This request is served after the high priority cores issue their requests. Hence, this request suffers from delay. We calculate the *WCET* for this request. From Figure 4.9 and 4.10, it is obvious that read request needs t_{RL} to issue and write request needs entails t_{WL} to issue. Each read request issued after write request suffers from write then read timing constraint (t_{WTR}) and each write issued after read request suffers from read then write constraint (t_{RTW}). On the other side, for most DRAM memories the summation of t_{RL} and the data bus time t_{BUS} is lower than the summation of t_{WL} and t_{RTW} ($t_{RL}+t_{BUS} < t_{WL}+t_{RTW}$) so we can conclude that each read request to be issued suffers from ($t_{WL}+ t_{BUS} +t_{WTR}$) and each write request suffers from t_{RTW}. Hence the Worst Case Open Request (*WCOR*) between t_0 and t` are derived for *read* and *write* requests as shown in eq.1 and eq.2 respectively.

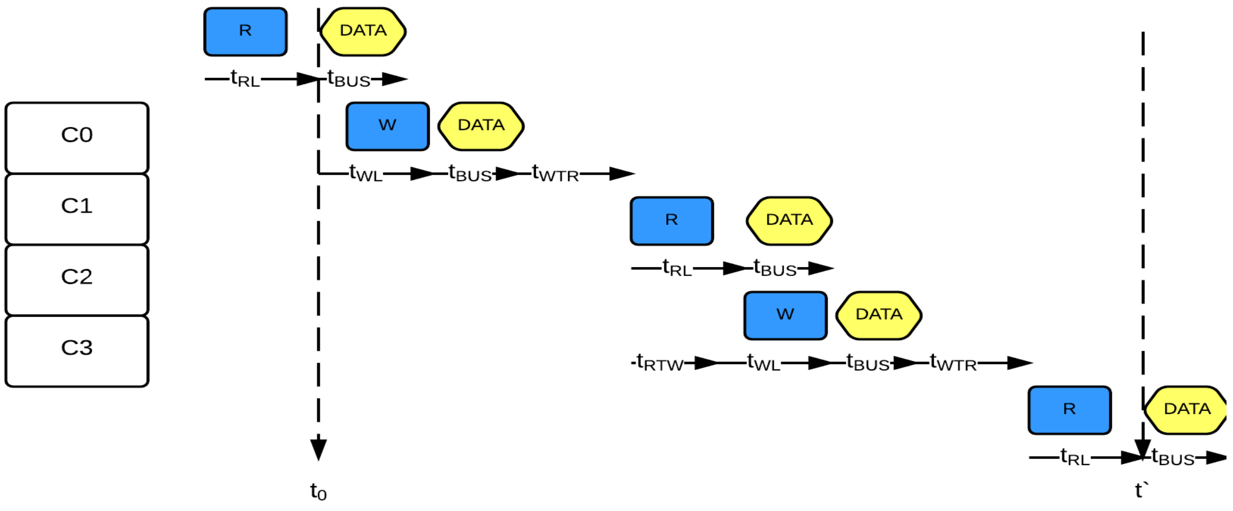


Figure 4.9 *WCOR* Read

$$T_{OR}^{read} = \frac{C_N}{2}(t_{WL} + t_{BUS} + t_{WTR} + t_{RL}) + \left(\frac{C_N}{2} - 1\right) t_{RTW} \qquad (1)$$

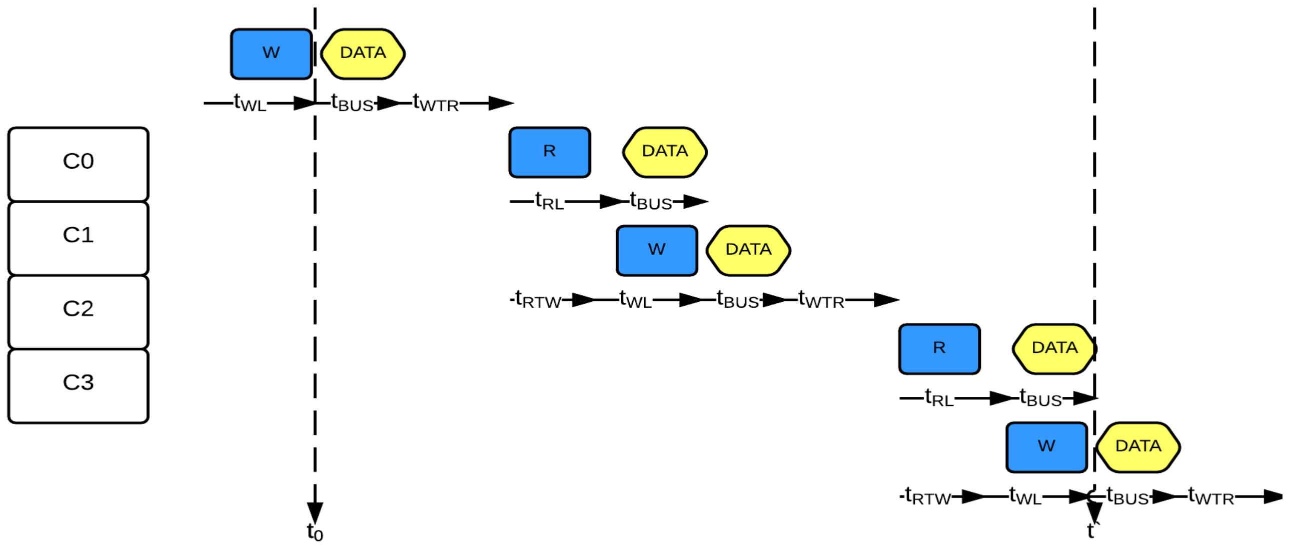


Figure 4.10 *WCOR* Write

$$T_{OP}^{write} = \frac{C_N}{2}(t_{BUS} + t_{WTR} + t_{RTW}) + (C_N - 1)t_{WL} \qquad (2)$$

4.3.2 Closed Request *(CR)*

In Closed Request *CR*, *ACT & PRE* will be added to command timing constraints because each request is targeting different row. As mentioned in chapter 2, we need the *PRE* command (t_{RP}) to clear the row buffer by returning the old row and *ACT* command (t_{RCD}) to load the new one. We assume all requests reached Global Queue at t_0 and all *ACT* commands are issued at first before breaching four window active time constraint (t_{FAW}). If one or more *ACT* commands are issued after t_{FAW}, we need to consider the effect of activation commands after $t_{FAW.}$ As per Figure 4.11 and 4.12, in order to calculate Worst Case Closed Request *(WCCR)*, each *read* suffers from (t_{WL} + t_{BUS} + max (t_{WR}, t_{WTR}) and each *write* suffers from t_{RTW}. Finally, the calculated closed request T_{CR} for *read* and *write* requests between t_0 and t` are as shown in eq.3 and eq.4 respectively.

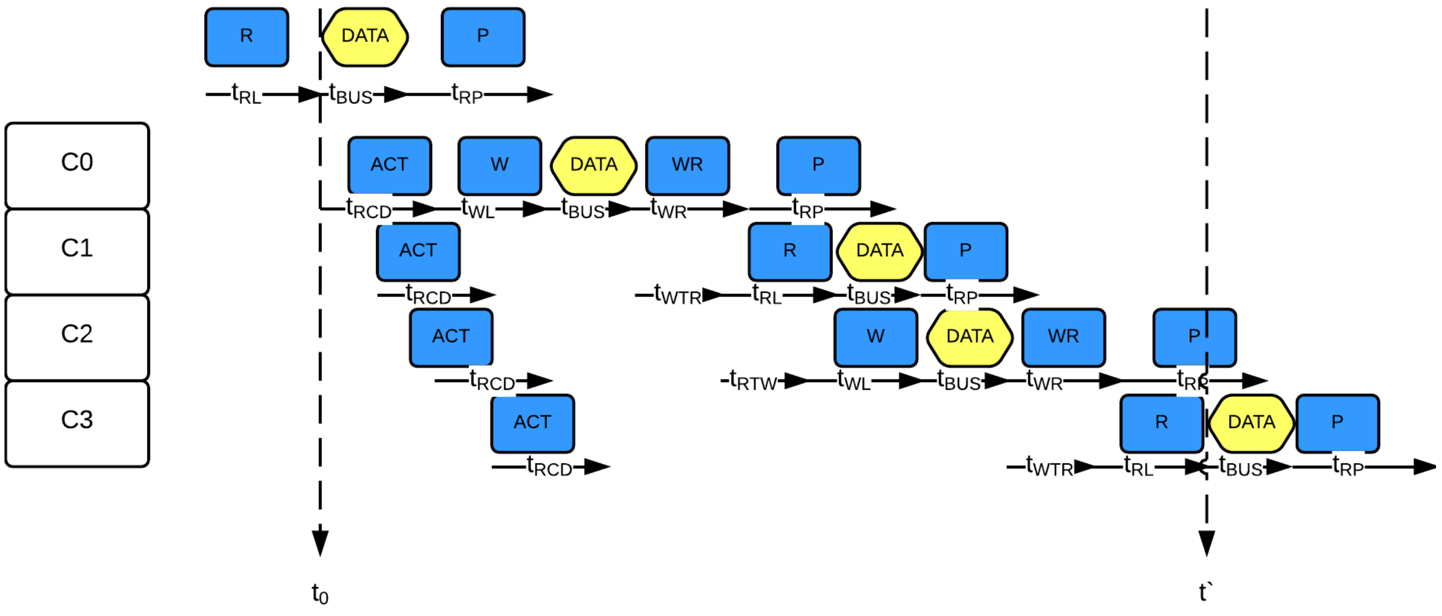


Figure 4.11 *WCCR* Read

$$T_{CR}^{read} = max(C_N t_{RCD}, t_{FAW}) + \frac{C_N}{2}(t_{WL} + \ t_{BUS} + t_{WTR} + t_{RL}) + \left(\frac{C_N}{2} \ - 1\right) t_{RTW} \qquad (3)$$

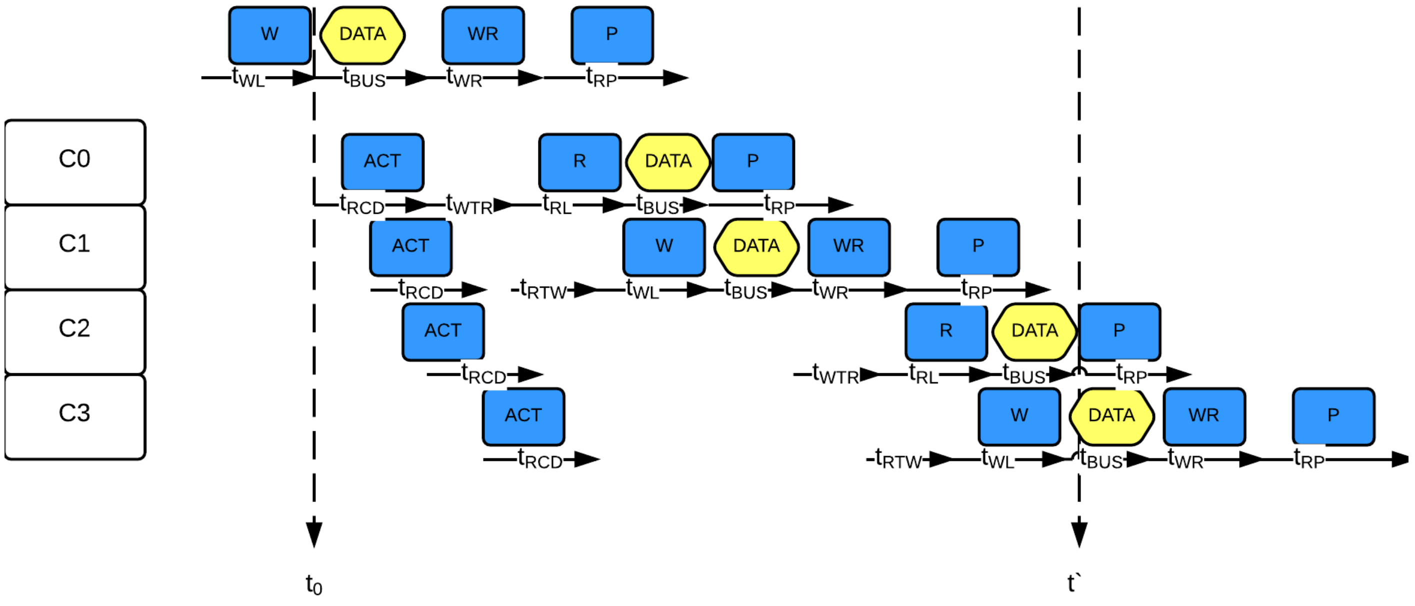


Figure 4.12 *WCCR* Write

$$T_{CR}^{write} = max(C_N t_{RCD}, t_{FAW}) + \ \frac{C_N}{2}(t_{WL} + t_{RTW} + t_{WTR} + t_{RTW}) + (C_N \ - 1)\ t_{WL} \qquad (4)$$

4.3.3 Total *WCET*

In the above analysis, we discussed *WCET* for either all Open Requests or all Closed Requests. Since in real scenario, requests are a mixture of both and no one can guarantee the exact scenario for the actual traffic load. Hence the total WCET time is the accumulation of number of open tasks (N_{OT}) plus number of closed tasks (N_{CT}) plus the estimate time to switch between these tasks $\Delta\tau$ as shown in eq.5. $\Delta\tau$ is divided to $\Delta\tau_{OR}$ for time elapsed to switch from open request to closed request as shown in equation.6 and $\Delta\tau_{CR}$ for time switching closed request then open request as shown in eq.7

$$\begin{aligned}T_{Total} = {} & N_{OT}\,.\,max(T_{OR}^{read}, T_{OR}^{write}) + \\ & N_{CT}\,.\,max(T_{CR}^{read}, T_{CR}^{write}) + 2\Delta\tau_{xx}.\,min(N_{OT}, N_{CT})\end{aligned} \tag{5}$$

$$\Delta\tau_{OR} = \begin{cases} max(t_{RL}, t_{RTW}) + t_{BUS}, & ORD\ then\ CWR \\ t_{WL} + t_{BUS} + t_{WTR}, & OWR\ then\ CRD \end{cases} \tag{6}$$

$$\Delta\tau_{CR} = \begin{cases} max(t_{RC}, t_{RCD} + max(t_{RL}, t_{RTW}) + t_{BUS} + t_{RP}), \\ \qquad CRD\ then\ OWR \\ max(t_{RC}, t_{WL} + t_{BUS} + max(t_{WR}, t_{WTR}) + t_{RP}) \\ \qquad CWR\ then\ ORD \end{cases} \tag{7}$$

Chapter 5. Simulators

In this chapter, we will discuss the tools used in testing the algorithm and achieving results output. To test the algorithm two simulators were used: MARSS core emulator [18] integrated with PTLsim simulator to implement core, cache configuration. The other tool used is DRAMSim2 simulator [19] used for memory controller design. We selected CHstone benchmark to test our design [13] which is a suite that includes various real-time applications.

