Design of time-predictable architecture and DDR memory bank allocation for COTS multi-cores

Vasudevan, Sriram

2020

Vasudevan, S. (2020). Design of time-predictable architecture and DDR memory bank allocation for COTS multi-cores. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/143308>

<https://doi.org/10.32657/10356/143308>

This work is licensed under a Creative Commons Attribution-NonCommercial 4.0 International License (CC BY-NC 4.0).

Downloaded on 26 Apr 2023 18:36:26 SGT

**NANYANG
TECHNOLOGICAL
UNIVERSITY**

SINGAPORE

**Design of Time-Predictable Architecture and DDR Memory Bank
Allocation for COTS Multi-cores**

**SRIRAM VASUDEVAN
Interdisciplinary Graduate School
ERI@N**

Design of Time-Predictable Architecture and DDR Memory Bank Allocation for COTS Multi-cores

SRIRAM VASUDEVAN

**Interdisciplinary Graduate School
ERI@N**

A thesis submitted to the Nanyang Technological University in partial fulfillment
of the requirement for the degree of
Doctor of Philosophy

2020

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

[Input Date Here]

27/03/2020

.....
Date

V. Sri Ram

[Input Signature Here]

SRIRAM VASUDEVAN

.....
[Input Name Here]

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

[Input Date Here]

27/03/2020

.....
Date

[Input Supervisor Signature Here]

DR. ARVIND EASWARAN

.....
[Input Supervisor Name Here]

Authorship Attribution Statement

This thesis contains material from 1 paper published in the following peer-reviewed journal(s) where I was the corresponding author.

The idea behind Chapter 5 is published as N. Shivaraman, S. Vasudevan and A. Easwaran, "Demo Abstract: Predictable SoC Architecture Based on COTS Multi-Core," *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, Vienna, 2016, pp. 1-1.

The contributions of the co-authors are as follows:

- Associate Professor Dr.Arvind Easwaran provided the initial project direction and edited the manuscript drafts.
- I prepared the manuscript drafts which was reviewed by Nitin Shivaraman.
- I studied the literature and designed the architecture of the SoC.
- I also designed the state machines which are the building blocks of the architecture.
- Nitin Shivaraman translated my design into hardware implementation.
- I also analyzed the data obtained from the tests.

27/03/2020

.....
Date

.....
Sriram Vasudevan

Acknowledgements

My passion to pursue my doctoral degree was made possible by the excellent environment at NTU. During the pleasant years as a Ph.D student, I have received a lot of help and support from many people both in research and daily life without which this dissertation could not have been completed. I wish to express my sincere thanks to these people for their kindness.

Firstly, I would like to express my sincere gratitude to my advisor Prof.Arvind Easwaran for the continuous support of my Ph.D study and related research, for his patience, motivation, inspiration and immense knowledge. His guidance helped me to understand the importance of a Ph.D study and the tools and techniques to identify problems and solve them step by step. I will cherish every meeting, technical discussion and presentation with him for the rest of my life. I could not have imagined having a better advisor and mentor for my Ph.D study.

Besides my advisor, I would like to thank the rest of my thesis committee: Prof.Anupam Chattopadhyay, Prof.Douglas Maskell, and Dr.Anshuman Tripathi, for their insightful comments, constant encouragement and also posing hard questions which motivated me to widen my research from various perspectives.

My sincere thanks goes to Dr.Saravanan Ramanathan, who not only was my colleague during my Ph.D but also mentored me at the right time to solve several research problems and also guided me to complete my thesis. I would also like to thank all the colleagues and friends in CPS-research-group and Hardware and Embedded Systems Lab (HESL), Vijaya Kumar Sundar, Ng Jun Xian Daniel, Chiam Zhonglin, Nitin Shivaraman, Xiaozhe Gu, Gopalakrishna Hegde, Rahul Garg and many others, for the

Acknowledgments

past four years of good time. I really learnt a lot from you, and the delightful memory will never fade away. Without the precious support it would not be possible to conduct this research.

I would like to express my sincere gratitude to my parents who gave birth to me and love me unconditionally. My parents' trust and support all through my life has been a significant motivation for me to seek the life I want. Without their inspiration, I would not have been able to complete my studies at NTU. Finally, I would like to thank my wife who constantly motivated and supported me throughout my Ph.D.

Abstract

An increasing trend in the chip manufacturing industry is the adoption of multi-core processors. This overwhelming trend of using Commercial Off-The-Shelf (COTS) multi-core processors has benefited most of the industries that require high computing performance at low power and greater efficiency. However, real-time systems are those systems where not only the correctness of execution is critical but also the time at which this execution occurs is equally critical. The multi-core processors offer a very high computing performance owing to the increased number of processing cores, but do not guarantee a predictable Worst-Case Execution Time (WCET), which is defined as the maximum time an application can take to execute on a given hardware.

This uncertainty in the estimation of the WCET arises from a multitude of challenges which are inherently present in the multi-core architecture. As a first step, these challenges are identified and then further focused on exploring a representative multi-core processor. The primary challenges identified in this thesis are related to the sharing of hardware components in the multi-core architecture which includes the Double Data Rate (DDR) memory and its controller, caches and system bus. A detailed architecture analyses of a COTS multi-core is performed along with a series of experiments which help to understand and quantify the unpredictability arising due to the sharing of hardware resources. This study not only helps to understand the challenges but also the severity of each challenge.

As a next step, to mitigate this primary challenge of shared memory subsystem in the multi-cores, a novel hybrid System On Chip (SoC) architecture is proposed which couples COTS multi-core with a Field Programmable Gate Array (FPGA). The proposed architecture is designed on the Xilinx Zynq 706 platform. The key feature of this proposed architecture is that the shared hardware components which

causes the unpredictability inside the multi-core are disabled and are designed separately on the FPGA to aid predictable execution of applications. The thesis specifically focuses on one of the components in the shared memory subsystem, which is the DDR memory and its controller. A predictable memory controller is designed for this architecture to show that the ARM A9 multi-core can be coupled with the custom designed memory controller on the FPGA. The thesis also shows that the latency on the FPGA can be bounded theoretically and hence this architecture aids in predictable execution of applications.

As a final step, the thesis focuses on the DDR memory and its controller and address one of the key challenges which is DDR memory bank allocation to the cores. Although the execution time of an application can be bounded in a predictable memory controller, it is important to allocate only the required number of banks to the cores such that the worst-case execution time requirement of the application is satisfied. This allocation determines the utilization of the DDR memory. Hence, it is important that the number of banks should not be overloaded for any application. The thesis specifically focus on two scenarios. One, where the banks of the DDR are shared by the cores and the other where the banks are private to the cores. In each of the scenario, an optimal bank allocation policy is presented. The key significance of this contribution is the fact that the algorithms presented are optimal and always return the best allocation whenever feasible. Thus, in this dissertation an outline to use the multi-cores in a predictable manner for real-time systems has been provided.

Table of Contents

Acknowledgments	I
Abstract	III
List of Figures	IX
List of Tables	XII
List of Abbreviations	XIII
1 Introduction	1
1.1 Real-Time System	1
1.2 Motivation and Problem Statement	3
1.3 Contributions and Organization	4
2 Background	6
2.1 Background - Multi-cores	6
2.1.1 Generic COTS Multi-core Architecture	6
2.1.2 WCET Analysis	10
2.1.3 Challenges in Estimating WCET on multi-cores	13
3 Related Work	16
3.1 Introduction	16
3.2 Analysis of COTS hardware and unmodified middleware	17
3.2.1 Cache Analysis	18
3.2.2 Interconnect Analysis	19
3.2.3 DDR controller Analysis	21

TABLE OF CONTENTS

3.2.4	Cache Locking and Partitioning	21
3.3	Unmodified COTS hardware and modified middleware	22
3.4	Custom hardware and unmodified middleware	24
3.4.1	Cache Designs	24
3.4.2	ScratchPad Memories (SPM) Designs	26
3.4.3	Comparison of different Memory Architectures	27
3.5	Memory Controllers	28
3.6	Interconnect Designs	31
3.7	Conclusion	33
4	Challenges in using COTS multi-core processors in real-time systems	34
4.1	Motivation	34
4.2	Contributions of this work	34
4.3	Code Warrior TAP	36
4.4	P4080 configuration	37
4.5	Generic Experimental Setup	39
4.6	Experiment I: Contentions in DDR controller	41
4.6.1	Description	41
4.6.2	Result and Analysis	41
4.7	Experiment II: CPC contentions in partitioned CPC controller	43
4.7.1	Description	43
4.7.2	Results	44
4.8	Experiment III: CPC contentions in non partitioned CPC controller	46
4.8.1	Description	46
4.8.2	Results	46
4.9	Experiment IV: Coherency Contentions in CCF	48
4.9.1	Description	48
4.9.2	Results	48
4.10	Experiment V: Cache evictions in non-partitioned CPC controller	49
4.10.1	Description	49

TABLE OF CONTENTS

4.10.2 Results	50
4.11 Experiment VI: Cache evictions in partitioned CPC controller	53
4.11.1 Description	53
4.11.2 Results	55
4.12 Conclusion	55
5 Time-Predictable System-On-Chip architecture for COTS multi-cores	59
5.1 Introduction	59
5.2 Platform Overview	61
5.3 Proposed Architecture	64
5.3.1 Design Overview	64
5.3.2 Arbiter-Controller Framework	65
5.3.3 DDR controller Design	68
5.3.3.1 Salient Features of DDR controller	68
5.3.3.2 DDR controller State Machines	69
5.3.3.3 Controller Policies	72
5.4 Implementation Details	76
5.5 Analysis of worst-case latency of the implemented controller	78
5.6 Test results and challenges in testing the architecture	82
5.7 Conclusion	90
6 Bank Allocation Algorithm for requestors	91
6.1 Motivation	91
6.2 Background	92
6.2.1 DDR Memory Mapping	92
6.2.2 Memory controller Architecture	92
6.2.3 Formalization and Notations	95
6.3 Model for Requestors, DDR and Bank Allocation	98
6.4 Private bank allocation	99
6.4.1 Latency Equation for private bank allocation	99
6.4.2 Algorithm to Allocate Private banks to Requestors	101