5.1 MARSSx86

MARSSx86 is a tool for cycle accurate full system simulation of the x86-64 architecture, specifically multicore implementations. MARSS is multicore simulation environment for the x86-64 ISA, with detailed pipeline model, based on Processor Technology Laboratory Simulator (PTLsim). MARSS has extensive enhancements for improved simulation accuracy and it includes detailed models for Coherent Cache and On-Chip Interconnections with implementation of the MESI, MOESI Protocols. MARSS is emulator that run an ISO image contain simulation engine (PTLsim) and the required benchmark needed to handle the tests on ash shown in Figure 5.1.

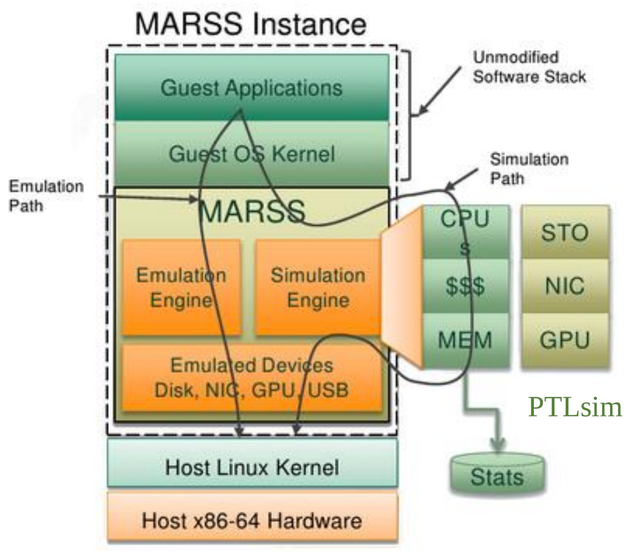


Figure 5.1 MARSSx86 architecture

5.2 DRAMSim2

DRAM simulator is a cycle accurate memory system simulator that includes DDR2/3 memory system model. Both full system and trace-based simulations can be used. DRAMSim2 uses MARSS as front-end to generate traffic to DRAM model. Figure 5.2 describes the DRAM architecture.

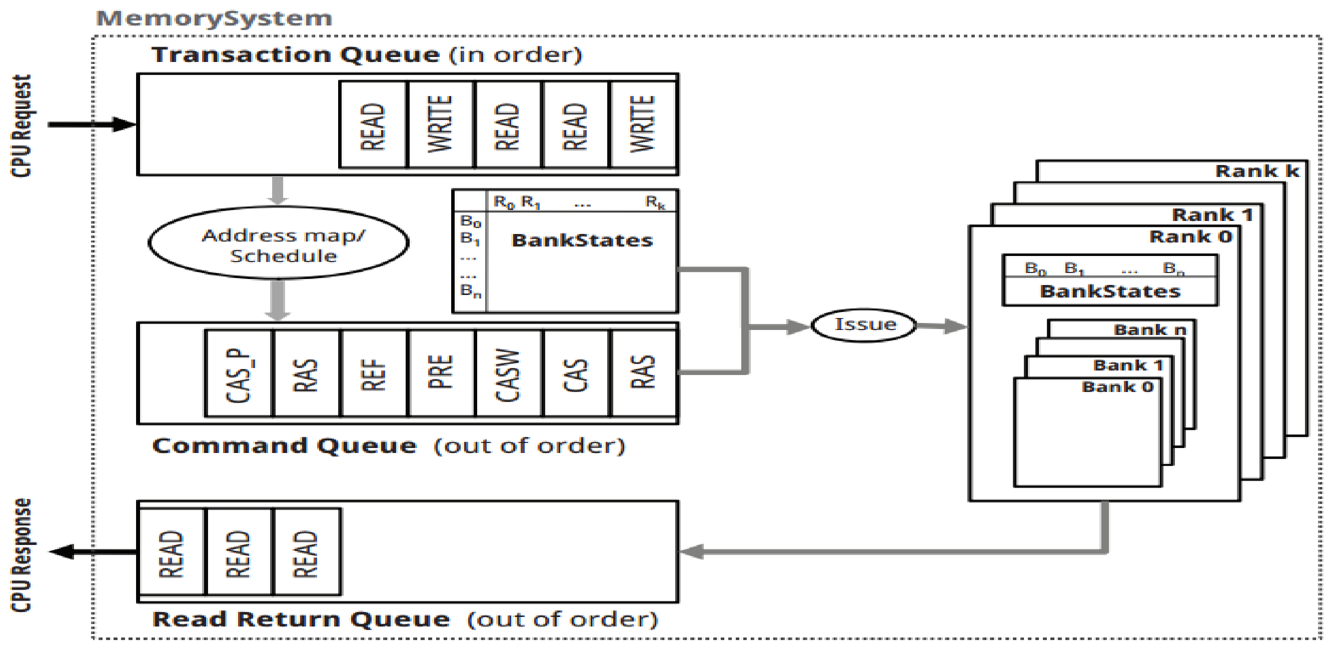


Figure 5.2 DRAMSim2 architecture

5.3 CHstone

The CHStone benchmark suite has been developed for C-based high-level synthesis (*HLS*). CHStone consists of 12 programs which are selected from various application domains such as arithmetic, media processing, security and microprocessor. The CHStone benchmark programs are written in the standard C language. Table 5.1 illustrates the benchmark suite programs.

Table 5.1 Chstone Suite applications

Program	Design Description
DFADD	Double-precision floating-point addition
DFMUL	Double-precision floating-point multiplication
DFDIV	Double-precision floating-point division
DFSIN	Sine function for double-precision floating-point numbers
MIPS	Simplified MIPS processor
ADPCM	Adaptive differential pulse code modulation decoder and encoder
GSM	Linear predictive coding analysis of global system for mobile communications
JPEG	JPEG image decompression
MOTION	Motion vector decoding of the MPEG-2
AES	Advanced encryption standard
BLOWFISH	Data encryption standard
SHA	Secure hash algorithm

Chapter 6. MCES Evaluation

In this section, we compare our approach with *WCAD* [8]. Both *AMC* & *WCAD* use fair round robin arbitration without prioritizing cores and this leads to neglecting memory intensive applications. Also PREDATOR [6] and BWPS in [7] use prioritized cores without request ranking accompanied by *interleaved* banks and *CP*. We believe that putting these approaches together improve the overall performance especially when taking into account different application priorities. It is also worth mentioning that applying *private* bank, eliminates the inter-task interference between cores and applying OP policy utilizes the DRAM performance.

6.1 Experimental Setup

We run our simulations using MARSS core emulator [18] linked with DRAMSim2 simulator [19], for *MC* design, by assigning for each core a different priority and quota. Each core has a private 256KB instruction/data L1 cache and 2MB of unified L2 cache. Hence, all cores run on 2.0 Ghz and all the upcoming experiments run on 4GB DDR3 DRAM memory. Any more details is mentioned in Table 6.1. We assign the priorities to cores in an offline mode based on the number of requests hold by each core or by the memory intensive workload. Hence, benchmarks are investigated first and executed solely to gather profiles to identify their general behavior by applying various simulation runs to achieve the best results by best matching the workloads and cores. We tried different scenarios: normal and reversal scenario. Normal scenario is that each core will be assigned the suitable adequate workload. On the other side, reversal/swapped scenario is switching the benchmark assignments to the cores. Hence, each high priority core will run low benchmark and every low priority core will run the memory intensive application. Highway scenario describes that all cores run the same memory intensive benchmark application. Also during simulation run, priorities and quotas can be changed based on the cores utilization however we didn't do experiments for this. For now, we assume that the intensive workload is assigned to cores in an ascending order, i.e. the 1st core has the highest priority and quota and the last core, 4th core in our experiments, has the lowest priority and consequently quota for 4 cores system and 8th core will be the lowest priority for 8 core systems. We are using in our simulations wide range DDR3 DRAM (different models and sizes as in table 2.6) devices, as it is the most recent technology used by modern CMPs, with 64-bit data BUS width, one data channel and one command channel. We run multiple simulations on benchmark CHstone [20] to obtain memory traces for performance measurement and analysis. All benchmark suites are assigned to cores based on their size and how intensive they are to utilize the memory. The intensity of benchmarks was calculated by 2 ways: 1- calculate the time used to run the executable benchmark file. 2- Run each benchmark solely on a core and check the number of requests generated.

6.2 Evaluation metrics

Results regarding performance measurements are calculated using weighted speedup however fairness measurements are obtained using harmonic speedup which is used as a balance between performance and fairness where N is number of cores used [21]

$$Weighted\ Speedup = \sum \frac{Perf._{MCES}}{Perf._{WCAD}} \tag{8}$$

$$Harmonic\ Speedup = \frac{N}{\sum \frac{Perf._{MCES}}{Perf._{WCAD}}} \tag{9}$$

In order to determine if our solution design (MCES) can be determined as performance oriented or energy oriented or comprehensive performance and energy design, we used another combined metric called EDP [22] which calculate inverse perf. /energy

$$EDP = \frac{1}{\sum \frac{perf.}{Energy}} \tag{10}$$

Table 6.1 4-CORE/8-Core system simulation parameters

Processor	2.0 Ghz, x86 processor, 128 entry re-order queue size, 36 entry issue queue size,48 entry load queue size and 32 entry store queue size
L1 cache	write through ,256 KB, 8-way set associative,2 cycles latency, 64 bytes line size, 2 read ports and 1 write port
L2 cache	Write through, 2MB, 8-way set associative,5 cycles latency, 64 bytes line size, 2 read ports and 2 write ports
MCES	128 entry re-order private buffer, 2 private buffers per core
DRAM components	512MB , 1 GB ,2 GB , 4GB , 8 banks, 64 B row buffer per bank

6.3 Results

6.3.1 Sensitivity analysis

In this section, we will discuss some experiments to show our algorithm behavior response with changing some parameters. First option we change is the core frequencies we use. Figure 6.1 describes the average throughput for MCES data bus for different running core frequencies for 4-core system. As the core frequency increases, the final throughput for our memory controller increases. Hence, MCES can handle high core frequencies. In Figure 6.2, we checked variable cache sizes. Results achieved that as increasing the size by doubling, the average execution time enhanced by 1% , 17% , 4% for 64K , 128K, 256K. As the cache size increases, the percentage of hit ratio increases which lead to a decrease in the number of requests sent to the memory to retrieve the data. These results show the scalability of our new proposed MCES against frequency scaling and cache sizes of the future multicore systems.

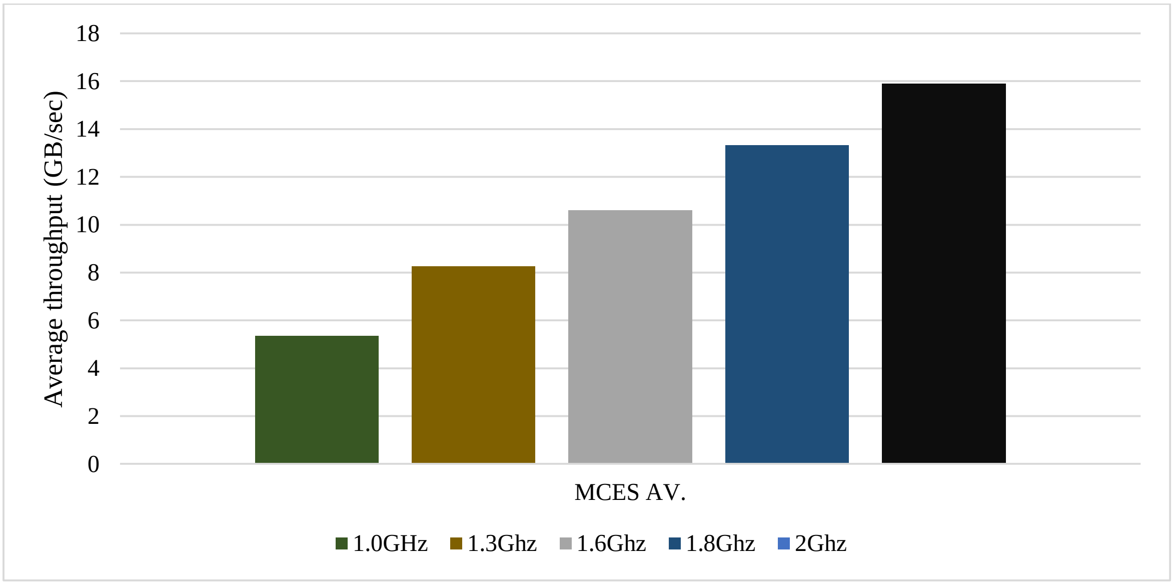


Figure 6.1 MCES average throughput

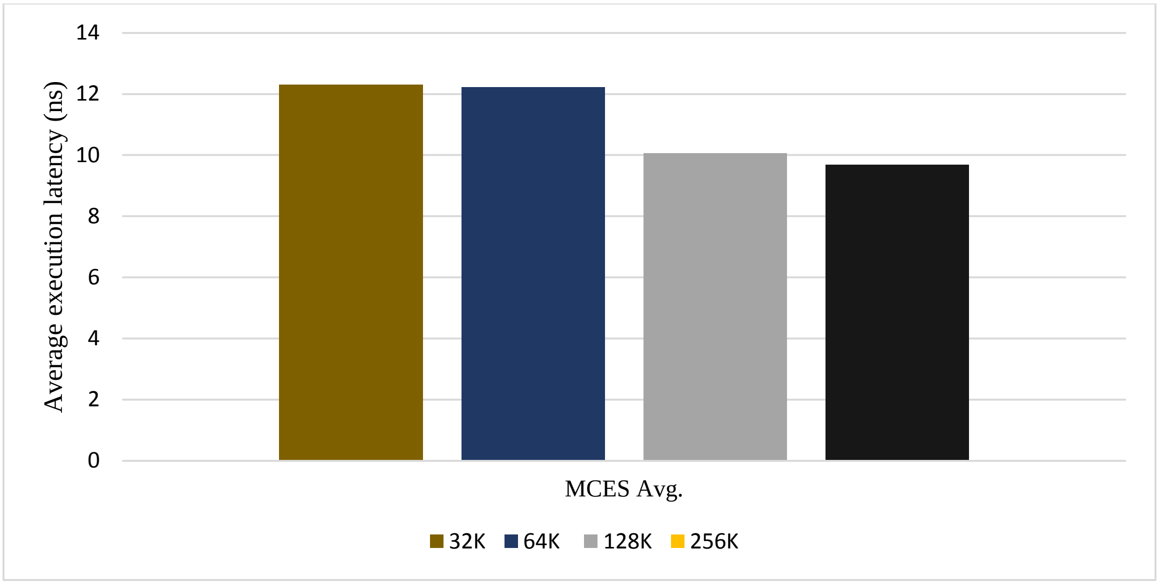


Figure 6.2 MCES average execution latency versus Cache size

Figure 6.3 represents a comparison between global queue latency and total latency measured in memory cycles (y-axis) compared to global queue size in number of requests enqueued which represents the summation of all cores' requests (x-axis). Total latency represents end-to-end latency or in other words the latency for a request from the time it reaches the core's private buffer till it is returned back to the core. Global Queue (*GQ)* latency describes the latency of request since it reached the *GQ* after dispatching from the private buffer till it is issued by the *MC.* We tried different queue sizes to study resulted latency. For *GQ* size = 4 requests, which is the lowest size as each core will have one request quota, the latencies reach the maximum value because the amount of requests arrived in the private buffer are much more than that issued and this proves that running memory intensive applications using equal shares among the cores, degrades the performance. Both latencies decrease as size increase till we reach the minimal optimum point at *GQ* size = 6 requests. This recommends the uneven distribution of the quota per core which is the main proposal of our research in this thesis. Moving beyond this point, and increasing the size of the queue more than 6 requests, the latencies start to slightly increase again but it didn't reach the first point because as the core share increases, the upcoming requests after the first request wait for some memory cycles before issuing.