TABLE OF CONTENTS

6.5	Shared bank allocation	105
6.5.1	Latency Equation for shared bank allocation	106
6.5.2	Shared bank Allocation	107
6.6	Conclusion	112
7	Conclusions and Future Work	113
7.1	Summary of Contributions	113
7.2	Future Work	115
7.2.1	Further analyses of Freescale P4080	115
7.2.2	Improvement to the SoC-COTS architecture	116
7.2.3	Optimal Bank Allocation Algorithm when banks can be either private or shared between requestors	117
	References	119

List of Figures

2.1	Generic multi-core processor architecture	7
2.2	Architecture of P4080 [1]	8
2.3	DRAM architecture	9
3.1	Work towards predictable WCET	18
4.1	CodeWarrior TAP (source: 4.1)	36
4.2	P4080 non-partitioned memory hierarchy	37
4.3	P4080 partitioned memory hierarchy	38
4.4	P4080DS Hardware setup	39
4.5	Contentions in non-partitioned DDR	42
4.6	Contentions in partitioned DDR	43
4.7	Contentions in partitioned CPC	45
4.8	Contentions in partitioned CPC	45
4.9	Contentions in non-partitioned CPC	47
4.10	Contentions in non-partitioned CPC	47
4.11	Coherency contentions in non-partitioned CPC	49
4.12	Contentions in non-partitioned CPC for 1 MB, write through and fifo	51
4.13	Contentions in non-partitioned CPC for 1 MB, write back and fifo	51
4.14	Contentions in non-partitioned CPC for 1 MB, write through and plru	52
4.15	Contentions in non-partitioned CPC for 1 MB, write back and plru	52
4.16	Curve fitting coefficients for 1 MB data size	52

LIST OF FIGURES

4.17	Contentions in non-partitioned CPC for 2 MB, write through and fifo	53
4.18	Contentions in non-partitioned CPC for 2 MB, write back and fifo	54
4.19	Contentions in non-partitioned CPC for 2 MB, write through and plru	54
4.20	Contentions in non-partitioned CPC for 2 MB, write back and plru	55
4.21	Curve fitting coefficients for 2 MB data size	55
4.22	Contentions in partitioned CPC for 256KB, write back and plru	56
5.1	Zynq Platform Overview	60
5.2	Xilinx Zynq Architecture(source: Xilinx Zynq TRM) [2]	62
5.3	PS-PL Request-response Path	63
5.4	Predictable SoC architecture	63
5.5	Arbiter Controller Framework	65
5.6	Arbiter with AXI interfaces	66
5.7	Custom DDR Controller	67
5.8	Bank Controller FSM	69
5.9	Bank Specific FSM	71
5.10	Posted CAS and Normal CAS [3]	72
5.11	Refresh Control Logic	73
5.12	Memory Mapping	74
5.13	Command Generation	75
5.14	Controller Front-end Implementation	75
5.15	Controller Core Implementation	76
5.16	Controller Back-end Implementation	77
5.17	Read Request from the Core to the DDR	77
5.18	Memory Controller serving Read Request	78
5.19	Write Request from the Core to the DDR	79
5.20	Memory Controller serving Write Request	79
5.21	Experiment Setup	82
5.22	Experiments for the architecture	84
5.23	Memory Read Request to PS DDR - Single Core	84

LIST OF FIGURES

5.24	Memory Read Request to PL DDR - Single Core	85
5.25	Memory Write Request to PS DDR - Single Core	86
5.26	Memory Write Request to PL DDR - Single Core	86
5.27	Memory Read Request to PS DDR - Dual Core	87
5.28	Memory Read Request to PL DDR - Dual Core	88
5.29	Memory Write Request to PS DDR - Dual Core	88
5.30	Memory Write Request to PL DDR - Dual Core	89
6.1	Memory Controller Architecture [5]	93
6.2	Timing Dependencies in Dynamic Scheduling	95
7.1	Predictable COTS-FPGA Architecture	116

List of Tables

2.1	DRAM Signals (source:JEDEC Standards)	10
2.2	DDR Commands (source:JEDEC Standards)	10
2.3	Timing Constraints (source:JEDEC standards)	11
3.1	Real-time Memory controllers	29
6.1	Basic Notations	94
6.2	Latency of requestors under different scenarios	109

List of Abbreviations

RTS	Real-time Systems
RTA	Real-time Applications
FPGA	Field Programmable Gate Array
CPS	Cyber Physical Systems
ASIC	Application Specific Integrated Circuits
RT CPS	Real-Time Cyber Physical Systems
ECU	Electronic Control Unit
IC	Integrated Circuits
WCET	Worst-Case Execution Time
COTS	Commercial Off The Shelf
SWaP	Size Weight and Power
DDR	Dual Data Rate
MMU	Memory Management Unit
LRU	Least Recently Used

Abstract

FIFO	First In First Out
MRU	Most Recently Used
CCF	Corenet Coherency Fabric
FIFO	First In First Out
MESI	Modified Exclusive Shared Invalid
TDMA	Time Division Multiple Access
DRAM	Dynamic Random Access Memory
CPC	Corenet Platform Cache
SoC	System On Chip
AXI	Advanced eXtensible Interface
AMBA	Advanced Micro-controller Bus Architecture
PS	Processing System
PL	Programmable Logic
SCU	Snoop Controller Unit
FSM	Finite State Machine
JEDEC	Joint Electron Device Engineering Council

Chapter 1

Introduction

1.1 Real-Time System

Cyber-physical systems (CPS) are embedded computing systems in which the cyber world of computation and communication closely interacts with the physical world of sensors and actuators. These systems are abundant in many industries including avionics, automotive, health-care, consumer applications, etc. Many applications in CPS are real-time, meaning its correctness not only depends on its output but also the time within which the operation is performed. Therefore, these systems have to meet their timeliness requirements. Real-time Applications (RTA) are classified based on the consequence of not meeting their deadlines as hard, soft and firm [6]. Hard RTA are those in which missing a deadline will result in the failure of the entire system. Hence, these applications are safety-critical in nature. Examples of such RTA are anti-lock braking, adaptive cruise control, and inflation of airbags. Firm RTA are those in which infrequent deadline misses are tolerable but will degrade the system performance. The utility of the result after its deadline is zero. An example of firm RTA is storm forecast systems. Soft RTA are similar to firm RTA, in that infrequent deadline misses are tolerable, but the utility of the result after the deadline is not zero. Instead, it degrades as time passes after the deadline. An example of soft RTA is video conferencing.

The hardware commonly used to realize Real-Time Cyber Physical Systems (RT-CPS) are Application Specific Integrated Circuits (ASICs), which are electronic circuits customized for specific purposes [7]. These ICs typically contain a processing element having a single core, and several peripheral management units to access I/O devices. But the complexity and number of applications in RT-CPS is always on a steady rise, as a result of which there is a growing demand for ASICs that are more pow-

erful. For example, in a modern day car there are around 200 ASICs (called Electronic Control Units) networked together, serving a host of applications such as adaptive cruise control, anti-lock braking, battery management, collision detection and avoidance, parking assist, etc. More powerful ASICs will help to mitigate scalability issues arising from this trend of increasing number of applications by way of consolidation. Multi-core ASICs are ICs where several processing cores are packaged into one die, and they seem like a natural choice to satisfy this demand. One of the primary advantages of using multi-cores is that the die can fit into a single package, thus providing better Size Weight and Power (SWaP) characteristics than a system with multiple single-core processors.

One such typical homogeneous Commercial-off-the-shelf (COTS) multi-core processor from Freescale Semiconductors is the P4080. It comprises 8 cores that use shared interconnect and memory for inter-core communications (described under section 2 in reference manual [1]). The on-chip interconnect in P4080 is termed as “corenet coherency fabric”. The SWaP benefits of this processor arise from the fact that it shares a significant amount of hardware resources between the cores (caches, memory controllers, main memory, interconnect and access to input/outputs (I/O) and other peripherals).

COTS multi-cores offer significant advantages, but they are by design not suitable for use in RT-CPS. This is mainly due to the unpredictable nature of application execution on these processors. A typical COTS multi-core share several internal components and peripherals of which memory subsystem forms an integral part. The memory subsystem consists of the DDR memory and its controller, the shared caches and its controller and the system interconnect. The memory subsystem forms a key part of an application execution as every application requires processing of data which are stored in the memory. For example, consider a scenario where there are 8 cores and 2 memory controllers in a multi-core and there are 2 applications executing one on each core 1 and core 2. If both core 1 and core 2 applications require a memory access at the same instant to a memory location controlled by the same memory controller, then only one of the two application’s request can be processed. This implies that the execution time of one of the application can be delayed in the memory controller waiting for the request from the other application to be completed. A similar situation can arise in any of the other shared components such as the caches or the system interconnect. This implies that the unpredictability in the execution of application caused by the sharing of hardware resources forms the primary challenge in using multi-cores for RT-CPS.

An application in RT-CPS typically comprises several tasks with timeliness requirements. To be able to meet those requirements and provide guarantees, it is then essential to know apriori the maximum amount of execution time required for each task. This maximum amount, denoted as the Worst-Case Ex-

ecution Time (WCET) depends on various factors including application inputs, hardware architecture, tasks that are concurrently scheduled on the other cores in a multi-core processor, etc. The Certification Authorities Software Team (CAST) examined multi-cores (specifically dual-core processors) and identified issues that could affect the safety and performance of any software related to aircraft systems when executed on such processors [8]. The paper clearly identifies shared resources as a source of interference which leads to unpredictability in estimating WCET. It also presents a list of experiments that must be performed and results that need to be shown before using a multi-core hardware for safety-critical applications. Thus, estimation of WCET on multi-cores is a major challenge for adoption of these platforms for RT-CPS.

1.2 Motivation and Problem Statement

As the single core processors started to become obsolete, the industries that rely on these processors, were forced to adopt the more efficient, less power consuming and high performance multi-core processors. This augments well for most of the commercial industries such as those that rely on performance and efficiency alone (eg., commercial computers, mobile computing etc.). However, the real-time industry which specifically focuses on safety aspect resorted to a more viable option which is to use only one of the core and switch off all the others. This is extremely inefficient and considering that there is an ever increasing requirement of higher performance, there needs to be other alternatives to bridge this gap.

The real-time research community has been actively looking for solutions to address the unpredictability in the execution of applications in multi-cores. Looking into the existing contributions, it can be concluded that most of these targets solving the unpredictability in using the shared resources in the multi-core architecture. Also, most of the existing work target a specific shared resource rather than a unified approach which takes into account of the interaction between the different resources and in turn with the processing core itself. The key motivations for this dissertation are as follows.

1. Although there is a general consensus on the challenges due to shared hardware resources to obtain predictable execution on multi-cores, the severity and the impact of each shared resource under various available internal configurations of the multi-core have not yet been studied. This requires detailed analyses of a commercially available multi-core platform.
2. There have been solutions which address the challenge in every shared resource individually and there are solutions which build custom processors. In either case, the solutions do not target

COTS multi-core as the compatibility of these solutions for COTS multi-core are not established. This requires a re-look into the architectural design of a multi-core by retaining as much COTS component as possible.

3. There have been considerable efforts in designing custom memory controllers for real-time systems. However, these memory controllers do not explore the aspect of optimally allocating the memory resource (specifically the banks of a DDR memory) to the cores.

1.3 Contributions and Organization

The main technical contributions and organization of this dissertation are presented below:

- In Chapter 2, a brief background on the architecture of multi-core processors and its relevance in real-time industry is provided. The chapter further provides a background on worst-case execution time analysis which is a key component in any real-time system and also provide a detailed understanding of the challenges that the real-time industry faces in using these multi-core processors.
- In Chapter 3, an extensive study on the literature pertaining to the use of multi-cores in real-time systems is provided. Specifically, these contributions are separated into three categories, a rationale for this categorization is provided and also the alignment of the work in this dissertation with these existing contributions is also detailed.
- The Chapter 4, analyzes a commercially available multi-core processor (P4080DS), for its use in the real-time systems. A detailed architectural analyses and targeted experiments on the P4080DS which help identify the primary challenges that is briefly explained in Section 1.1 is presented. The key contribution of this chapter is that, it not only analyze these primary challenges at architectural level, but also quantify the extent of resource contention for every shared resource inside the multi-core.
- In order to mitigate one of the primary challenges that has been explained in Chapter 4, which is shared memory controllers, a novel multi-core System-On-Chip (SoC) architecture solution is presented in Chapter 5. This solution involves using a SoC platform to integrate the ARM A9 COTS multi-core with a Field Programmable Gate Array. The key idea behind this solution is that the shared hardware components (memory subsystem) inside the multi-core are disabled and

a predictable memory controller is implemented outside the multi-core, on the FPGA to handle the memory requests from the ARM cores. The architectural details, implementation methodology and also an analysis of the predictability of the architecture is presented.

- The Chapter 6, focuses on a specific challenge in one of the shared resource which is to optimally allocate banks of a DDR memory to the cores. The banks are allocated to the cores based on the worst-case execution time requirement of each application. Over allocating banks to a single core reduces the utilization of the DDR as there would be many unused memory locations. Hence, it is important to find the correct number of banks to be allocated. The contribution in this chapter is significant as it proposes optimal algorithms to allocate banks to cores under two scenarios. First is when the banks of the DDR are shared among all the cores and the second is when the banks are private to the cores.
- Chapter 7 summarizes the contributions of this dissertation and discusses potential directions for future research work.

Chapter 2

Background

This chapter provides a background on multi-core processor architecture and then proceed to explain the importance of WCET analysis. The chapter also explains the different types of analysis to estimate WCET and then proceeds to provide a classification of the major challenges that have been identified in estimating WCET with these processors.

2.1 Background - Multi-cores

2.1.1 Generic COTS Multi-core Architecture

“Multi-core is a design where a single processor has the core logic of more than one processor” [9]. The generic architecture of a multi-core is shown in Figure 2.1 which has ‘n’ cores with each core having one level of private cache (local to the core and represented as private L1 cache in Figure 2.1). All the cores share at least one level of cache (accessible to all the cores and represented as Shared L2 cache in Figure 2.1). Finally, there is a system interconnect which helps in the communication between the cores and the Double Data Rate (DDR) main memory. Let us consider an example COTS multi-core, in particular, the Freescale P4080 shown in Figure 2.2, to understand the architectural features. The P4080 processor has eight e500mc cores, each of which is pipelined and superscalar.

The e500mc core integrates two **instruction units**, one **floating point unit** and one **load/store unit** to process data and instructions. The core also includes on-chip first-level data and instruction Memory Management Unit (MMU), a second-level unified MMU and two levels of on-chip cache memories. The instruction unit interfaces the system interconnect and L1 instruction cache and is used to buffer instruc-

Figure 2.1: Generic multi-core processor architecture

tions that need to be executed. Memory unit consists of the MMU and a cache hierarchy. MMU consists of Translation Lookaside Buffers (TLB) that accelerate the virtual to physical addresses translation.

Caches are on-chip memories used by the processor to reduce the average time to access data and instructions from DDR memory. Caches may be implemented as independent or unified for handling data and instructions. Data caches are usually arranged as a hierarchy of several levels (L1, L2 and so on). The e500mc core has separate 32-KB, 8-way set associative L1 data and instruction caches. It also includes a 128-KB unified, 8-way set associative L2 cache. Whenever there is a write or read request to a location in the DDR memory, the initial step is to check the caches. If the requested content is already present in the cache, then the processor directly reads or writes to the cache. This is termed as a “cache hit”. On the other hand, when the cache does not have the data it is termed as a “cache miss”, and the requested data is brought into the cache from either the next level of caches or DDR memory. To create space for new data upon a cache miss, the caches may need to evict some existing data especially when they are full. This is referred to as the “cache replacement policy” that guides the eviction process. Many cache replacement policies have been developed such as First In First Out (FIFO), Most Recently Used (MRU), Least Recently Used (LRU), etc. [10].

System interconnect forms the infrastructure for connectivity and enables to implement multi-core systems that are coherent. For example in P4080, it is termed as “Corenet Coherency Fabric” (CCF),

Figure 2.2: Architecture of P4080 [1]

which is a bus infrastructure to interconnect cores, caches, memory subsystem, I/O devices and other peripheral devices.

Multi-cores possess another level of cache beyond the system interconnect that facilitates sharing of data across the cores. For example, in P4080 this is termed as “Corenet Platform Cache” (CPC), which is the L3 shared cache. The CPC functions as a general purpose cache and connects CCF to the memory controller. It is 1-MB and 32-way set associative. When data is shared between the cores and as a consequence there are multiple copies of data in the core specific caches, then the coherency protocol maintains consistency between these copies of data. P4080 supports MESI (Modified/Exclusive/Shared/Invalid) based coherency protocol [11].

Caches can be configured to handle data in two ways: inclusive and exclusive. For an inclusive cache, data in any lower cache level must also be present in every other higher level. On the other hand, exclusive caches do not require the data in L1 to be present in the other levels.

The **memory complex** of COTS multi-cores typically consists of DDR main memory and its controller. The purpose of a memory controller is to manage the flow of data to and from the DDR memory. It has digital logic that is required to read, write and refresh data in the DDR memory. As the number of cores increase, the controller should also manage requests from different cores by arbitrating among them. For example in P4080 two independent DDR memory controllers are connected to two DDRs each and handle requests from the eight cores.

Memory is an integral component of a computing system. Dynamic random-access memory (DRAM)

Figure 2.3: DRAM architecture

is being used as system main memory since 1975. This uses a single MOS transistor and a storage capacitor as primitive element to hold 1 bit of data. The memory controller is responsible for driving control signals to read and write data based on timing defined by Joint Electron Device Engineering Council (JEDEC) standards. This introduces added latency during data transfer as the system bus is synchronous. Hence, synchronous version of DRAM (SDRAM) was developed and standardized by JEDEC. DDR implies Double Data-Rate which is one of the improved versions of SDRAM. DDR SDRAMs allow movement of data on both rising and falling edge of clock signal, thereby effectively doubling the bandwidth.

A DDR is made of numerous 1-bit cells arranged in regular 2D grid structure as shown in Figure 2.3. The Figure 2.3 shows a single cell which is replicated as a 2 dimensional array. All word lines in a single row are tied together to form the row address lines and all bit lines in a column are connected electrically to form the column address. Such grids are accompanied with a row buffer and sense amplifiers to form a bank as shown in Figure 2.3. A Dual In-line Memory Module (DIMM) can have multiple banks in a single memory system.

Table 2.1 shows the DRAM interface signals with description as defined in JEDEC standards [12]. The Table 2.2 shows the list of most frequently used commands with the corresponding signals. The ACT command activates specified row from given bank by copying the data into row buffer. Now the row is said to be open. Once the row is opened, either a RD or a WR command can be sent to perform a memory read or a write. The PRE command closes the opened row by copying the data from the row buffer to opened row. The command bus is kept busy with NOP command when there are no other

DDR Signal	Type	Signal Function
CK, CK#	Input	Clock Inputs to DDR memory. Represents the two edges of the clock cycle in which the data is sampled
CKE	Input	Clock Enable Signal
CS#	Input	Chip Select Signal
ODT	Input	Termination Resistance to be connected to the DDR Memory
RAS#, CAS#, WE#	Input	Row and Column Access strobe, Write Enable
DM	Input	Write Input Data Mask Signal
BA	Input	Signal to denote Bank Address
A0-A15	Input	Signal for Address Bus
DQ	Input	Data Bus Signals
DQS, DQS#	Bi-Directional	Bi-directional Strobe Signal for Data
RESET#	Input	Reset Signal for DDR Memory

Table 2.1: DRAM Signals (source:JEDEC Standards)

Command	Description	Signaling		
		Command Bus	Address Bus	Data Bus
ACT	Activate	011	Bank, Row Address	-
NOP	No Operation	111	-	-
RD	Read	101	Column Address	Read Data
WR	Write	100	Column Address	Write Data
PRE	Precharge	010	Bank Address	-
REF	Refresh	001	Bank Address	-

Table 2.2: DDR Commands (source:JEDEC Standards)

commands to send. The REF commands must be sent with specific refresh interval to maintain data integrity. For each of this command the corresponding command bus signal is shown under the column “Command bus”. The content of the address and the data bus is also provided for each command. These commands must obey certain timing constraints for deterministic operation of DRAM. The JEDEC standard specifies these constraints. Major timing constraints are listed in Table 2.3.