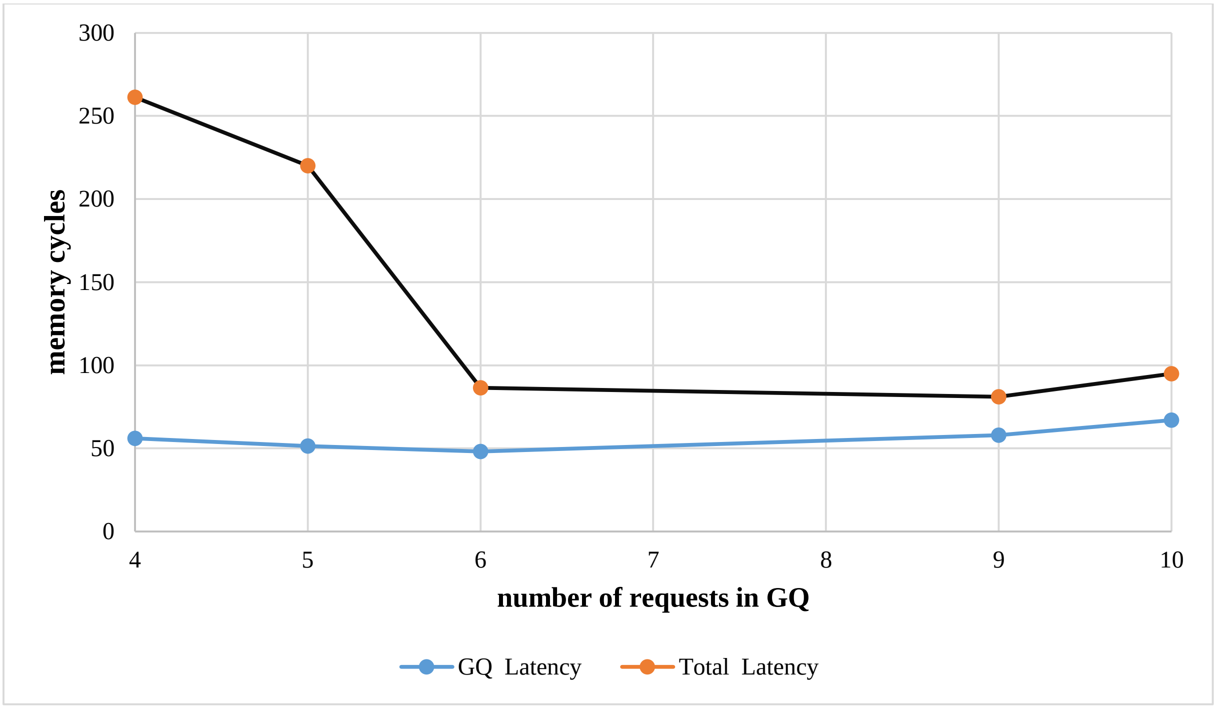


Figure 6.3 Request total latency versus GQ latency

Figure 6.4 represents a comparison between our two evaluation metrics, performance measured by arithmetic mean and fairness measured by harmonic mean (y-axis) versus multiple values for *MOT* ,from zero till 200 memory cycles, (x-axis). We can infer from the figure that at *MOT* from 40 to 50 memory cycles, we have the best tradeoff between performance and fairness. Hence, we choose this value for our figures and results. For *MOT* between 0 and 40, no significant variation is noticed for the performance level, however the higher *MOT* in that range is the better the fairness. This is due to short period of time given to the cores to issue their requests specially the lower priority ones. After this point, performance deteriorate and fairness increases slightly because lower priority cores will have time slots to issue their requests from global queue and more chance to achieve more progress. Hence, at MOT from 40 to 50 memory cycles, we get the benefit of the both metrics to achieve maximum performance without starving the lower priority cores.

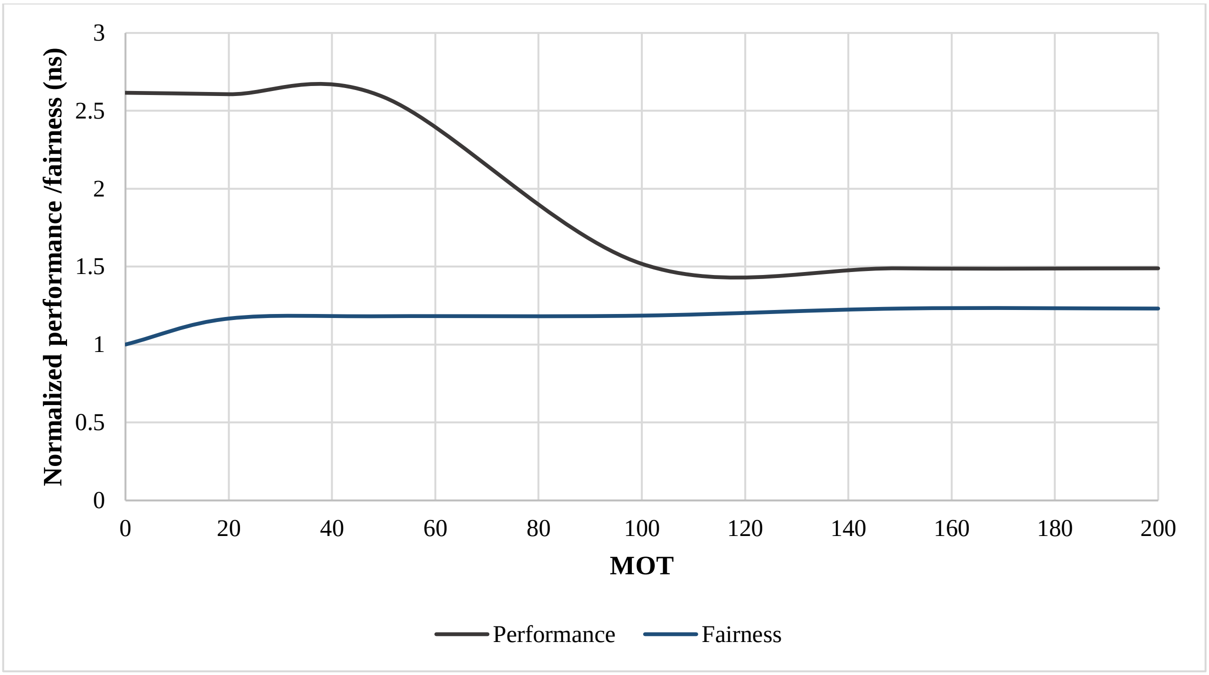


Figure 6.4 MCES Performance versus fairness

Figure 6.5 and 6.6 represent percentage of performance (y-axis) versus different versions of DDR3 DRAMs (x-axis). Both figures capture the effectiveness of *MCES* and *WCAD* controllers regarding *HRT* and *NHRT* requests. The increased performance for *HRT* tasks ranges from 19% for DRAM 512 MB to 23% for DRAM 4GB because as the DRAM size increases, the *row hit* ratio increases. Hence, performance increases. Also *WCAD* can't distinguish between these types of tasks and treats them equally. Regarding NHRT results, *MCES* performance increased by 14% for 512 MB to 3% for 4GB over *WCAD*. This reduction in the performance enhancement for NHRT requests is caused by re-ordering policy between HRT and NHRT in the private buffers as per the re-ordering request ranking rule mentioned in the arbitration rules.

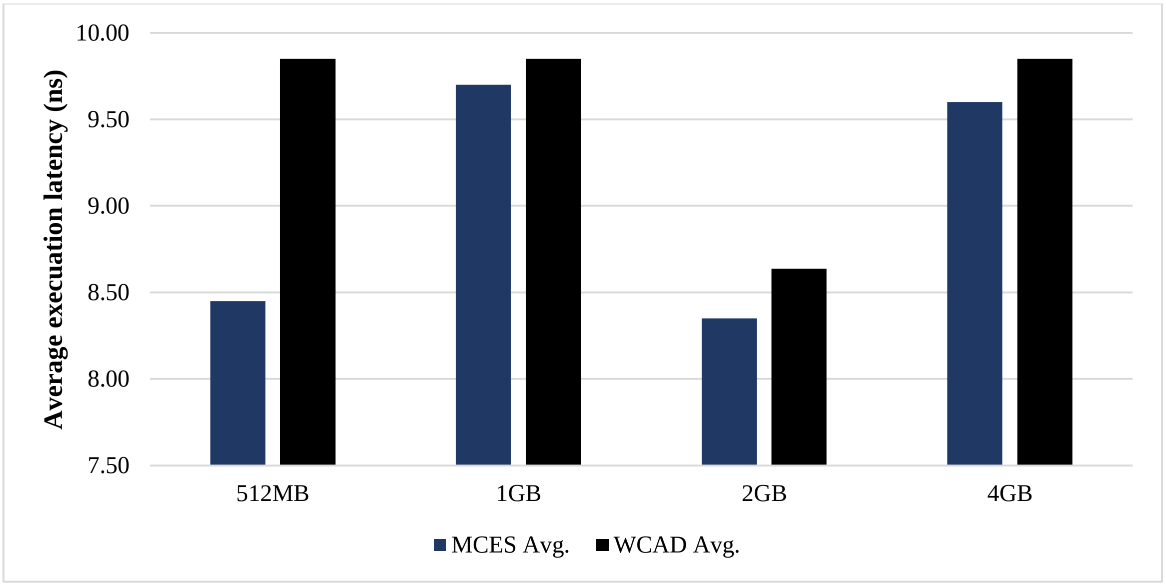


Figure 6.5 DDR3 MCES vs WCAD NHRT execution latency

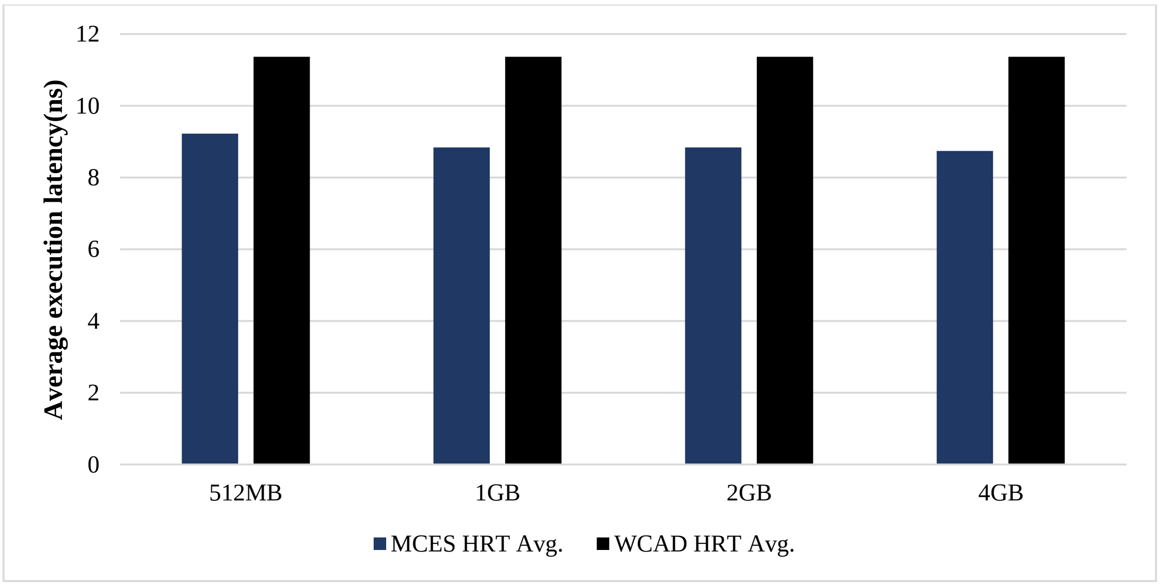


Figure 6.6 DDR3 MCES vs WCAD HRT execution latency

Figure 6.7 discusses the trend of request processing for the both MCES and WCAD algorithms versus time by calculating the number of requests served (y-axis) versus cycles elapsed (x-axis). First the both algorithms serve huge number of requests but with improvement compared to MCES by 70%. After that, the number of served requests decreases till it reach the minimal point at 1500 cycles. The number of requests continues to increase again and settles at 2500 cycles but with more requests served by MCES by 40 %.

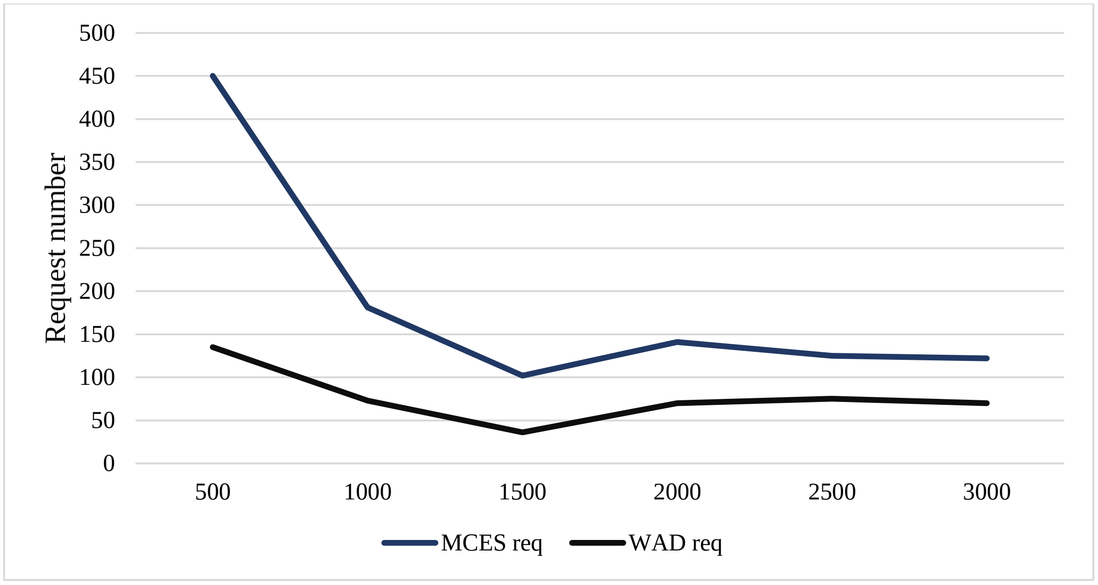


Figure 6.7 MCES versus WCAD request processing trend

6. 3.2 Algorithm Comparative Analysis

6. 3.2.1 4 Core System

In Figure 6.8 and 6.9, we compare our performance and energy consumption results to simulation results obtained by *WCAD* algorithm [4] for 4 core system. The y-axis represents the percentage of average execution latency in nano seconds and energy consumed in joules on a per core basis and x-axis represents simulated cores with assigned suites. Multiple suites are tested based on their size and ability to memory access. In Figure 6.8, the results obtained are 33% and -23% for low priority cores (C_2 and C_3 respectively) because they have lower quotas nearly equal to *WCAD* cores. The application running on C_3 can issue its requests in a timely manner although it suffers from being the last core its performance (due its low priority order). On the other side, high priority cores (C_0 and C_1) achieve marginal performance improvements (68% and 23% respectively) using the new scheduling of MCES, so cores dispatch more requests, and hence, performance increases. Figure 6.9 compares the average energy consumption in joule per core performed by *MCES* and *WCAD*. MCES saves energy consumption by 18% and 23% for high priority cores (C_0 and C_1) and 10% and -17% for low cores (C_2 and C_3). C_3 consumes the most energy because it remains idle most of the time waiting for its slot to issue its requests. On average across all the four cores, *MCES* achieves overall high throughput using weighted speedup reached 31% and mostly equal energy consumption to WCAD for 4-core system with overall EDP 31%. On the other side, WCAD scored harmonic speedup by 30%

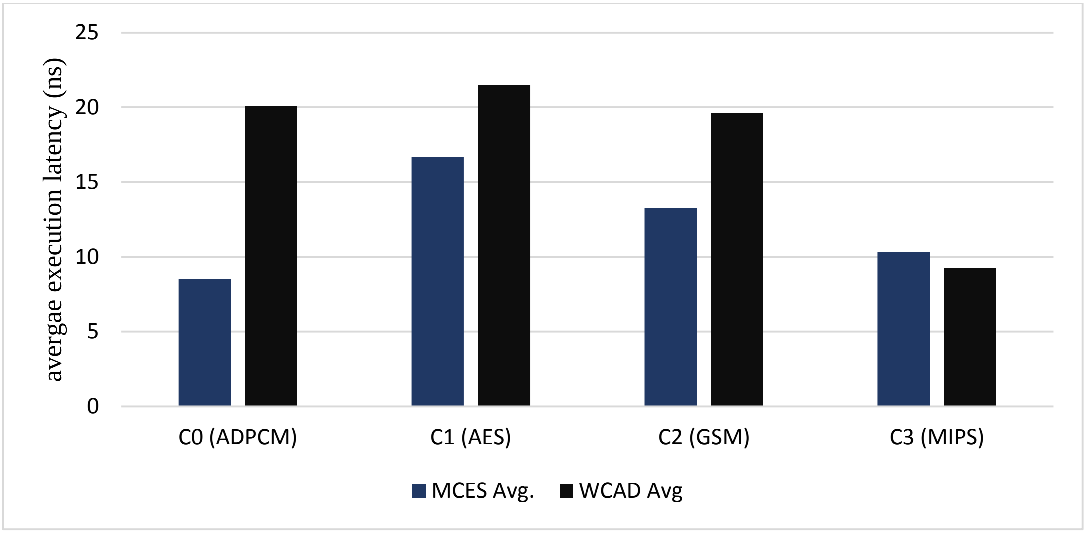


Figure 6.8 MCES versus WCAD Normal run performance

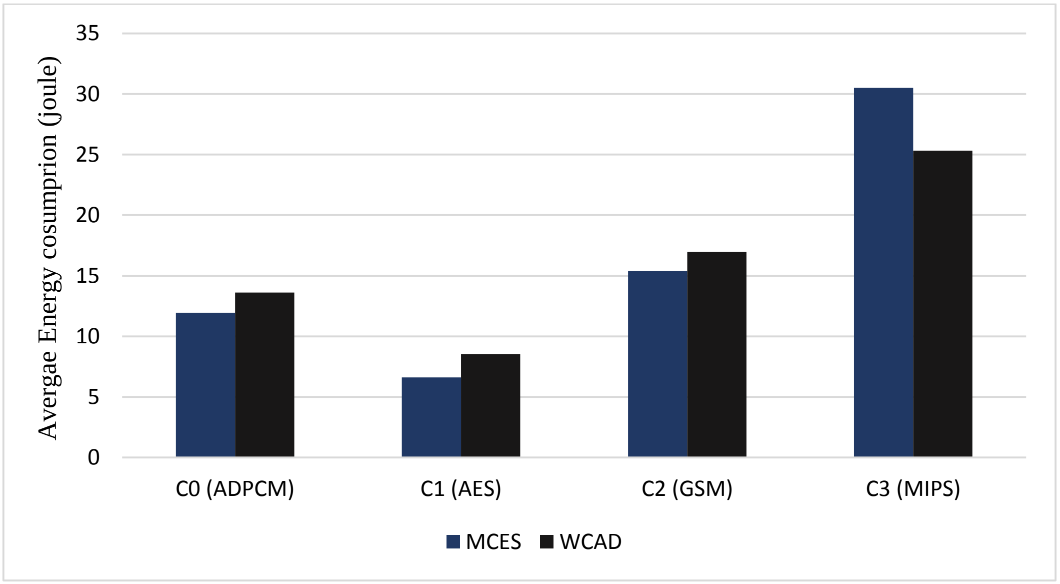


Figure 6.9 MCES versus WCAD Normal run energy consumption

Figures 6.10 and 6.11 discuss the same parameters (average latencies and energy consumptions) for 4-core system. If we swapped the workloads i.e. the low priority cores run the high memory intensive applications and the high priority cores run the low memory applications. We find that the high priority cores (C_0 and C_1) achieve almost 0% in performance improvement compared to WCAD results. This is due to that they don’t exploit the advantage of the priority and quota. We still find the lowest priority core (C_3) suffocates due to high number of requests waiting for issuing with the lowest priority and quota hence maximizing the latency. Overall performance across all cores MCES shows performance degradation of about 10%. On the other hand in Figure 6.11, the power savings for high priority cores (C_0 & C_1) are almost 45% because they finish their requests early and these results increased for C_2 to reach 55% and finally the last core C_3 by 88 % saving of energy consumption. Overall energy saving reaches 46%. For the reverse execution scenario of running the benchmarks, although MCES is not achieving performance enhancement (-10%), it can achieve very high overall energy saving of 46% and 10% harmonic speedup with EDP equal to 40%

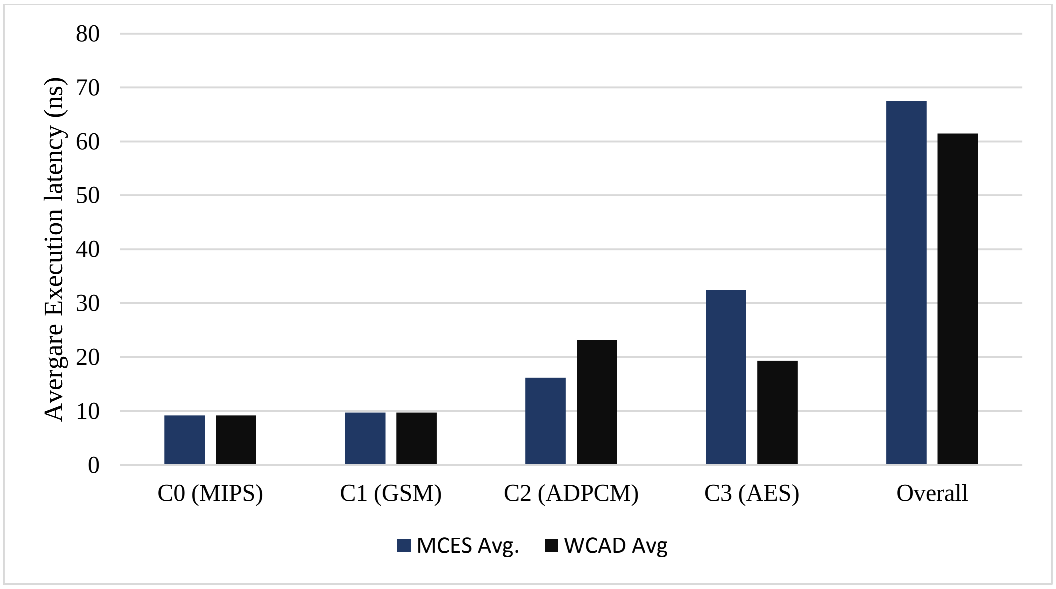


Figure 6.10 MCES versus WCAD Reverse run performance

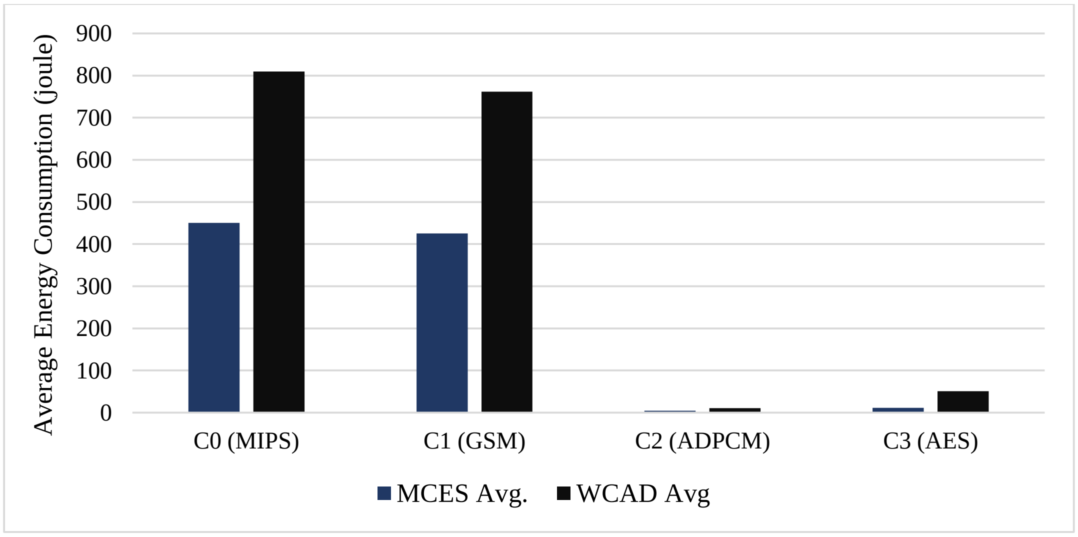


Figure 6.11 MCES versus WCAD Reverse run energy consumption

Figure 6.12 and 6.13, discuss the high-way run results when all cores run the highest memory intensive application ADPCM). In figure 6.12, results range from 68% for C_0, 65 % for C_1, 22% for C_2 and degraded by 70 % for C_3 resulting in overall weighted speedup improvement of 13 % comparing MCES to WCAD. While in Figure 6.13, energy saving ranges from 8% for $C_{0,}$ 12 % for $C_{1,}$ to 42% for C_2 and C_3 comparing MCES to WCAD resulting in overall energy saving 18% and overall EDP 28%. On the other side, WCAD scores harmonic speedup 13%

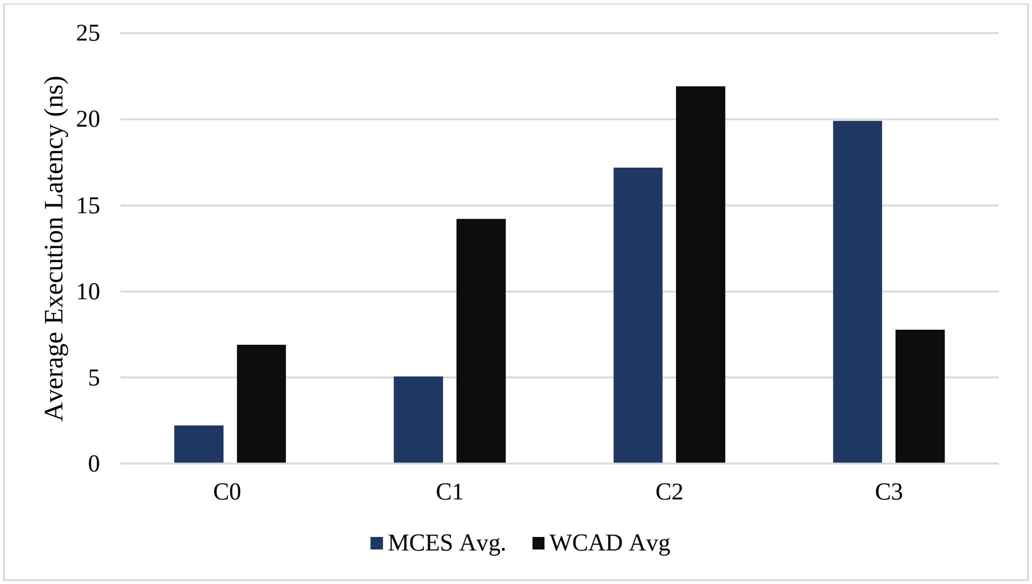


Figure 6.12 MCES versus WCAD High-way run performance

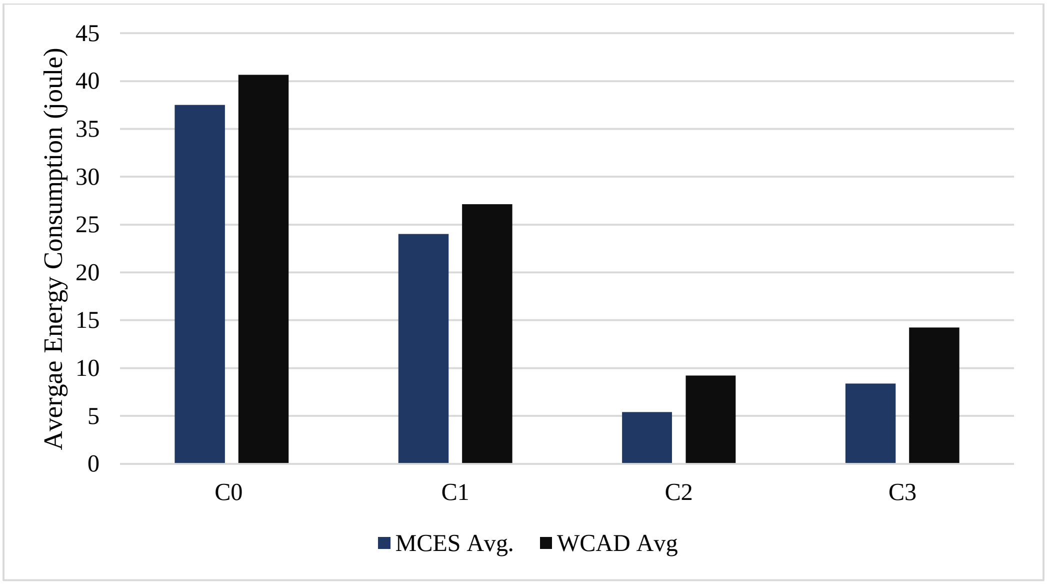


Figure 6.13 MCES versus WCAD High-way run energy consumption

6. 3.2.2 8-Core System

In this section, we experiment our algorithm versus WCAD for 8-core system. We assign the priorities and quotas in an ascending order (Normal Execution Scenario). Hence, C_0 is the highest priority and quota while C_7 is the lowest priority and quota. Figures 6.14 and 6.15 discuss the run under 8-core system with the *normal*[1] execution scenario (the workloads are assigned as per their memory intensity). Hence, the highest memory intensive benchmark (ADPCM) is assigned to the most powerful core C_0 with assigned quota 9 requests per MOT and least workload (MPEG2) on C_7 with quota 2 requests. In Figure 6.14, results achieved are 43% enhancement for the high priority cores and 21% degradation for the lower priority cores with overall weighted speedup enhancement of 16%. Figure 6.15 describes the energy consumption. Results ranged from improvement of 20% for high priority core (C_0) to 19% degradation for low priority core (C_3). This is because the more time the bank opens waiting for issuing, the background energy consumption increases. Hence, the high priority cores usually use lower energy consumptions as their powerful abilities to issue their requests and finish reading/writing in memory while, the lower priority cores wait till the high priority cores to finish, which forces the bank to wait for the memory controller to issue the required request. MCES scored overall EDP 13% while WCAD achieves 14% harmonic speedup