2.1.2 WCET Analysis

Worst-Case Execution Time (WCET) is defined as the maximum time that an application takes to execute on a specific hardware. Consider the application that inflates airbags when a car is involved in a high-impact accident. For safety during a car accident, the airbags need to inflate within a specific time interval. The airbag crash sensor needs to pass information to the controller about a crash, and the controller, in turn, needs to respond by inflating the airbags. This application comprises several smaller

JEDEC Parameter	No. of Clock Cycles	Parameter Description
tRCD	9	Activate to Read/Write Delay
tRL	9	Read to Start of Data
tWL	7	Write to Start of Data
tBUS	4	Time to transfer Data through the bus
tRP	9	Precharge to Activate Delay
tWR	10	Write Recovery Time
tRTP	5	Read to Precharge Delay
tRAS	24	Activate to Precharge Delay
tRC	33	Delay between two Activate commands of the same bank
tRRD	4	Delay between two Activate commands of the different bank
tFAW	20	Four Bank Activation Window
tRTW	7	Read to Write Delay
tWTR	5	Write to Read Delay
tRTR	2	Delay to switch between two Ranks
tRFC	160ns	Total time to refresh a DDR Row
tREFI	7.8us	Time between two DDR Refreshes

Table 2.3: Timing Constraints (source:JEDEC standards)

tasks that perform specific activities like sensing, control, and actuation. One of the key components in the response time of an application is the time the software program takes to execute on the allocated hardware and perform various tasks. In our example, this software program may for instance monitor the values from the crash sensors, execute a control loop to determine whether a crash has occurred, and eventually send control actions to the actuators. Further, since this application is safety-critical in nature, it is essential to have knowledge on the WCET of these tasks at design time so that the designer can allocate sufficient resources to the application. If the WCET of tasks can be accurately estimated at design time, then the worst-case response time of the safety-critical application can also be estimated.

System developers rely on either executing the task and measuring the execution time under various operating contexts or performing static code and hardware analysis, to obtain a WCET estimate at design time. Static analysis typically results in estimates that are pessimistic when compared to the actual WCET, mainly because of the complex nature of modern hardware and software. On the other hand, measurement-based approaches may have a scenario wherein the maximal observed execution time is less than the actual WCET. This is because in a real world scenario it is practically impossible to verify the entire input space and all possible hardware states for every task. Different inputs may cause a task to execute differently and hence may alter its execution time. Also, the hardware states of caches, processor pipelines and interrupts may also affect the execution time of the task.

Static code and hardware analysis is a classical approach for estimating the WCET of a task that does not require the task to be executed on the hardware. Instead, it takes the task as input along with annotations from the user, finds all possible control flow paths, combines control flow and an abstract hardware model and estimates an upper bound to the WCET of the task. A typical example of this

method is static cache analysis, wherein the process tries to predict the contents of cache at every instant of task execution and arrives at a particular solution for determining the statistics of cache hits and misses. This analysis is usually hard in the sense that it requires accurate knowledge of all possible task inputs that can result in a different number of cache hits and misses.

Static analysis depends on an abstract hardware architecture model to estimate hardware states during task execution. The most common hardware model for multi-core static analysis is as shown in Figure 2.1. Each core has at least one level of private cache; there is at least one level of shared cache and finally, a shared bus connects the processor to the DDR memory. Further, these analyses also make some assumptions on every component in the architecture. For instance, the shared bus can be assumed to use Time Division Multiple Access (TDMA) arbitration scheme or caches can be assumed to use LRU replacement policy. These assumptions are mainly for the purpose of simplifying the analysis, although some of them may not be very realistic. For instance, most COTS multi-cores do not use LRU cache replacement policy because it is very expensive (time and memory) to implement. When all the aspects of the analysis are completed the values are input to an Integer Linear Program (ILP) solver. This last stage of processing is called Implicit Path Enumeration Technique (IPET) analysis in which the results of all the previous stages are combined together with information about infeasible paths in the task to arrive at a WCET estimate.

On the other hand, measurement-based approaches execute tasks on the given hardware for a given input set and determine the distribution of the observed execution times. The most common approach to obtaining measurements is to execute the task repeatedly with many possible input scenarios and instrument the code to collect timestamps for determining the execution time. Measuring execution times is also done externally by using hardware debuggers and tracer units that can collect the timestamps with minimal intrusion on the executing task. Measurements alone are not useful for safety-critical applications that require timing guarantees, as they do not consider the complete operating context such as all possible hardware states or all possible input values. Therefore they are combined with some code analysis techniques such as value, loop bound, and path analyses to improve the confidence in the resulting WCET estimates. The main advantage of measurement-based methods over the static analysis methods is that they do not need a hardware model for analysis. This is a significant advantage because modern hardware architectures are too complex to be amenable to precise modeling. Another advantage of measurement-based methods is that the resulting WCET estimates are less pessimistic when compared to the estimates obtained by static analysis methods. However, because measurement-based methods are not guaranteed to consider the complete operating context of the task, the resulting estimates may be incorrect. Few examples of tools that implement measurement-based methods are SymTA/P [13],

Rapitime [14], etc. The Rapitime tool, for instance, provides several features such as WCET estimation, identification of program code that is on the worst-case path, generation of execution time profiles to illustrate variability in code executions, and path analysis to provide an exact model of the software code structure. The subsequent sections will look into the challenges that exist in this WCET estimation with reference to the multi-core processors.

2.1.3 Challenges in Estimating WCET on multi-cores

The challenges in estimating WCET on multi-cores have been discussed in several studies ([15], [16], [17], [18], [19], [20], [21], [22] and [23]). These challenges are summarized below:

1. Shared caches and their controllers,
2. Shared interconnect,
3. Shared memory controller,
4. Power saving strategies,
5. System interrupts,
6. TLB misses, and
7. Hardware prefetching.

It is to be noted that the challenges 4, 5, 6 and 7 are not specific to multi-cores but also applicable to single core processors as well. These challenges are discussed briefly in the following paragraphs.

Shared caches and their controllers All cores share at least one level of cache in a COTS multi-core as shown in Figure 2.2, allowing efficient sharing of data when inter-dependent tasks are running on different cores. Consider an example of two tasks executing in parallel on two different cores of P4080 (co-running tasks). There can exist a scenario when both contend for the same line in the shared cache, even though they share no instructions or data. This is because multiple locations in memory can map to a single cache line. In such a scenario, WCET can depend on how frequently the task accesses the cache, as every access can potentially evict useful data of the other task. If data is removed from the cache, it needs to be fetched again from the DDR memory which increases the task's execution time. Further, even if the two tasks do not access the same cache line, the cache controller which is shared

between the cores can delay one of the tasks while serving the other. This also increases the task's execution time. Thus interference arising from co-running tasks due to shared caches has the potential to impact a task's WCET on a multi-core platform.

A simple strategy to overcome this interference is to partition the cache such that no two cores can access the same cache line. This approach is effective only when independent tasks are running on the different cores. But as more applications are parallelized to exploit the multi-cores better, sharing data through caches will significantly reduce the response times as the number of memory accesses decrease. Hence shared caches are important to exploit parallelism offered by multi-cores although they affect predictability. Also, the contention at the shared cache controller cannot be solved by partitioning. The work [24] provides a survey on cache management techniques for real-time systems.

Shared Interconnect When there are multiple cores with a shared memory unit, there must be a common communication link that connects all these cores to the memory. This is the system interconnect (CCF in Figure 2.2). Although most multi-cores have high bandwidth for the shared interconnect, there exists contention as the access to this interconnect is not only dependent on the cores but also on other architectural features such as the caches and the traffic due to coherency protocol. Also, bounding this contention may be hard as the chip manufacturers do not provide extensive details on the architecture of the interconnect or the arbitration mechanism it uses.

Analysis of the shared interconnect depends on accurate knowledge of the interconnect architecture. When there is a lack of information, WCET estimation is performed using assumptions that may or may not be realistic. For instance, the arbitration scheme for the shared interconnect is assumed to be Time Division Multiple Access (TDMA) in many methods.

When a system maintains both private and shared memory components such as caches, any change of data at any location needs to be updated in every other memory resource where a copy of this data is held. The coherency mechanism keeps all these copies in harmony with each other by continuously updating every change. Multiple cores updating shared data in their respective local caches can cause heavy coherency traffic on the interconnect, which in turn can cause a delay in the execution of applications.

Shared Memory Controller The DDR memory controller (two 64-bit DDR3 memory controllers shown in Figure 2.2) is essential for controlling the interaction of the cores with the DDR memory. The number of memory controllers is usually lesser than the number of cores, primarily because of packaging constraints that is limited by the number of pins available [25]. For example, the P4080DS

is an octa-core processor with two DDR memory controllers shared between the cores. This uneven distribution of the number of cores to the controller itself is a reason for contention. Whenever there are multiple cores simultaneously trying to access the memory, there may be contention. Also the arbitration scheme employed by COTS controllers is mostly unknown, which increases the difficulty in the analysis of such controllers for WCET estimation.