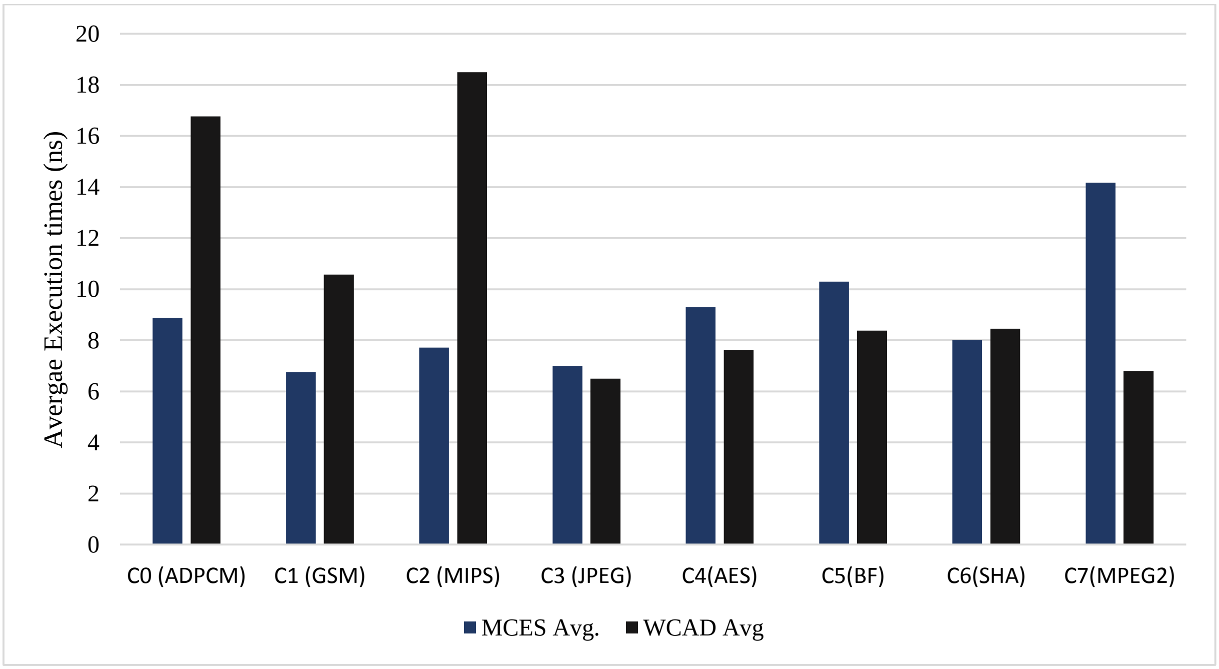


Figure 6.14 MCES versus WCAD Normal run 8-core performance

[1] For more information, check experimental setup section 6.1

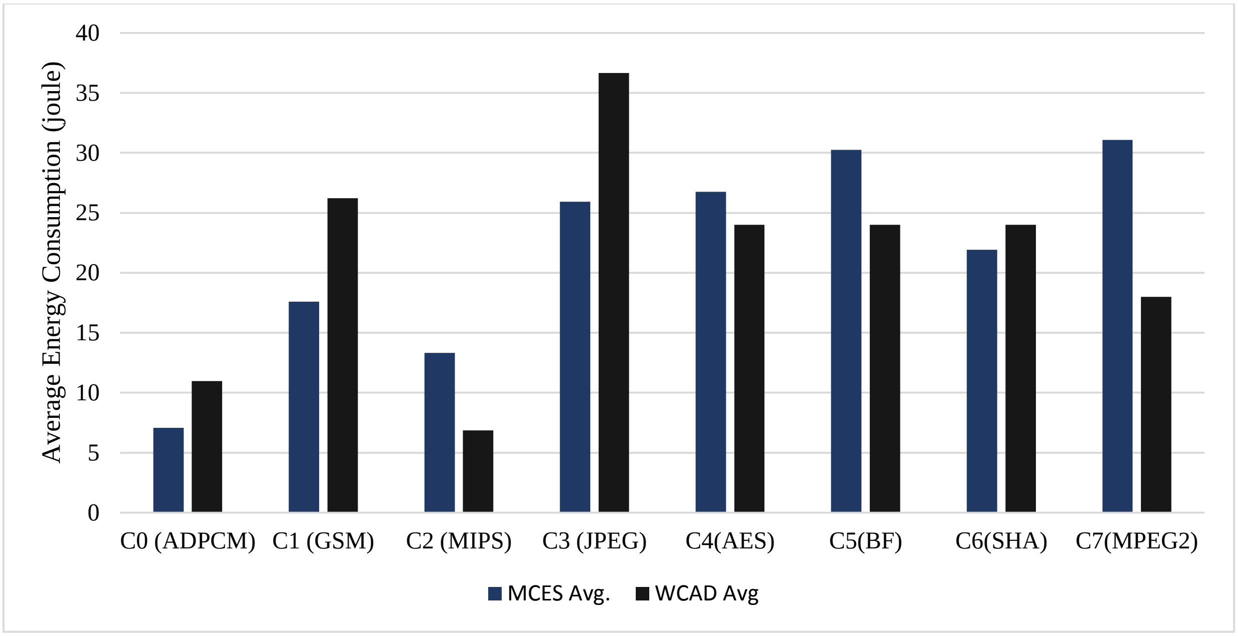


Figure 6.15 MCES versus WCAD Normal run 8-Core energy consumption

Figures 6.16 and 6.17 discuss the reverse order benchmark to core assignment run (reversal scenario[2]). Hence, all the memory intensive applications will run on the low priority cores and vice versa. In Figure 6.16, performance enhancements range from 49% for high cores since they are able to finish their assigned application in time while low priority cores have53% performance degradation causing 48% overall degradation. In Figure 6.17, results ranges from 72% saving in energy for high cores and 15% for lower cores because after high cores serve their applications, they let the lower cores able to serve their part which lead to lower memory opening hence, lower energy consumption Hence, the overall energy enhancement reached 54% , 48% harmonic speedup and EDP reaches 13%

[2] For more information, check experimental setup section 6.1

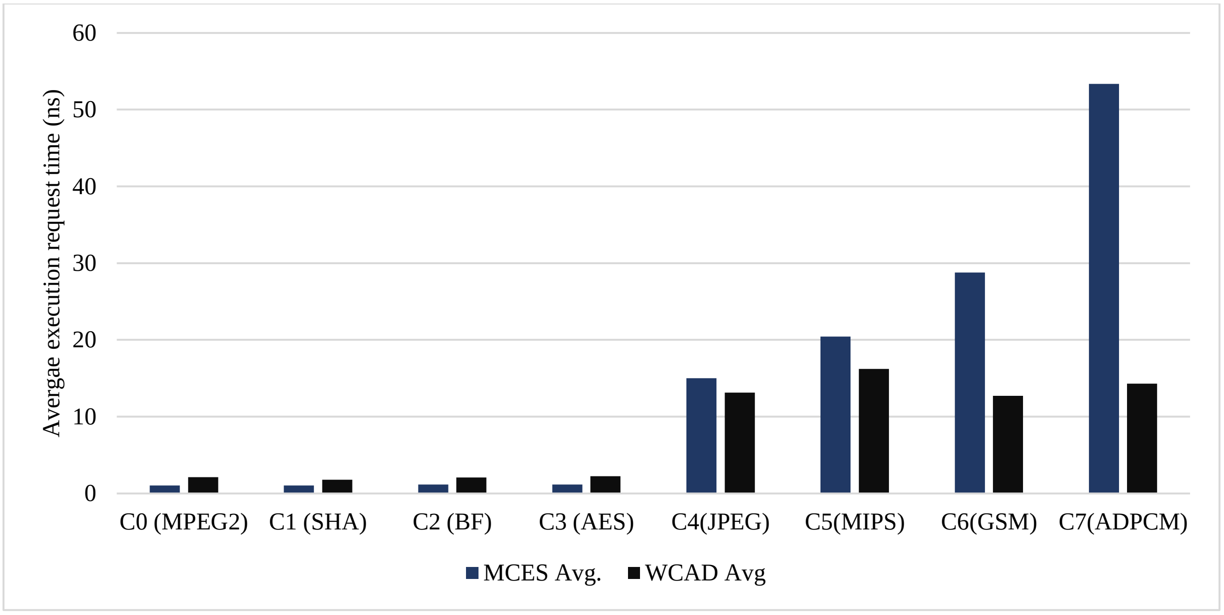


Figure 6.16 MCES versus WCAD Reverse run 8-Core performance

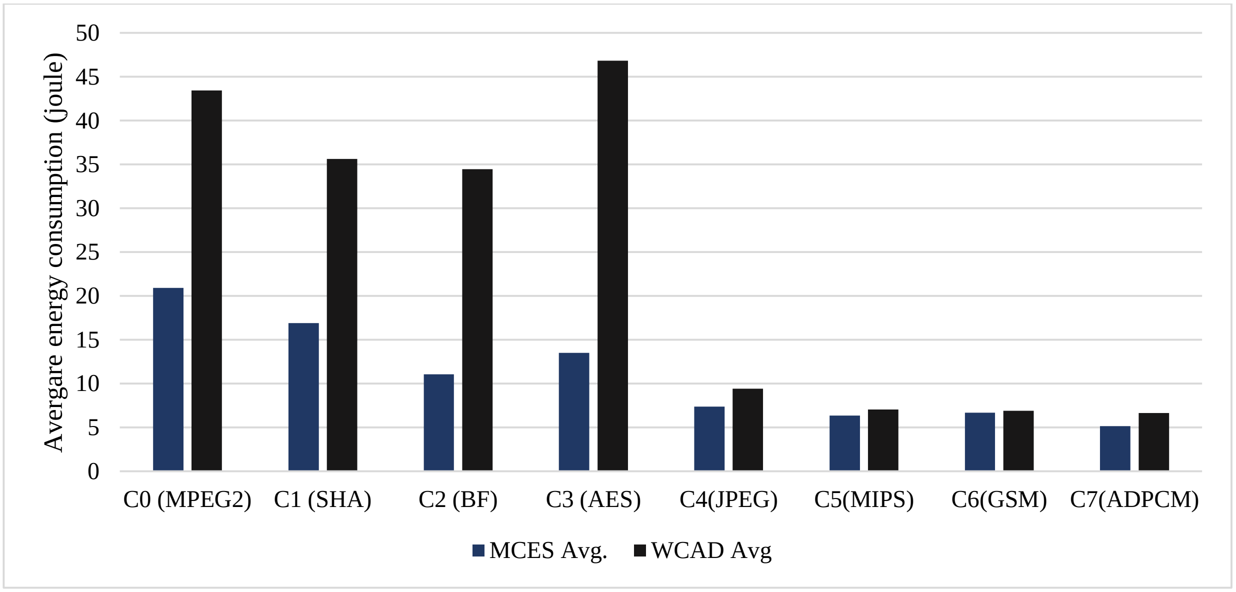


Figure 6.17 MCES versus WCAD Reverse run 8-Core energy consumption

Figure 6.18 and 6.19 describe the highway run for all cores. Hence, the memory intensive applications (ADPCM, GSM, MIPS, AES, and JPEG) run on all cores in parallel. In Figure 6.18, results ranges from 41% improvement for high cores and 69% degradation for low priority cores resulting in 44% for overall degradation. In Figure 6.19, results range from 5% degradation for high cores and 14 % improvement for low cores. Although the high performance degradation for some benchmarks, the energy-delay product results show a completely different dimension of the MCES.

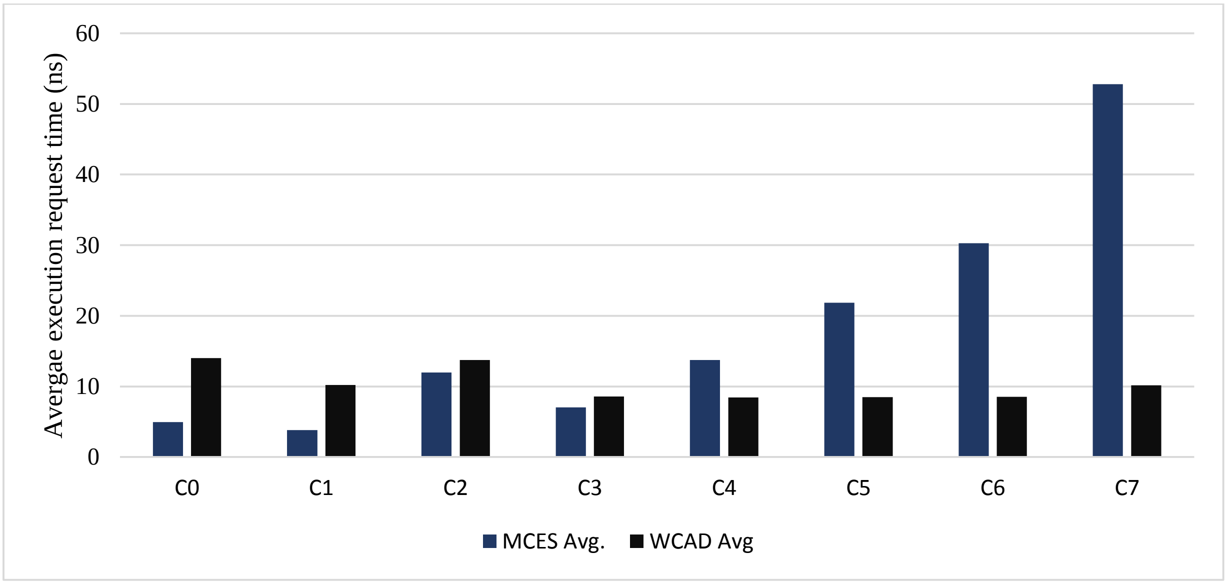


Figure 6.18 MCES versus WCAD High-way 8-Core performance

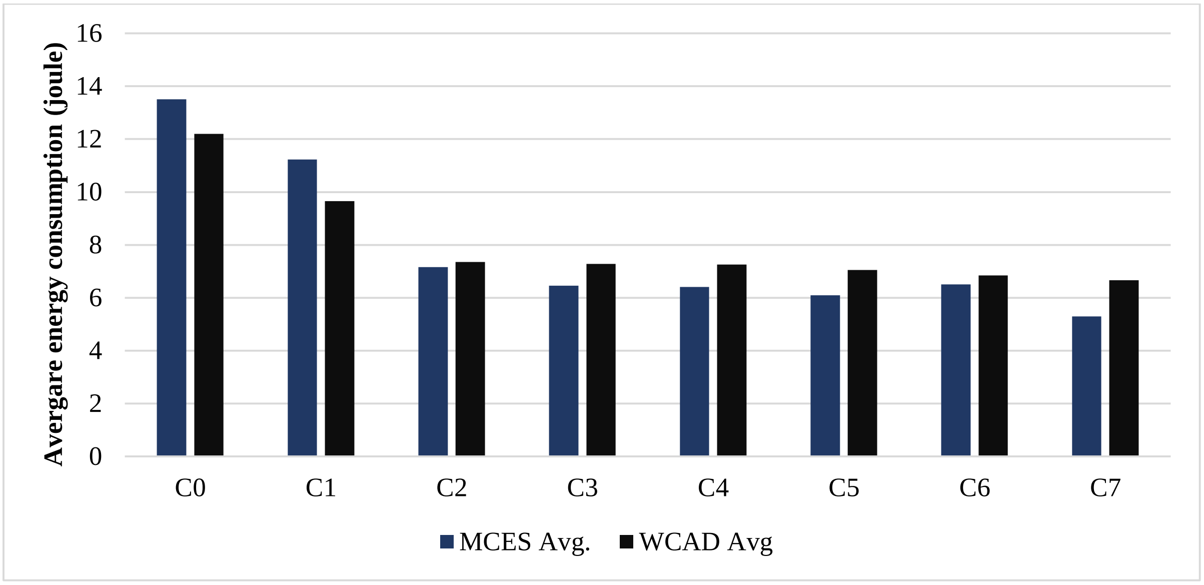


Figure 6.19 MCES versus WCAD High-way run 8-Core energy consumption

To summarize all the figures in one chart, we add all the metrics and compare their percentage. WS4N stands for weighted speedup for 4 cores Normal run, HS4N is Harmonic Speedup for 4 Cores Normal run and EDP4N is Energy to Delay product combination for 4 cores Normal run as shown in Table 6.2. From the Figure 6.20, we can get that MCES scores better WS and EDP results for normal run in four and eight cores, while degraded HS because MCES concerns about the performance of highly memory intensive applications which impacted the fairness metric of lower cores. On the other side, MCES scores better HS and EDP for reverse runs and degraded WS because MCES switched the interest to serve the lower intensive application which reflected in higher fairness and energy/performance combination with degraded overall performance. This Figure describes the strengths and weakness of MCES algorithm.

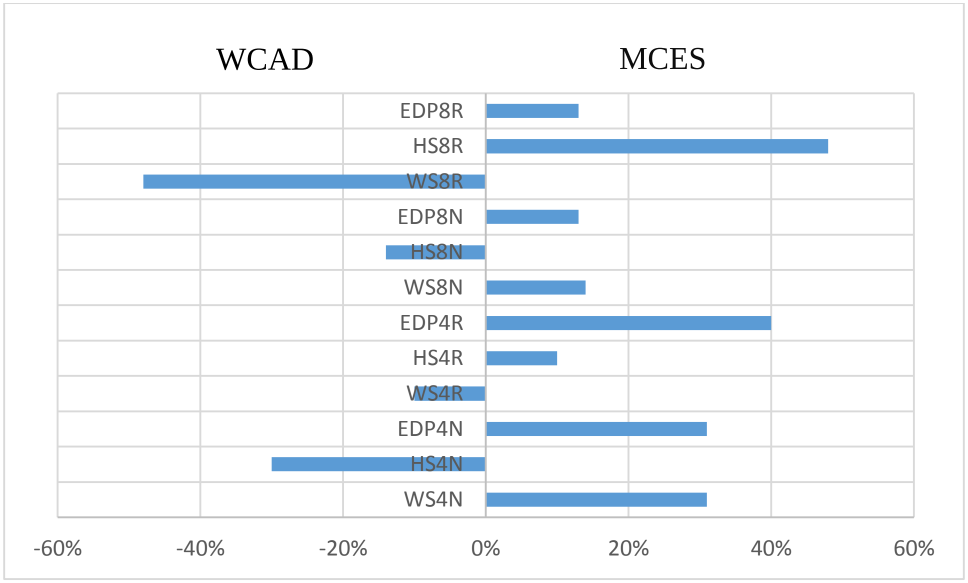


Figure 6.20 MCES versus WCAD comparison metrics WS, HS, and EDP for all workload scenarios

Table 6.2 Summary metrics abbreviations

Parameter	Description
WS4N	Weighted Speedup for 4 core Normal run
HS4N	Harmonic Speedup for 4 core Normal run
EDP4N	Energy Delay Product for 4 core Normal run
WS4R	Weighted Speedup for 4 core Reverse run
HS4R	Harmonic Speedup for 4 core Reverse run
EDP4R	Energy Delay Product for 4 core Reverse run
WS8N	Weighted Speedup for 8 core Normal run
HS8N	Harmonic Sppedup for 8 core Normal run
EDP8N	Energy Delay Product for 8 core Normal run

Chapter 7. Conclusion & possible future work

7.1 Conclusion

This thesis proposed *MCES*, a new design for the memory controller that considers core priority in request scheduling to fit real-multimedia applications alongside with request ranking within cores to suit hard real-time applications. Variable core quota approach introduced in *MCES* allows adaptation of the core needs of request service. Such scheduling leads to improved timing and performance to the high priority cores while providing fairness among different requests for lower priority cores. *MCES* uses the state information of DRAM as shared memory resource in multi-core system to provide an estimated latency for *HRT* and *NHRT*

MCES utilizes *Private* Banks and *Open Page* policy to eliminate the inter-task interference between cores and improves the DRAM performance by exploiting the row-hit cases. We calculate the timing latencies for MCES requests under analysis including open requests only, closed requests only and mixture of both to be able to estimate the WCET for these tasks.

In order to test the performance of our design, we used two simulators. MARSS for core and cache design. MARSS was able to run the workloads and transfer the core requests to our memory controller algorithm implemented on DRAMSim2 simulator.

We compared our design, MCES, with the last released design in the literature. We applied different workload scenarios. We defined normal, reverse, high-way scenarios. We used metrics to check the ability of our algorithm to compete and we used weighted speedup to measure performance, harmonic speedup to measure performance alongside with fairness. Last metric used was EDP to test if MCES is energy oriented or performance oriented or can work as tradeoff between energy and performance. MCES achieved better performance results in normal scenario using weighted speedup reached 31%, 14% for 4-core and 8-core systems while degraded fairness and power-performance results using harmonic speedup and EDP. On the other side, MCES scored better fairness and power-performance in reverse and highway scenarios reached 10%, 48% for harmonic speedup and 51%, 76% for EDP.

Hence, MCES can be used as performance oriented memory controller algorithm if used in normal scenario, while fairness and power-performance oriented if used in reverse or high-way scenarios.

7.2 Possible future work

Our future work will focus on

1- Studying the effect of cache interference on WCET analysis beside the effect of using heterogeneous core model and what is the impact row hit ration overall threads latency
2- Trying dynamic priority scheduling. Although applying this technique may lead theoretically to timing anomalies for *HRT* requests and consequently overall performance degradation, further study and investigation is needed.
3- Applying the same algorithm on higher level aspects as big data, data cloud applications. This upgrade will need to determine if we can use the same design or we can more than one memory controller to handle the tremendous amount of data generated in such applications
4- Designing map algorithm to expect all the WCET scenarios before execution. This action will undertake huge pre-computation but will deliver high results output
5- As per the summary Figure 6.20, the EDP scores better results in all scenarios. Hence, focusing on energy enhancement which is vital requirement for various modern embedded systems
6- Finally, implementing the solution on circuit design to precisely measure the performance, power consumption and area overhead of our design.

Chapter 8. Publications

· Ahmed S. S. Mohamed, Ali A. El-Moursy, "Real-Time Memory Controller for Embedded Multi-core System", In the Proceeding of the 17th International Conferences on High Performance Computing and Communications (HPCC-2015), New York, USA August 24 - 26, 2015.

· Ali A. El-Moursy, Ahmed S. S. Mohamed, Hossam A. H. Fahmy, "Quantum-Based Memory Controller Design of Enhanced Scheduling for Embedded Multi-core Processors", submitted, Journal of Parallel and Distributed Computing.

References

[1] Kumar, Rakesh, et al. "Heterogeneous chip multiprocessors." *Computer* 11 (2005): 32-38.

[2] Wilhelm, Reinhard, et al. "The worst-case execution-time problem—overview methods and survey of tools." *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008): 36.

[3] Cuppu, V., Jacob, B., Davis, B. and Mudge, T., 2001. High-performance DRAMs in workstation environments. *Computers, IEEE Transactions on, 50*(11), pp.1133-1153.

[4] JEDEC, "DDR3 SDRAM Standard JESD79-3F," July 2012.

[5] Paolieri, Marco, et al. "An analyzable memory controller for hard real-time CMPs." *Embedded Systems Letters, IEEE* 1.4 (2009): 86-90.

[6] Akesson, Benny, Kees Goossens, and Markus Ringhofer. "Predator: a predictable SDRAM memory controller." *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2007.

[7] Shah, Hardik, Andreas Raabe, and Alois Knoll. "Bounding WCET of applications using SDRAM with priority based budget scheduling in MPSoCs." *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*. IEEE, 2012.

[8] Wu, Zheng Pei, Yogen Krish, and Rodolfo Pellizzoni. "Worst case analysis of DRAM latency in multi-requestor systems." *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*. IEEE, 2013.

[9] Shah, Hardik, Andreas Raabe, and Alois Knoll. "Priority division: A high-speed shared-memory bus arbitration with bounded latency." *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011.

[10] JEDEC. "DDR3 SDRAM standard (revision F)". July 2012. Retrieved 2015-07-05.

[11] JESD79-2E. JEDEC. April 2008. p. 78. Retrieved 2009-03-14.

[12] Shevgoor, Manjunath, et al. "Understanding the Role of the Power Delivery Network in 3D-Stacked Memory Devices."

[13] Esmaeilzadeh, Hadi, et al. "Architecture support for disciplined approximate programming." *ACM SIGPLAN Notices*. Vol. 47. No. 4. ACM, 2012.

[14] Heithecker, Sven, and Rolf Ernst. "Traffic shaping for an FPGA based SDRAM controller with complex QoS requirements." *Proceedings of the 42nd annual Design Automation Conference*. ACM, 2005.

[15] Abts, Dennis, et al. "Achieving predictable performance through better memory controller placement in many-core CMPs." *ACM SIGARCH Computer Architecture News*. Vol. 37. No. 3. ACM, 2009.

[16] Paolieri, Marco, et al. "Hardware support for WCET analysis of hard real-time multicore systems." *ACM SIGARCH Computer Architecture News*. Vol. 37. No. 3. ACM, 2009.

[17] Paolieri, Marco, et al. "IA^ 3: An Interference Aware Allocation Algorithm for Multicore Hard Real-Time Systems." *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2011 17th IEEE*. IEEE, 2011.

[18] Patel, Avadh, Furat Afram, and Kanad Ghose. "Marss-x86: A qemu-based micro-architectural and systems simulator for x86 multicore processors." *1st International Qemu Users' Forum*. 2011.

[19] Wang, David, et al. "DRAMsim: a memory system simulator." *ACM SIGARCH Computer Architecture News* 33.4 (2005): 100-107.

[20] Hara, Yuko, et al. "Proposal and quantitative analysis of the CHStone benchmark program suite for practical C-based high-level synthesis." *Journal of Information Processing* 17 (2009): 242-254.

[21] Ge, Rong, et al. "Powerpack: Energy profiling and analysis of high-performance systems and applications." *Parallel and Distributed Systems, IEEE Transactions on* 21.5 (2010): 658-671.

[22] Kim, Yoongu, et al. "Thread cluster memory scheduling: Exploiting differences in memory access behavior." *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*. IEEE, 2010.

[23] Wilhelm, Reinhard, et al. "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems." *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 28.7 (2009): 966-978.

[24] Rixner, Scott, et al. *Memory access scheduling*. Vol. 28. No. 2. ACM, 2000.

[25] Smith, James E., and Andrew R. Pleszkun. *Implementation of precise interrupts in pipelined processors*. Vol. 13. No. 3. ACM, 1985.

[26] Hennessy, John L., and David A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[27] 2007 [Online]. Available: www.merasa.org, MERASA EU-FP7Project:

[28] Al-Omari, R., Arun K. Somani, and G. Manimaran. "Efficient overloading techniques for primary-backup scheduling in real-time systems." *Journal of Parallel and Distributed Computing* 64.5 (2004): 629-648.

[29] Subramaniam, Samantika, Milos Prvulovic, and Gabriel H. Loh. "PEEP: Exploiting predictability of memory dependences in SMT processors." *High Performance Computer Architecture, 2008. HPCA 2008. IEEE 14th International Symposium on*. IEEE, 2008.

[30] Li, Xianfeng, Abhik Roychoudhury, and Tulika Mitra. "Modeling out-of-order processors for WCET analysis." *Real-Time Systems* 34.3 (2006): 195-227.

[31] Kim, Dongki, et al. "A quantitative analysis of performance benefits of 3D die stacking on mobile and embedded SoC." *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*. IEEE, 2011.

[32] Cho, Su-Jin, et al. "Performance analysis of multi-bank DRAM with increased clock frequency." *Circuits and Systems (ISCAS), 2012 IEEE International Symposium on*. IEEE, 2012.

[33] Elhelw, Amr S., Ali El Moursy, and Hossam Fahmy. "Time-Based Least Memory Intensive Scheduling." *Embedded Multicore/Manycore SoCs (MCSoc), 2014 IEEE 8th International Symposium on*. IEEE, 2014.

[34] El-Moursy, Ali A., Walid El-Reedy, and Hossam AH Fahmy. "Fair memory access scheduling algorithms for multicore processors." *International Journal of Parallel, Emergent and Distributed Systems* ahead-of-print (2014): 1-23.

[35] Bourgade, Roman, et al. "Accurate analysis of memory latencies for WCET estimation." *16th International Conference on Real-Time and Network Systems (RTNS 2008)*. 2008.

[36] Wilhelm, Reinhard, et al. "Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems." *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on* 28.7 (2009): 966-978.

[37] Reineke, Jan, et al. "PRET DRAM controller: Bank privatization for predictability and temporal isolation." *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2011.

[38] Pellizzoni, Rodolfo, and Marco Caccamo. "Impact of peripheral-processor interference on WCET analysis of real-time embedded systems."*Computers, IEEE Transactions on* 59.3 (2010): 400-415.