Power Saving Strategies Most modern processors have reached operating speeds that are almost saturated and are now primarily focused on reducing the power consumption. Hence a separate system is dedicated to implementing power saving strategies for the processor. One of the simplest strategies used to save power is to change the frequency of the processor. Reducing the frequency naturally reduces the operating voltage and hence the total power consumed. Such optimization techniques may cause unpredictability in the execution of tasks on the cores, and therefore can impact WCET estimation.

System Interrupts A system might have interrupts that cannot be prevented and these interrupts can impact task executions. A simple scenario is when a processor heats up, it may either be turned off or taken to a system management mode that would bring it back to normal execution. This consumes processor cycles and hence must also be taken into account when estimating the WCET.

TLB misses Virtual memory is an important aspect of most multi-cores. A dedicated Memory Management Unit (MMU) is assigned to translate these virtual memory addresses to physical ones. The cores refer to virtual memory addresses which in turn are translated to physical addresses by the MMU. MMU has a cache called Translation Look-aside Buffer (TLB) that stores the recent translations. The flip side of using this buffer is that if there is a TLB miss, the translation request needs to be sent to the DDR memory and then mapped back. This would incur additional cycles in the execution of a task. Since this cache is shared, it becomes hard to predict the eviction of translations from different cores. These evictions cause unpredictability and therefore also impact WCET.

Hardware Prefetching Prefetching allows the processor to fetch blocks of instructions and data before the task actually requests for them. This is generally due to speculative techniques such as branch prediction and cache block prefetching. Speculation may on an average improve the execution time of tasks if it manages to predict the future correctly on a regular basis. But it also negatively affects WCET of tasks, because for scenarios when the speculation is wrong, the resulting execution time also includes the misprediction penalty.

Chapter 3

Related Work

3.1 Introduction

This chapter provides an extensive background on the different contributions in the literature to obtain predictable WCET estimation on multi-cores and discuss their relation with the contributions in this dissertation. In order to understand the different contributions in the literature, a classification methodology is first provided to categorize the existing work. Further, the different contributions under each category along with their relation with this dissertation is discussed.

Classification of the existing work: Based on the challenges explained in Chapter 2, a number of different solutions have been proposed in the literature to alleviate them. The existing research towards obtaining a WCET estimate for tasks on multi-cores are categorized (described in Figure 3.1) as follows:

1. COTS hardware and unmodified middleware,
2. COTS hardware and modified middleware, and
3. Custom hardware and unmodified middleware,

The first category, *COTS hardware and unmodified middleware* focuses on research that develops WCET estimation techniques for COTS multi-cores. This category of research includes studies on cache analysis, interconnect analysis, pipeline analysis, etc. Although these methods do not require any modifications to the hardware or middleware components, they make simplifying assumptions on the architecture that may or may not be valid for COTS platforms. The primary studies of interest that belong to this category are those that provide specific solutions to overcome the challenges in

analyzing multi-cores. This category of research is further subdivided into four subcategories based on the aspect of hardware that the work analyzes. These subcategories are cache analysis, cache locking and partitioning, DDR controller analysis and shared interconnect analysis.

The second category of research *COTS hardware and modified middleware* does not require any modifications to the hardware (i.e., considers COTS multi-cores), and instead focuses on modifying the middleware components to obtain predictable WCET estimates. An example of work in this category is the implementation of fork and join operation within the OS where a task is segmented into smaller subtasks and executed individually and then joined to obtain predictable execution pattern.

The final category, *custom hardware and unmodified middleware* focuses on building custom components of the multi-core architecture on Field Programmable Gate Arrays (FPGAs), without making any changes to the middleware components. Specifically, this category includes research on designing predictable caches, memory controllers and system interconnects on FPGAs. Projects such as MERASA [26], CoMPSOC [27] and parMERASA [28] fall into this category. The primary research problem this category addresses is the modifications required at the hardware level to estimate WCET with minimum pessimism, while assuming that the middleware components remain unchanged. Note that although these studies use FPGAs to illustrate their hardware designs, the final designs could always be transformed into ASICs. Within this category, the contributions are classified into four subcategories based on the hardware component that is designed. These subcategories are caches, scratchpads, memory controllers, and system interconnects.

For each category, the different contributions and their difference to the contributions of this thesis is explained.

3.2 Analysis of COTS hardware and unmodified middleware

This section discusses the research that performs static analysis on COTS multi-cores and obtains WCET estimates for the tasks. There are four sub-categories which fall within this section. They are cache analysis, interconnect analysis, DDR controller analysis and cache locking/partitioning mechanisms.

Figure 3.1: Work towards predictable WCET

3.2.1 Cache Analysis

This subsection presents research that addresses the unpredictability in the access to shared caches by various analysis methods (box 5 in Figure 3.1). A timing analysis technique for a COTS multi-core with at least one level of shared cache was proposed by Liang *et al.* [29]. The source of unpredictability addressed by this analysis is inter-core cache evictions described in Section 2.1.3 of Chapter 2. The analysis determines the lifetime of all the tasks that are executing simultaneously and finds the potential interference in the access to the shared cache. Once this estimate is obtained, cache accesses are classified either as hits, misses, first access miss or unclassified. Using this information along with static analysis approaches WCET of tasks are obtained.

Another study proposes a method to mitigate the effect of shared cache interferences [30], [31]. The major contribution of this work is that it offers a method to reduce the pessimism in the estimated WCET while analyzing the inter-core interference. This is achieved by identifying code blocks that are used only once during the execution and avoid caching them. This work assumes that every core has at least one level private cache and all cores share another level of cache. Also, the caches use LRU replacement policy.

Discussions: The static analysis work on caches try to solve the problem of shared caches by analyzing them off-line and without making any changes to the multi-core architecture. The classical cache

analysis is the concrete analysis that provides information about every cache state [32]. The drawback of using this approach was that the number of cache states increases exponentially. To mitigate this problem, the concept of Abstract Interpretation (AI) was introduced, which reduces the number of cache states. With the advent of multi-cores, shared caches were introduced in the architecture which caused inter-core data eviction. This was analyzed using inter-core cache analysis discussed in this section. Both studies discussed in this section, Liang *et al.* [29] and Hardy *et al.* [31], use AI to perform their single-core analysis. The difference between them is how inter-core analysis is performed. The study by Liang *et al.* [29] uses task graph and an iterative mechanism to estimate the cache misses that occur in the shared cache due to inter-core cache conflicts. On the other hand, the study by Hardy *et al.* [31] uses a conflict bound to identify the blocks of program that interfere with each other. Also, the study by Hardy *et al.* [31] improves upon the existing analysis technique by identifying blocks of program that are used only once (single used blocks), and removing them from cache analysis thereby reducing the pessimism in WCET estimation. It is also essential to note that the evaluation by Hardy *et al.* [31] ignores task lifetimes and scheduling, whereas Liang *et al.* [29] uses this information in the analysis.

One of the major drawbacks of both these analyses is that they rely on assumptions such as LRU cache replacement policy and non-preemptive processing. These unrealistic assumptions pose a serious question on the applicability of these analyses for COTS processors. On the contrary, robust analysis techniques without these assumptions is an open problem in this domain of research. Since all the work is predicated on modifications to the hardware and specifically on DDR memory controllers, the cache analysis does not directly relate to any of the contributions in this thesis.

3.2.2 Interconnect Analysis

This subsection presents research that addresses the unpredictability in access to the shared interconnect (box 8 in Figure 3.1) by analysis methods. A technique to analyze the delay in access to the shared interconnect was proposed by [33]. The assumed system model is a multi-core architecture with every core having one level private cache and an in-order pipeline. The cores connect to a shared interconnect which performs TDMA arbitration. This interconnect is connected to another level of cache which is also shared and finally to the DDR memory via a controller. The cache in this analysis is exclusive and uses LRU replacement policy. To analyze the shared interconnect, this study assumes a TDMA scheme where a fixed length time slot is assigned to every core. Thus, each core can obtain access to the interconnect only during its slot. Otherwise, it needs to wait to obtain a slot. Depending on the time when a task requests access to the interconnect, the amount of time that this request would have to wait before

getting access to the interconnect is determined. This approach helps to bound the interconnect delay and is used to estimate the WCET of the task. The analysis is embedded into the CHRONOS WCET analysis framework [34].

Another study proposes an approach to estimate WCET by taking into account different access times due to the concurrent usage of shared resources in multi-cores [35]. This work is an offline analysis which is supported by monitoring technology that enforces guarantees on shared resource usage at run-time. The timing analysis is performed in two steps where the first is to quantify delay within a core and the second is to quantify delay due to sharing of resources. Every shared resource is abstracted using its bandwidth, and every requesting task is designated a certain portion of it. Using this allocated resource share, the technique ensures that none of its resources are over-utilized. Run-time monitoring is used continuously to update the resource usage of a task. Monitoring identifies if a particular resource is over-utilized by a task and provides a trigger for suspending that task.

Another study proposes a model checking based approach for timing analysis of multi-cores [36]. This work tries to solve the unpredictability in multi-cores by modeling the cache and shared bus using Timed Automata (TA). Tasks are bounded using local cache analysis and assuming a predictable arbitration on the shared interconnect.

The study [17] computes a bound on the maximum memory access requests that a core can generate and also proposes a methodology to find the delay that a task incurs when multiple tasks execute on a multi-core. This improves the work proposed in [37]

Discussions: Static analysis techniques for the system interconnect try to address the problem of unpredictability arising due to inter-core interference on the shared interconnect (described under Section 2.1.3). The studies on shared interconnect presented in this section view the problem differently, wherein one of them assumes TDMA arbitration for access to the shared interconnect, while another computes the total interference on the interconnect and monitors them at run-time, another work develops a timed automata for the functioning of the shared interconnect and the last contribution bounds the number of memory accesses to the interconnect. All these works primarily assume the arbitration to be TDMA or round-robin. This assumption may not be valid for a COTS hardware since most of the processor manufacturers do not disclose their arbitration schemes. Also, none of the analysis techniques presented in this section include the effect of shared caches on the interconnect, or consider the effect of running a coherency protocol. As mentioned previously, all the work described in this thesis is predicated on modifications to the hardware and the contributions under Section 3.2.1 does not directly relate to any of the contributions in this thesis.