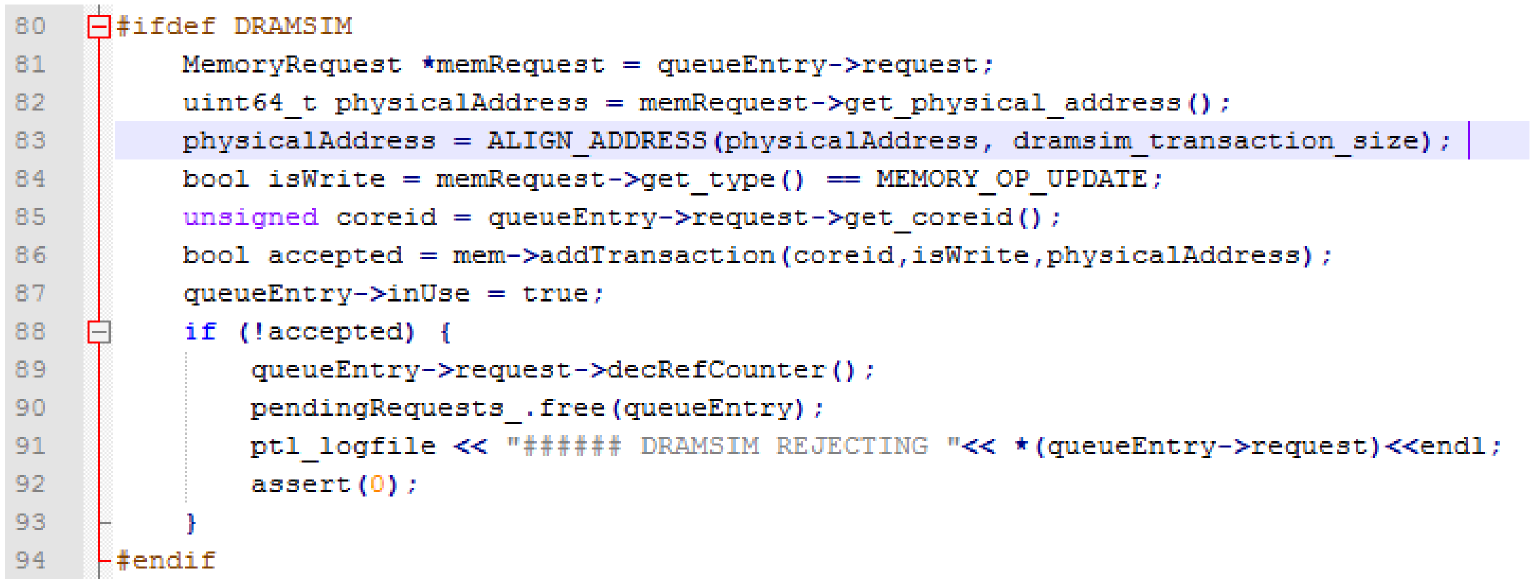
[39] Muralidhara, Sai Prashanth, et al. "Reducing memory interference in multicore systems via application-aware memory channel partitioning."*Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, 2011.

[40] Kim, Yoongu, et al. "ATLAS: A scalable and high-performance scheduling algorithm for multiple memory controllers." *High Performance Computer Architecture (HPCA), 2010 IEEE 16th International Symposium on*. IEEE, 2010.

[41] El-Moursy, Ali, et al. "Compatible phase co-scheduling on a CMP of multi-threaded processors." *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006.

Appendix A

1 MARRS DRAMSim Integration Code

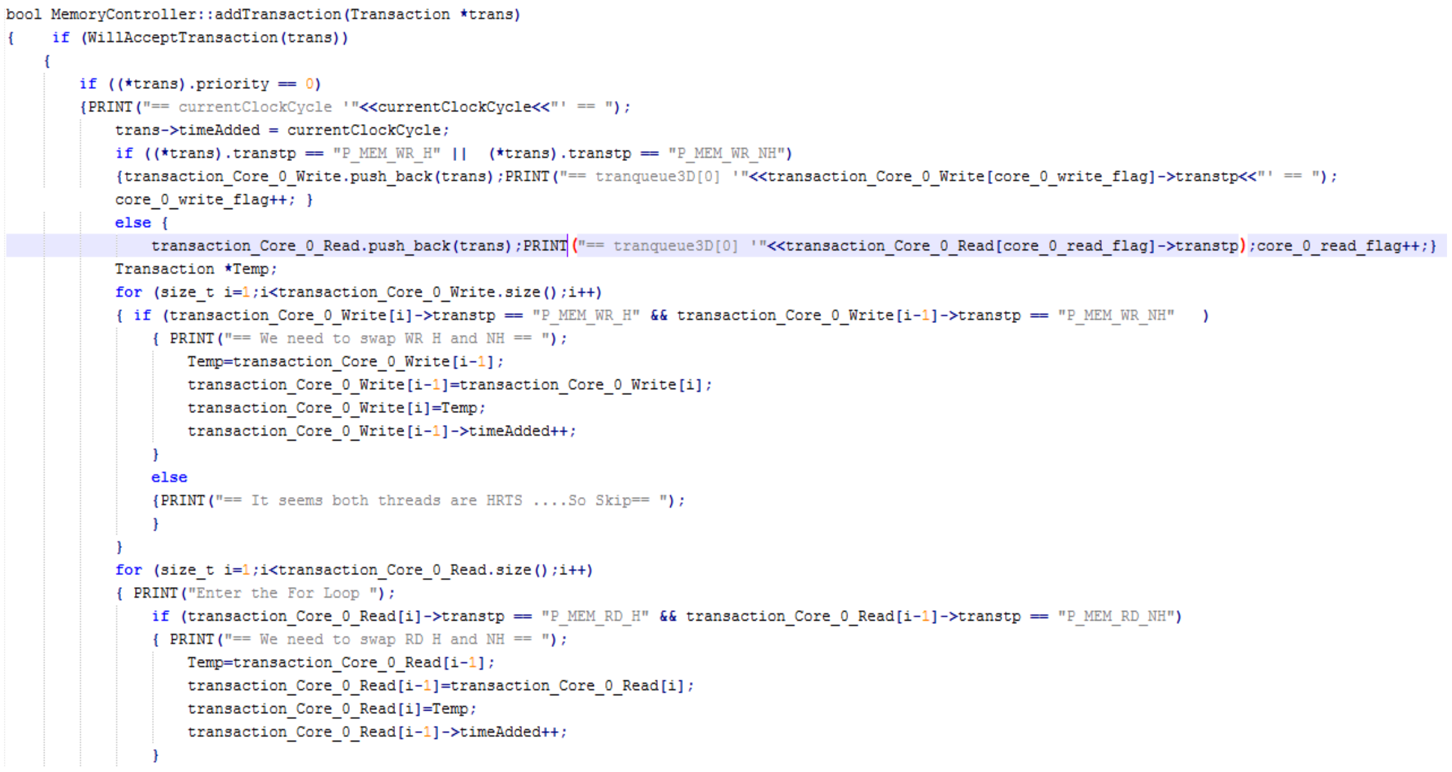


```
#ifdef DRAMSIM
    MemoryRequest *memRequest = queueEntry->request;
    uint64_t physicalAddress = memRequest->get_physical_address();
    physicalAddress = ALIGN_ADDRESS(physicalAddress, dramsim_transaction_size);
    bool isWrite = memRequest->get_type() == MEMORY_OP_UPDATE;
    unsigned coreid = queueEntry->request->get_coreid();
    bool accepted = mem->addTransaction(coreid,isWrite,physicalAddress);
    queueEntry->inUse = true;
    if (!accepted) {
        queueEntry->request->decRefCounter();
        pendingRequests_.free(queueEntry);
        ptl_logfile << "###### DRAMSIM REJECTING "<< *(queueEntry->request)<<endl;
        assert(0);
    }
#endif
```

Figure A.1 MARSS DRAMSim2 integration

2 DRAMSim

The below function shows the developed code to add the core transaction to private buffers and apply the swapping function that swaps the HRT request with NHRT requests

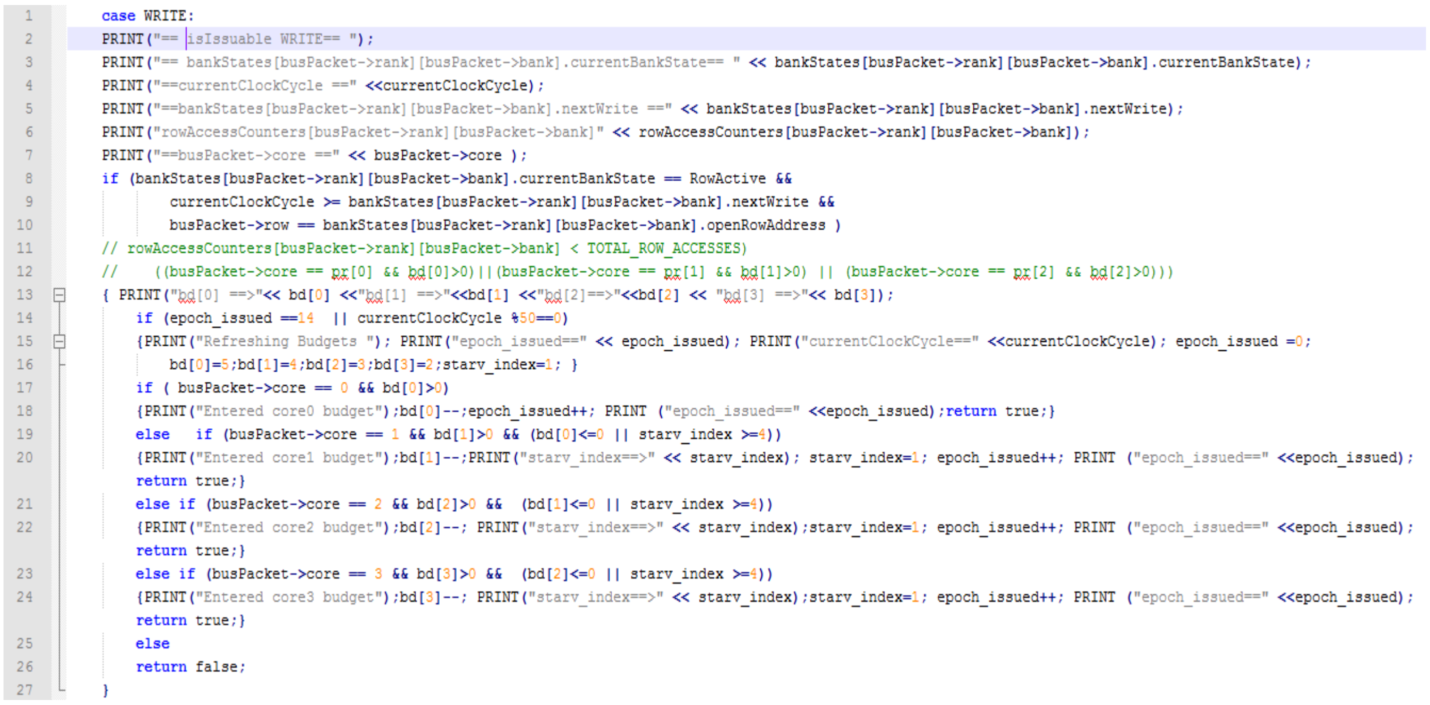


```
bool MemoryController::addTransaction(Transaction *trans)
{    if (WillAcceptTransaction(trans))
     {
         if ((*trans).priority == 0)
         {PRINT("== currentClockCycle '"<<currentClockCycle<<"' == ");
             trans->timeAdded = currentClockCycle;
             if ((*trans).transtp == "P_MEM_WR_H" ||  (*trans).transtp == "P_MEM_WR_NH")
             {transaction_Core_0_Write.push_back(trans);PRINT("== tranqueue3D[0] '"<<transaction_Core_0_Write[core_0_write_flag]->transtp<<"' == ");
             core_0_write_flag++; }
             else {
                 transaction_Core_0_Read.push_back(trans);PRINT("== tranqueue3D[0] '"<<transaction_Core_0_Read[core_0_read_flag]->transtp);core_0_read_flag++;}
             Transaction *Temp;
             for (size_t i=1;i<transaction_Core_0_Write.size();i++)
             { if (transaction_Core_0_Write[i]->transtp == "P_MEM_WR_H" && transaction_Core_0_Write[i-1]->transtp == "P_MEM_WR_NH"   )
                 { PRINT("== We need to swap WR H and NH == ");
                     Temp=transaction_Core_0_Write[i-1];
                     transaction_Core_0_Write[i-1]=transaction_Core_0_Write[i];
                     transaction_Core_0_Write[i]=Temp;
                     transaction_Core_0_Write[i-1]->timeAdded++;
                 }
                 else
                 {PRINT("== It seems both threads are HRTS ....So Skip== ");
                 }
             }
             for (size_t i=1;i<transaction_Core_0_Read.size();i++)
             { PRINT("Enter the For Loop ");
                 if (transaction_Core_0_Read[i]->transtp == "P_MEM_RD_H" && transaction_Core_0_Read[i-1]->transtp == "P_MEM_RD_NH")
                 { PRINT("== We need to swap RD H and NH == ");
                     Temp=transaction_Core_0_Read[i-1];
                     transaction_Core_0_Read[i-1]=transaction_Core_0_Read[i];
                     transaction_Core_0_Read[i]=Temp;
                     transaction_Core_0_Read[i-1]->timeAdded++;
                 }
```

Figure A.2 MCES Re-ordering requests in transaction queue

3 Issuable Part

The below code is the issuable part in the Global queue.

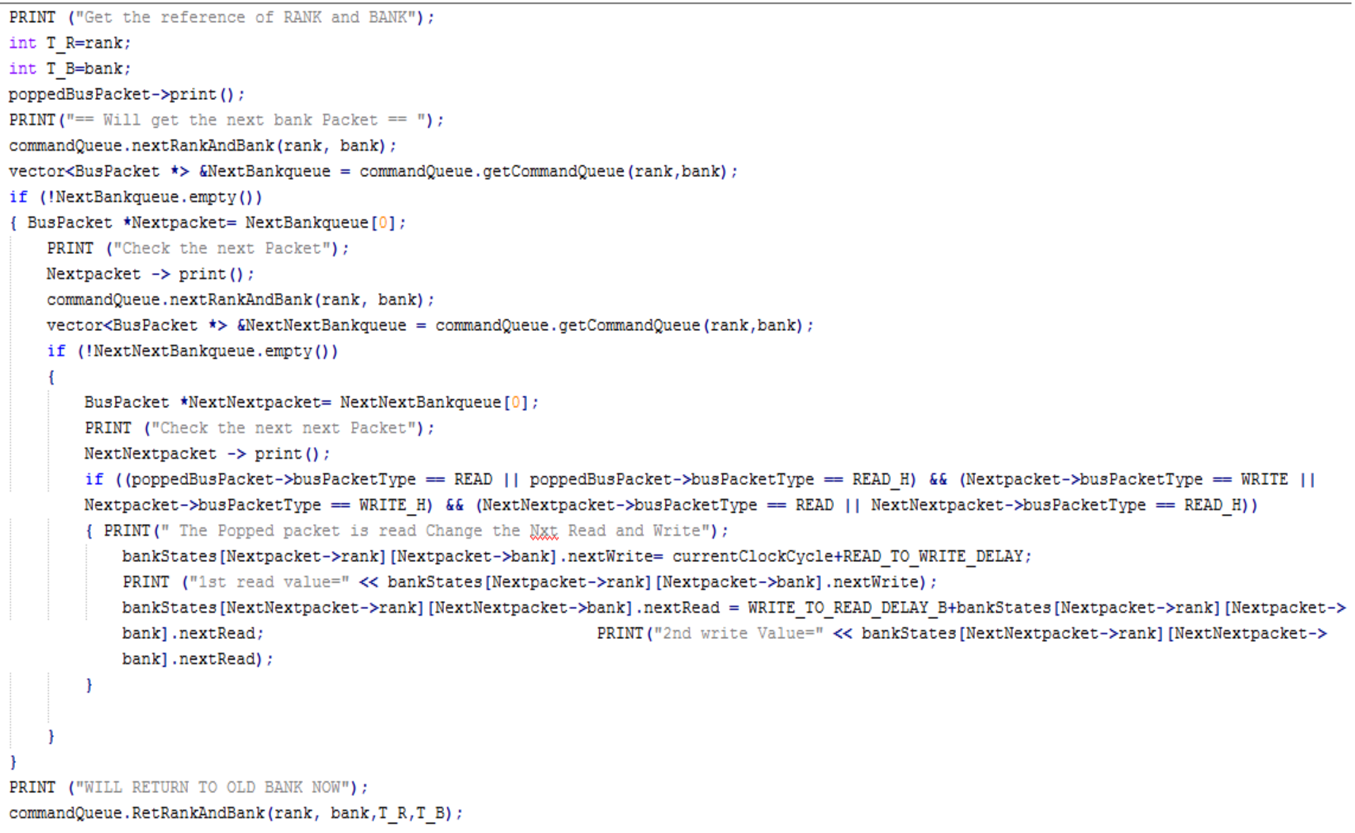


```
case WRITE:
PRINT("== isIssuable WRITE== ");
PRINT("== bankStates[busPacket->rank][busPacket->bank].currentBankState== " << bankStates[busPacket->rank][busPacket->bank].currentBankState);
PRINT("==currentClockCycle ==" <<currentClockCycle);
PRINT("==bankStates[busPacket->rank][busPacket->bank].nextWrite ==" << bankStates[busPacket->rank][busPacket->bank].nextWrite);
PRINT("rowAccessCounters[busPacket->rank][busPacket->bank]" << rowAccessCounters[busPacket->rank][busPacket->bank]);
PRINT("==busPacket->core ==" << busPacket->core );
if (bankStates[busPacket->rank][busPacket->bank].currentBankState == RowActive &&
        currentClockCycle >= bankStates[busPacket->rank][busPacket->bank].nextWrite &&
        busPacket->row == bankStates[busPacket->rank][busPacket->bank].openRowAddress )
// rowAccessCounters[busPacket->rank][busPacket->bank] < TOTAL_ROW_ACCESSES)
//    ((busPacket->core == pr[0] && bd[0]>0)||(busPacket->core == pr[1] && bd[1]>0) || (busPacket->core == pr[2] && bd[2]>0)))
{ PRINT("bd[0] ==>"<< bd[0] <<"bd[1] ==>"<<bd[1] <<"bd[2]==>"<<bd[2] << "bd[3] ==>"<< bd[3]);
    if (epoch_issued ==14  || currentClockCycle %50==0)
    {PRINT("Refreshing Budgets "); PRINT("epoch_issued==" << epoch_issued); PRINT("currentClockCycle==" <<currentClockCycle); epoch_issued =0;
        bd[0]=5;bd[1]=4;bd[2]=3;bd[3]=2;starv_index=1; }
    if ( busPacket->core == 0 && bd[0]>0)
    {PRINT("Entered core0 budget");bd[0]--;epoch_issued++; PRINT ("epoch_issued==" <<epoch_issued);return true;}
    else   if (busPacket->core == 1 && bd[1]>0 && (bd[0]<=0 || starv_index >=4))
    {PRINT("Entered core1 budget");bd[1]--;PRINT("starv_index==>" << starv_index); starv_index=1; epoch_issued++; PRINT ("epoch_issued==" <<epoch_issued);
    return true;}
    else if (busPacket->core == 2 && bd[2]>0 &&  (bd[1]<=0 || starv_index >=4))
    {PRINT("Entered core2 budget");bd[2]--; PRINT("starv_index==>" << starv_index);starv_index=1; epoch_issued++; PRINT ("epoch_issued==" <<epoch_issued);
    return true;}
    else if (busPacket->core == 3 && bd[3]>0 &&  (bd[2]<=0 || starv_index >=4))
    {PRINT("Entered core3 budget");bd[3]--; PRINT("starv_index==>" << starv_index);starv_index=1; epoch_issued++; PRINT ("epoch_issued==" <<epoch_issued);
    return true;}
    else
    return false;
}
```

Figure A.3 MCES Issuing requests

4 WCAD 4th Algorithm rule

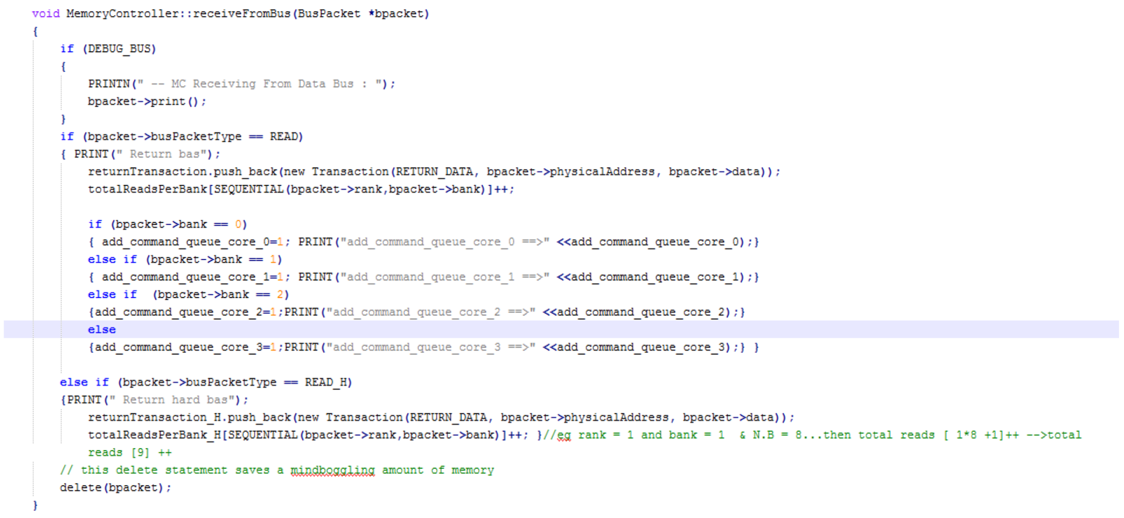
The below code is the coded WCAD 4th rule to apply fairness to starved request



```
PRINT ("Get the reference of RANK and BANK");
int T_R=rank;
int T_B=bank;
poppedBusPacket->print();
PRINT("== Will get the next bank Packet == ");
commandQueue.nextRankAndBank(rank, bank);
vector<BusPacket *> &NextBankqueue = commandQueue.getCommandQueue(rank,bank);
if (!NextBankqueue.empty())
{ BusPacket *Nextpacket= NextBankqueue[0];
    PRINT ("Check the next Packet");
    Nextpacket -> print();
    commandQueue.nextRankAndBank(rank, bank);
    vector<BusPacket *> &NextNextBankqueue = commandQueue.getCommandQueue(rank,bank);
    if (!NextNextBankqueue.empty())
    {
        BusPacket *NextNextpacket= NextNextBankqueue[0];
        PRINT ("Check the next next Packet");
        NextNextpacket -> print();
        if ((poppedBusPacket->busPacketType == READ || poppedBusPacket->busPacketType == READ_H) && (Nextpacket->busPacketType == WRITE ||
        Nextpacket->busPacketType == WRITE_H) && (NextNextpacket->busPacketType == READ || NextNextpacket->busPacketType == READ_H))
        { PRINT(" The Popped packet is read Change the Nxt Read and Write");
            bankStates[Nextpacket->rank][Nextpacket->bank].nextWrite= currentClockCycle+READ_TO_WRITE_DELAY;
            PRINT ("1st read value=" << bankStates[Nextpacket->rank][Nextpacket->bank].nextWrite);
            bankStates[NextNextpacket->rank][NextNextpacket->bank].nextRead = WRITE_TO_READ_DELAY_B+bankStates[Nextpacket->rank][Nextpacket->
            bank].nextRead;                                  PRINT("2nd write Value=" << bankStates[NextNextpacket->rank][NextNextpacket->
            bank].nextRead);
        }

    }
}
PRINT ("WILL RETURN TO OLD BANK NOW");
commandQueue.RetRankAndBank(rank, bank,T_R,T_B);
```

Figure A.4 WCAD Scheduling Code



```
void MemoryController::receiveFromBus(BusPacket *bpacket)
{
    if (DEBUG_BUS)
    {
        PRINTN(" -- MC Receiving From Data Bus : ");
        bpacket->print();
    }
    if (bpacket->busPacketType == READ)
    { PRINT(" Return bas");
        returnTransaction.push_back(new Transaction(RETURN_DATA, bpacket->physicalAddress, bpacket->data));
        totalReadsPerBank[SEQUENTIAL(bpacket->rank,bpacket->bank)]++;

        if (bpacket->bank == 0)
        { add_command_queue_core_0=1; PRINT("add_command_queue_core_0 ==>" <<add_command_queue_core_0);}
        else if (bpacket->bank == 1)
        { add_command_queue_core_1=1; PRINT("add_command_queue_core_1 ==>" <<add_command_queue_core_1);}
        else if  (bpacket->bank == 2)
        {add_command_queue_core_2=1;PRINT("add_command_queue_core_2 ==>" <<add_command_queue_core_2);}
        else
        {add_command_queue_core_3=1;PRINT("add_command_queue_core_3 ==>" <<add_command_queue_core_3);} }

    else if (bpacket->busPacketType == READ_H)
    {PRINT(" Return hard bas");
        returnTransaction_H.push_back(new Transaction(RETURN_DATA, bpacket->physicalAddress, bpacket->data));
        totalReadsPerBank_H[SEQUENTIAL(bpacket->rank,bpacket->bank)]++; }//eg rank = 1 and bank = 1  & N.B = 8...then total reads [ 1*8 +1]++ -->total
        reads [9] ++
    // this delete statement saves a mindboggling amount of memory
    delete(bpacket);
}
```

Figure A.5 WCAD applied rules

الملخص

في الوقت الحاضر تصبح رقاقات متعددة المعالجات الحديثة اكثر تطلبا بسبب الأداء العالي بهم وخاصة في انظمة الوقت الحقيقي نظرا لانخفاض الطاقة اللازمه للتشغيل والأداء العالي. على الجانب الأخر, يجب تحقيق اعلى كفاءة و عداله كافيه بين انظمة التشغيل. فى انظمة الزمن الحقيقي تقاس الجوده بالوقت التقديري لأسوأ حاله. فى الأنظمه الحديثه يتم الاعتماد على معدل مزدوج البيانات فى الذاكره الحديثه. نقترح تحكم في الذاكرة الجديدة التي تعطي الأولوية و تعين حصص ثابته للنوى و بالتالى يمكن دراسة التطبيقات في الوقت الحقيقي الوسائط المتعددة والتطبيقات في الوقت الحقيقي الصعبة.

مهنـــــدس: أحمد شفيق شافعى محمد

تاريخ الميلاد :

الجنسيه: مصري

تاريخ التسجيل: 2010 /10/01

تـاريخ المنـح: / / 2016

القســــم: هندسةالألكترونيات و الأتصالات الكهربية

الدرجــــة: ماجيستير العلوم

المشرفون: أ.د. حسام علي حسن فهمي

أ.م.د. علي علي علي المرسي أستاذ مساعد بقسم الحاسبات و الأنظمه

الممتحنــون : أ.د. حازم محمود عباس كلية الهندسة , جامعة عين شمس

أ.د. محمد رياض الغنيمي

أ.د. حسام علي حسن فهمي

أ.م.د. علي علي علي المرسي أستاذ مساعد بقسم الحاسبات و الأنظمه

عنـوان الرسالـة :

دائرة التحكم في ذاكرة أنظمة الزمن الحقيقي متعددة النواة

الكلمات الدالة : – منظمة الذاكرة , الزمن الحقيقى , دائرة متعددة النواه , الأنظمة المدمجة

ملخـــص البحـــث :

في الوقت الحاضر تصبح رقاقات متعددة المعالجات الحديثة اكثر تطلبا بسبب الأداء العالي بهم وخاصة في انظمة الوقت الحقيقي نظرا لانخفاض الطاقة اللازمه للتشغيل والأداء العالي. على الجانب الأخر, يجب تحقيق اعلى كفاءة و عداله كافيه بين انظمة التشغيل. فى انظمة الزمن الحقيقي تقاس الجوده بالوقت التقديري لأسوأ حاله. فى الأنظمه الحديثه يتم الاعتماد على معدل مزدوج البيانات فى الذاكره الحديثه. نقترح تحكم في الذاكرة الجديدة التي تعطي الأولوية و تعين حصص ثابته للنوى و بالتالى يمكن دراسة التطبيقات في الوقت الحقيقي الوسائط المتعددة والتطبيقات في الوقت الحقيقي الصعبة.

دائرة التحكم في ذاكرة أنظمة الزمن الحقيقي متعددة النواة

اعداد

أحمد شفيق شافعى محمد

رساله مقدمه الى كليه الهندسه– جامعة القاهرة

كجزء من متطلبات الحصول على درجة ماجيستير العلوم

فى

هندسة الألكترونيات و الأتصالات الكهربيه

<u>يعتمد من لجنة الممتحنين:</u>

الاستاذ الدكتور: حسام على حسن فهمى	المشرف الرئيسى
الاستاذ الدكتور: علي علي علي المرسي	عضو
الاستاذ الدكتور:محمد رياض الغنيمي	الممتحن الداخلي
الاستاذ الدكتور: حازم محمود عباس كلية الهندسة , جامعة عين شمس	الممتحن الخارجي

كلية الهندسه–جامعة القاهرة

الجيزة–جمهورية مصر العربية

سنة

2016

دائرة التحكم في ذاكرة أنظمة الزمن الحقيقي متعددة النواة

اعداد

أحمد شفيق شافعى محمد

رساله مقدمه الى كليه الهندسه– جامعة القاهرة

كجزء من متطلبات الحصول على درجة ماجيستير العلوم

فى

هندسة الألكترونيات و الأتصالات الكهربيه

تحت اشراف

د. على على على المرسى	أ.د. حسام على حسن فهمى
....	....
أستاذ مساعد بقسم الحاسبات و الأنظمه	أستاذ بقسم الالكترونيات و الاتصالات
معهد بحوث الاكترونيات بالقاهرة	كليه الهندسة –جامعة القاهرة

كلية الهندسه–جامعة القاهرة

الجيزة–جمهورية مصر العربية

سنة

2016

دائرة التحكم في ذاكرة أنظمة الزمن الحقيقي متعددة النواة

اعداد

أحمد شفيق شافعى محمد

رساله مقدمه الى كليه الهندسه– جامعة القاهرة

كجزء من متطلبات الحصول على درجة ماجيستير العلوم

فى

هندسة الألكترونيات و الأتصالات الكهربيه

كلية الهندسه–جامعة القاهرة

الجيزة–جمهورية مصر العربية

سنة

2016