3.2.3 DDR controller Analysis

This subsection presents research that addresses the unpredictability in access to the DDR memory by analysis methods (box 7 in Figure 3.1). The work in [38] models the DDR memory resource and obtains a bound for the delay experienced due to tasks that are executing simultaneously on a multi-core. This work has three main contributions. First is considering a model which takes into account the timing details of DDR memory in accordance with JEDEC standards. Second is to bound the memory access interference of a task by considering its accesses and the interfering accesses during its execution. Finally, it also evaluates the consequence of using private against shared DDR banks.

The contribution by [39] improves the basic DDR analysis performed in [40] by considering a realistic model of interference of tasks executing simultaneously on a multi-core. This work also follows the two approaches as mentioned in [38]. One is to bound the delay based on the memory demand of the task under analysis and the second is to bound the delay due to the interference from the other tasks. The key factors which form the basis of this implementation are that the number of contending memory accesses from other cores are bounded and the interference of DDR commands at command scheduling level are considered. These factors help to reduce the pessimism in the memory interference analysis to provide a tighter bound.

Discussions This section focuses on the unpredictability in the access to the shared DDR memory through its controller. The studies presented ([38] and [39]) consider the effect of interference when a task executes with respect to the memory controller. However, these analyses are still pessimistic in the worst-case (as presented in [39]). Hence they require more architectural knowledge/assumptions that can reduce the pessimism. Since, the contributions in this thesis are focused on modifications to the hardware and bank allocation policies in DDR, these work do not directly relate with this dissertation.

3.2.4 Cache Locking and Partitioning

Locking and partitioning of caches are two hardware techniques that facilitate predictable access to the shared cache on a COTS multi-core platform. When a core locks a line in the cache, the data in the locked line cannot be evicted by tasks executing on the other cores. Orthogonally, caches can also be partitioned in a way that the different partitions are mapped to the distinct cores. This subsection presents three studies that focus on achieving predictability on the shared cache through locking or partitioning (box 6 in Figure 3.1). The work by Shekhar *et al.* [15] proposes to use cache locking and

locked cache migration to schedule a set of independent tasks using semi-partitioned scheduling strategy (strategy discussed in [41]). Given a set of tasks and cores, this work statically partitions as many tasks as possible on to the different cores. Those tasks that are not statically assigned to a core can migrate from one core to another among a predetermined set of cores. Within each core the task that has the earliest deadline is scheduled first for execution. The tasks within a single core can statically select and lock memory lines within the private cache of that core. While migrating, memory lines of that particular task are first unlocked and then moved to the target core and then locked again within the target core's private cache.

Another work explores locking and partitioning techniques for predictability in shared caches [42]. It uses different combinations of cache partitioning and locking to analyze the impact on the WCET of an application. The various methods evaluated are static and dynamic locking techniques with and without partitions in the shared cache.

Discussions Cache locking and partitioning try to address the challenge of predictably accessing the shared cache (described under Section 2.1.3). These are hardware features that can be used at the software level to perform cache accesses. Two central questions that arise while using these techniques are “What data to lock and for how long?” and “How much of the cache is to be partitioned?”. The studies discussed in this section try to address these issues by presenting various techniques for locking and partitioning. All these techniques, however, lock/partition the entire shared cache. But multi-core as an architecture is suited to share data across the cores. None of these techniques address this issue of efficient sharing of data between the cores when the cache is partitioned/locked. Also, since the focus of the thesis is on developing custom hardware solution along with DDR bank allocation policies for COTS multi-cores, these contributions are unrelated to this dissertation.

3.3 Unmodified COTS hardware and modified middleware

This section presents research that makes no changes to a multi-core hardware but modifies the middleware to obtain predictable execution of tasks in multi-cores (box 9 in Figure 3.1).

The work by Alhammad *et al.* [43] proposes a method to execute tasks such that the accesses to memory can be completely independent of the policies employed by the hardware. This is achieved using an algorithm that maps and schedules tasks on the multi-core using fork-join model. In this work, every sequential task is split into multiple segments and every segment is processed in two phases. In

the first phase, data that is required for task execution is brought to that particular core's private cache. This step ensures that the execution does not suffer any cache misses and hence would also not access DDR memory. In the second phase, the task is executed.

An OS-level technique to allocate shared cache partitions to different tasks was proposed by Kim *et al.* [44]. To predictably share caches, this work uses page coloring mechanism. Rows in the DDR memory are termed as pages. Page coloring algorithm allocates different colors to each page with every color physically mapped to different locations in the DDR memory.

Another study proposes the use of page coloring mechanism and real-time resource locking protocols to handle inter-task interferences on the shared cache [45]. The challenge with coloring mechanism is to find an optimal strategy to assign colors to tasks. The idea of this work is that it utilizes page colors as resources that can be shared, and the access to these resources could occur through arbitration using any real-time locking protocol.

Another work uses a virtual memory model that allocates memory pages of every task to only specific banks [46]. Operating systems lack details on the mapping of memory accesses to DDR banks. Most memory accesses are spread across several banks of memory. Consider a simple scenario when there are 2 cores and 2 banks. The best case memory access (the least time that would be required to complete a memory access) would occur when each core accesses only one bank or when only one of the cores requires memory accesses to both the banks. In these cases, there are no inter-core conflicts in the access to memory. But most memory accesses are such that both cores could potentially access data in both the banks. This causes unpredictability in the memory access times. This work presents a DDR bank allocation scheme to address this issue.

Another study implements a memory bandwidth reservation system that allocates, recovers and maximizes the utilization of the memory bandwidth [47]. This work provides an OS-level approach which not only increases the performance of each core but also provides predictable memory access. It works by regulating every core's request rate to the memory (the rate at which every core can generate request to the DDR memory). The main principle of regulating this bandwidth is that the total memory request rate is always maintained below the DDR's total capacity.

Another OS-level study develops a framework to obtain memory patterns of a task, and also a cache maintenance method for the most frequently accessed memory locations [48]. Shared caches are always a source of unpredictability due to the interference in the access to it by multiple cores. This study uses cache coloring and locking to provide predictable access to the memory.

A recent study combines multiple OS-level techniques to manage interference in shared resources [49].

This work combines [46], [47] and [48] techniques described before to address the challenges in shared hardware resources on COTS multi-cores.

Discussions This category of work (Unmodified COTS hardware and modified middleware) tries to address the challenges of shared hardware resources (specifically caches and memory controllers) described under Section 2.1.3 by making modifications to the middleware. The first contribution is a predictable execution model that tries to schedule every resource access and concurrent threads in two phases, prefetching and computing. This method of execution would suffer in throughput when there are a large number of cores that are executing real-time tasks. For predictable cache accesses, there are two contributions, both of which utilize the concept of page coloring mechanism. The study by Ward *et al.* [45] proposes a more dynamic method of scheduling the tasks to access the cache, whereas the work by Mancuso *et al.* [48] statically analyzes the tasks to obtain all possible memory requests. Finally, the work by Yun *et al.* [46] proposes PALLOC that provides predictable DDR accesses. Again the drawback of this method is the use of list data structure that can cause a significant overhead in real-time. Hence, this category of work does not completely alleviate the challenges pertaining to predictable execution of applications on a multi-core as it does not address all the challenges discussed in Section 2.1. Also, this category of work does not directly relate to any of the contributions as these contributions modify the middleware and the work in this thesis focuses on hardware modification and DDR memory bank allocation analysis.

3.4 Custom hardware and unmodified middleware

This section discusses the various solutions presented for designing predictable multi-core hardware (box 1 in Figure 3.1) and discuss the relation to the contributions in this thesis. There are four sub-categories in this section which are cache designs, scratchpad designs, DDR controller designs and shared interconnect designs.

3.4.1 Cache Designs

The source of unpredictability in shared caches on multi-cores is its replacement policy. This work introduces the Random Placement (RP) policy along with the Random Replacement (RR) to estimate the WCET of a task which is associated with an occurrence probability [50]. The advantage of this policy is that every memory access is associated with a probability of hit or miss independent of the

history of memory accesses. This probability depends on the total number of cache lines and also its associativity. By randomizing the placement of data in the cache, this approach computes the probability of hits and misses for every memory access in the task and obtains a probabilistic WCET estimate [50].

Another hardware cache implementation designed by [51] improves the average-case performance while providing predictable access to caches. This is achieved by counting the number of cache hits and misses along with a cache preloading algorithm. The cache preloading algorithm loads the cache with instructions to avoid the initial cold misses. The counters record at run-time the statistics of hits and misses and provide access to the cache only when the difference between hit and miss counters is positive. Whenever the difference is not a positive value, the memory accesses are taken to the DDR while still loading the cache until the difference becomes positive.

The architecture uses existing cache design with two counters to monitor the statistics of cache hits and misses. To avoid cold misses the cache blocks are preloaded with data by exploring all possible paths in the task. Once a task starts to execute the counters are incremented based on hits and misses that occur. When the value of the miss counter is greater than the hit counter, the cache is disabled and every access is directed to DDR memory. Simultaneously, the cache is loaded with the data that is accessed. The hardware remains in this state until the difference between hits and misses becomes positive.

Discussions This section discussed hardware solutions that address the unpredictability in the access to shared caches. Two studies on building cache architectures on FPGAs have been reviewed. These contributions are orthogonal to the contribution in this dissertation as these ideas may be implemented as one of the components in the hybrid SoC multi-core architecture described in Section 3.1 by making them AXI protocol compliant. One of them presents a probabilistic approach [50], whereas the other study proposes a cache architecture with hit and miss counters [51]. The drawback of using RR and RP policies in RTA have been identified by Reineke [52]. It shows that the probability of hits that are computed for the analysis are not independent. This means that the convolution of different execution time values obtained is not possible with randomized caches and hence is not favorable to be used in RTA. Although these studies contribute to building a predictable cache architecture, their implementations are limited to only a few aspects of the cache architecture. From the challenges described in Section 2.1.3, it can be seen that the unpredictability from the shared cache controller and coherency mechanism have not been addressed by any of these studies. To obtain predictable accesses to shared caches, these problems must be resolved.

3.4.2 ScratchPad Memories (SPM) Designs

To avoid the unpredictability arising from using shared caches, researchers have proposed to have local faster memory termed as scratchpads. These are lightweight, low-power and high-speed memory units that reside close to the processor. There are two methods by which functions are loaded into the scratchpads. They are static and dynamic. This section discusses the various contributions towards predictable data loading into scratchpads (box 3 in Figure 3.1).

Static SPMs have fixed data allocation, i.e., their contents cannot be changed at run-time. The work by Suhendra *et al.* [53] proposes three techniques to allocate data statically to the SPM. First is to formulate an ILP which considers all possible program paths and obtains the data to be loaded. Second is to use branch-and-bound search algorithm to find the best possible allocation by eliminating those paths which are infeasible in a task execution (this method is expensive due to the exhaustive search involved as the number of tasks increase). Finally, the third method is to reduce the complexity by using a heuristic where infeasible paths in the program are removed and then from the remaining paths, the one which leads to WCET (technique is discussed in [54]) is considered and data is allocated from it. This step is repeated till the SPM is fully filled. Another study by [55] proposes three different static loading scenarios. First scenario is where every task has a partitioned memory space in the SPM. Second scenario is where a task that is executing uses the entire SPM. Finally, the third scenario is where hard real-time tasks can use the space of non-real-time tasks. An ILP is formulated for each case which decides both on partitioning the SPM and also loading data into it.

Dynamic SPMs have data allocation that is managed at run-time. A Dynamic Instruction SPM for multi-cores was proposed by [56]. The primary idea of this work is that whenever there is a function call, the pipeline is stalled and all the instructions of that function are loaded into the scratchpad before execution. Another study targets a dynamic SPM that uses a memory management unit to manage the loading of data into the dynamic partitions in the SPM [57]. The OS partitions the SPM into three segments. The first segment is of fixed size and is used to hold the data related to the OS itself. The second and third partition sizes can be dynamically varied by the OS based on the size of the function being loaded into the SPM. The work by [58] explores different configurations of the SPM with regards to using it with multi-core architectures. In particular, it compares single against two-level SPM, unified against separate, and dynamic against static loading and obtains the merits and demerits of each scenario. Conclusions drawn from this work are that multi-level SPMs reduce WCET in most cases, separate SPMs provide better performance than unified and finally, dynamic loading helps in using the SPMs more efficiently to manage run-time demands. The study by [59] proposes two dynamic loading

techniques for scratchpads in multi-core architectures where the entire execution depends on contents in the SPM (DMA is used to load the SPM with functions from DDR whenever they are requested). The first technique is to find the optimal method to map functions to different portions of the SPM. An ILP is formulated which iteratively determines the best possible function mapping using a control flow graph of the program to be executed. The disadvantage of this method is the exponential complexity as the number of functions and size of SPM increases. A simpler heuristic (reduced complexity) has been proposed where two different strategies are used to obtain the mapping. First, is by mapping different functions to separate spaces on the SPM and then merging them to fit inside the SPM. Second, is to map only one function to the SPM and then follow an iterative process to partition the SPM space to different functions.

Another solution presented to address the unpredictability in shared caches is to use tickpad memories [60]. Tickpads are identical to caches and scratchpads as they provide quick access to memory, but are only used with synchronous programming languages such as Esterel [61] or Pret-C [62]. A synchronous language program is analyzed and a graph is generated which provides information on concurrent control flow and context switching points [60]. Using this graph, data allocation decision to the tickpad is obtained. With this information, the memory access time is computed and the execution time of the application in the worst-case is estimated.

3.4.3 Comparison of different Memory Architectures

The inequality between the clock speed of a processor and the time to access DDR memory requires the presence of a memory unit that is not just faster, but also closer to the processor. The initial solution was to have a cache architecture that is on-chip so that the delay in accessing data/instructions is reduced. Caches are designed having multiple levels with at least one of them shared among the cores. This required coherency protocol to maintain the consistency of data in the caches, that lead to an increased processing overhead. Also, sharing of caches created the possibility of useful data of one core being evicted by another. To avoid these complexities SPMs were introduced. Data allocation and its replacement on SPMs are software guided and decisions are taken at compile-time. This improved their average-case and worst-case performance when compared to caches. Unfortunately, SPMs do not allow sharing of data between the cores, which is a key requirement to exploit the parallelism of multi-core architectures in many applications. In the absence of efficient sharing mechanism, all data sharing must happen through DDR memory, which can be very time-consuming. Tickpads offer a trade-off between caches and SPMs by providing dynamic loading capabilities that are statically controlled. This allowed

them to perform better than SPMs, but they still do not provide support for efficient sharing of data between the cores.

Thus, these contributions have tried to address the problem of having a faster memory closer to the processor. These contributions are orthogonal to the work in this dissertation as they can be implemented and tested as a part of the architecture on the FPGA side and integrated with the COTS multi-core if these contributions can be made AXI protocol compliant. Also, the open problem that has not been addressed by these contributions is to provide solutions for predictably sharing the data between the cores outside of the DDR.

3.5 Memory Controllers

This section discusses the various contributions towards designing predictable memory controllers (box 4 in Figure 3.1) and their relation with the contribution in this dissertation. Note that JEDEC provides a set of operational timing standards that device manufacturers need to adhere while manufacturing DDRs. Hence, the controllers designed for these DDRs must also comply to these timing standards [12]. The generic structure of a DDR memory and a background on JEDEC standards based on which the memory controllers are designed is provided in the background Section 2.1.1.

Most of the existing contributions towards designing predictable memory controllers have been focused on several aspects of the controller which are as follows. The memory controller is viewed in two parts, the front-end and the back-end. The front-end generally is to handle the memory requests from different cores and arbitrate between them. The back-end focuses on scheduling the commands that the DDR requires to perform read, write operations. Hence, most contributions have been improving one or both of these aspects. Every contribution provides a combination of their arbitration policy along with a predictable mechanism by which the DDR memory commands are handled. The Table 3.1 summarizes a list of significant contributions towards obtaining a predictable memory controller. It can be seen from the Table 3.1, the main aspects that all contributions focus are the page policy, arbitration policy, command scheduling and memory mapping. The following paragraph summarizes these contributions and explain the key aspects of each of these controllers.

The work by [72] proposes a real-time controller implementation which performs round-robin arbitration between the different cores trying to access the DDR memory. This prevents the unpredictability in memory access due to inter-core interference. Interference delay is upper bounded using an exhaustive analysis based on the timing constraints provided by JEDEC standards. As an additional feature, to

Table 3.1: Real-time Memory controllers

Controller	Page policy	Arbitration Policy	Rank Scheduling	Bank Mapping (Interleaving vs Private)	Critical/ Mixed critical
Predator: a predictable SDRAM memory controller [63]	Close row	CCSP	-	Interleaved	Critical
PRET DRAM controller [64]	Close row	TDM	-	Private bank	Critical
Dramsim: a memory system simulator [65]	Open row	Round robin	Rank hopping	Interleaved	Non-critical
Architecture and analysis of a dynamically-scheduled real-time memory controller [5]	Close row	Round robin	-	Interleaved	Mixed critical
A reconfigurable real-time SDRAM controller [66]	Conservative open row	TDM	-	Interleaved	Mixed critical
A mixed critical memory controller using bank privatization and fixed priority scheduling [67]	Close row	Fixed priority	Rank switching	Private bank	Mixed critical
Architecture and optimal configuration of a real-time multi-channel memory controller [68]	Close row	CCSP	-	Interleaved	Mixed critical
A rank-switching, open-row DRAM controller [69]	Open row	Round robin	Rank switching	Private	Mixed critical
A predictable and command-level priority-based DRAM controller [70]	Close/Open	Fixed Priority	-	Private	Mixed critical
An Analyzable memory controller [71]	Close row	Round robin	-	Interleaved bank	Critical

avoid interference between the different tasks executing on the same core, this implementation provides one buffer per task.

A reconfigurable real-time DDR controller was proposed by Goossens *et al.* [66]. Since the architecture is reconfigurable, a trade-off has been obtained between the provided bandwidth, response time for a memory request and power consumption. The controller uses TDMA arbitration (this module is also reconfigurable at run-time). This protocol allows changing the allocation of TDMA slots at run-time without losing out on predictability. The proposed architecture is predictable due to the implementation of two important features. First is the grouping of memory requests into predefined memory command patterns to eliminate interference between memory requests. Second is the use of TDMA arbiter for predictably switching between the different cores.

Another controller design that not only focuses on predictable memory accesses but also improves the worst-case latencies was proposed by Reineke *et al.* [64]. The key aspect of this implementation is that it considers the DDR memory as multiple devices that may be shared among several clients individually. It exploits the parallelism in the DDR memory by segregating the physical address space of the device in accordance with ranks and banks. Refreshing the DDR is performed by row accesses rather than actually sending a refresh command. Closed-page policy is followed wherein the accessed row is closed (precharged) before performing the next access.

A predictable controller design by Krishnapillai *et al.* [69] focuses on using rank switching and partitioning mechanism. Rank switching increases the utilization of the DDR and avoids the problem of write to read transition latency which causes a significant delay in the memory accesses. Rank partition-

ing provides isolation by assigning banks in the ranks to specific cores thereby providing independent accesses to the banks.

Another predictable controller design (PREDATOR), which combines the properties of both static and dynamic controllers was proposed by Akesson *et al.* [63]. The unique feature of this controller is that instead of sending independent commands to the DDR, this controller groups and sequences several DDR commands together. These are read, write and refresh groups. Predictability is obtained by scheduling these groups of commands with a known arbitration scheme at run-time.

Comparison of Memory Controller Designs The unpredictability in memory controllers described in Section 2.1.3 has been addressed by the studies presented in the Subsection 3.5. The Table 3.1 summarizes the differences between the different contributions to predictable memory controllers. One of the common features in all these memory controller designs is the use of TDMA or round-robin arbitration among the real-time tasks and cores. A few of these works ([66] and [69]) rely on either generating composable patterns (patterns of DDR commands) or analyzing the task to obtain the nature of memory accesses (open-row or closed-row) that can increase or decrease the overhead in the memory controller operation.

Considering the state-of-art multi-cores that are clocked at several GigaBytes per second, we not only need real-time guarantees with respect to timing behavior but also increased performance to satisfy the growing demands in processing. Further, most of these studies are simulations and hence there is a need for implemented designs. Finally, the designed memory controller should be scalable with respect to the number of cores, i.e., as the number of cores increase, the memory controller should be able to provide similar performance and real-time guarantees for the system. Another aspect to fulfill is the need for composable architectures which can reduce the pessimism at the hardware level in estimating the WCET. Composable architectures are those where the execution of a task remains unchanged irrespective of the presence of other tasks (inter-core interference problem mitigated). This essentially means that when there are shared hardware resources, every shared resource must be able to guarantee similar service for all tasks in the system such that every task can be analyzed independently [73] [27].

The difference between these controllers and the predictable memory controller proposed in the architecture is that, the proposed architecture has a combination of features that not only make the memory predictable but also bounded for latency based on actual implementation. These are posted CAS operation with closed row policy and interleaved memory mapping technique with a key differentiator being the controller is AXI protocol compliant which implies that while designing for predictability,

the actual implementation aspect of the controller has been taken into account. Also, this controller has been physically interfaced with a COTS multi-core and to a DDR memory. The existing controllers are orthogonal to the proposed architecture as these controllers can be integrated and tested for their predictability by implementing them on the FPGA if they can be made AXI protocol compliant. Also, the contribution on bank allocation algorithms are not related to the existing work. This thesis performs analysis to obtain the number of banks to be allocated for each core, whereas the existing work discussed in this section develops memory controller designs.

3.6 Interconnect Designs

Another source of interference in multi-cores is the system interconnect that is shared among all the cores. This section discusses the various designs of interconnect (box 2 in Figure 3.1) and how they address the WCET estimation challenge that was discussed in Section 2.1.3. The system interconnect links the cores with the memory subsystem and is generally shared across all the cores.

Probabilistically analyzable interconnect designs for multi-cores was proposed by Jalle *et al.* [74]. The design obtains probabilistic WCET estimates using two arbitration policies: 1) Lottery arbitration and 2) Randomized-permutation. By randomizing the access to the bus, this work associates a probability with every memory access and estimates WCET.

The custom architecture described in Paolieri *et al.* [26] proposes an interconnect design which follows a fixed pattern for cores to access the DDR memory. The design has a two-level arbitration scheme: inter-core (arbitrates between different cores) and intra-core (arbitrates within a core). The intra-core real-time arbiter has buffers for each bank, wherein the memory requests from each core are sorted based on the bank that it needs to access. For real-time tasks, First-In-First-Out (FIFO) arbitration scheme is used and for all the other tasks, parallel out-of-order execution that targets different banks is used. When there are multiple real-time tasks from different cores trying to access memory simultaneously, a round-robin arbitration (inter-core arbiter) is performed to provide predictable access to the interconnect.

Predictable Network on Chip (NoC) Designs System interconnects are synchronous and allow one transaction at a time, have a central arbitration scheme, limited bandwidth, and low cost. NoCs on the other hand forms the communication subsystem on an integrated circuit. It is different from a system interconnect from the fact that it applies networking principles and improves scalability and

power efficiency of system designs. It allows pipelining of requests, different architectures, transaction re-ordering, etc.

A summary of different NoC designs and its scalability was studied by Goossens *et al.* [75]. NoCs have a constant performance to cost ratio, i.e., the costs linearly depend on bandwidth and latency. The study concluded that Time Division Multiplexed (TDM) router is the best-suited arbitration scheme that is predictable, composable and also low cost to implement for NoC designs. The potential future directions of work for NoCs are to improve its reliability and quality of service, provide guaranteed performance, reduce complexity and increase portability of the design.

A circuit switched NoC based on TDMA arbitration to be used in RTA was proposed by Schoeberl *et al.* [76]. This work introduces a fast and efficient (reduced complexity) hardware NoC design. It formulates an ILP to find a schedule that provides equal bandwidth between all the cores connected to the NoC. TDMA static scheduling assures guaranteed bandwidth and a fixed latency to transmit data over the NoC.

A time-predictable TDMA memory arbiter for multi-cores was studied by Schoeberl *et al.* [77]. Each core has a local interface to the NoC and the NoC itself has an interface with the memory through a memory controller. The local interface at the core has a local TDMA schedule. Whenever there is a memory request from the core and the local interface obtains a TDMA slot, an acknowledgment is sent to the core and the request is sent to the memory controller interface. At this interface, the request is served using a handshaking policy. After the request is serviced, a broadcast message is sent with the core identity tag to return the data back to the core.

Discussions This subsection presented solutions that address the unpredictability in the access to the shared interconnect. The interconnect designs focused on either adapting the existing interconnect architecture or generalizing this architecture to NoCs. Most solutions for real-time system interconnects has primarily focused on TDMA arbitration. The contributions discussed in this subsection are orthogonal to the proposed architecture and the DDR bank allocation algorithms as the solutions proposed for predictable NoC design can be integrated on the FPGA portion of the proposed architecture if they can be made AXI protocol compliant.

3.7 Conclusion

Thus, from the literature study it can be observed that there have been significant number of contributions to all three categories mentioned in the Figure 3.1 to obtain predictable execution of tasks on a multi-core. The first two categories which are "COTS hardware and unmodified middleware" and "COTS hardware and modified middleware" focused solely on analysis techniques and modifications to the middleware. These contributions do not directly relate to the dissertation as the focus is to build custom hardware solutions to address the challenges in multi-cores. The last category "Custom hardware and unmodified middleware" has contributions that individually address the challenges in the different components of the multi-core. These contributions are towards building hardware solutions to obtain predictable WCET estimation. The difference between the contribution in this thesis with respect to the contributions in this category is that, the thesis provides a SoC platform in which the COTS multi-core can be used in a predictable manner and also serves as a platform to test and verify different contributions. Also, the solutions proposed for allocating banks to cores is different from the contributions in this category as it does not focus on the analyses to bound worst-case execution time. Instead, it uses these results to obtain optimal bank allocation strategies under different scenarios which will help utilizing DDR memory optimally.

Chapter 4

Challenges in using COTS multi-core processors in real-time systems

4.1 Motivation

Based on the discussion in Section 2.1.3 of Chapter 2, it is evident that the primary challenges in estimating the WCET in multi-cores arise from the fact that these processors share much of their memory subsystem. Although, it is a known fact that sharing of resources within a multi-core processor would increase the execution time of the tasks executing on them, the aim of this chapter is to experimentally quantify the impact of these shared resources.

This chapter evaluates a representative Commercial-Off-The-Shelf (COTS) multi-core architecture, namely the P4080 QorIQ processor from Freescale Semiconductor. Through extensive architectural analysis and experimental evaluations it is shown that the execution overhead due to contention for shared resources increases exponentially with the number of contending cores in the worst-case.

4.2 Contributions of this work

The main contributions of this chapter can be summarized as follows:

1. A holistic architectural analysis of the chosen multi-core processor is presented through experiments for timing predictability. Previous studies have discussed the impact of shared hardware on execution time [78–80], and these are experimentally validated in this work. Nowotsch et al. [81]

have done some preliminary experiments related to the cache coherency mechanism and the work by Rodolfo Pellizzoni *et al.* [82] considers to estimate the memory interference delay for multi-core systems, but to the best of my knowledge, there is no comprehensive experimental study that focuses on all aspects of the shared hardware on a COTS multi-core platform. These experiments also help to quantify the amount of impact from contentions for these shared resources, which was not feasible in earlier studies.

2. **The experiments in this chapter show that the latency for accessing a shared resource as a function of contending accesses, increases exponentially with respect to the number of contending cores.** In fact the amount of increase is so much that it almost nullifies the benefits of parallel execution (e.g., upto four-fold increase in execution with seven cores executing in parallel).
3. From the analysis it can be concluded that the incentive of using a COTS multi-core (without any modifications to the architecture) for memory intensive applications is totally negated by the exponential waiting time across many shared resources. Through changes to the software architecture it may be possible to achieve predictability by ensuring that accesses to shared hardware from different cores do not interfere [83]. This solution however has limited scope, because it does not scale well with the number of cores, and it would be very challenging to execute memory-intensive applications in parallel while reducing contention.

The evaluation of the P4080 processor provides few key observations which are as follows. It is observed that, contention for shared resources such as memory and cache controllers leads to an exponential increase in the execution time of an application with increasing number of cores. Though the system bus is claimed to have high bandwidth, impact of cache coherency mechanism on the execution time of an application is also very significant and increases exponentially. Further, it is also observed that the data of a particular core decreases in the shared caches as the number of cores accessing the shared cache increases. Resource partitioning has been suggested as one of the methods to achieve timing predictability on these platforms. But the technical document on resource partitioning from Freescale clearly states that even if every resource on the core is partitioned, the system bus and the DDR controller is always a shared resource and hence the contention for this resource cannot be eliminated (see Page 31 in [84]).

Static analysis techniques generally assume either Time Division Multiple Access (TDMA) or Round-Robin (RR) arbitration strategies for shared resources. The experiments clearly show that this is not the case, because TDMA or RR would result in a linear increase in overhead as opposed to the exponential

Figure 4.1: CodeWarrior TAP (source: 4.1)

increase that is observed. Although hardware models in static analysis can be modified based on the observation in this work, the resulting pessimism will make them impractical for use with memory-intensive application.

P4080DS, an eight-core processor from Freescale, is used as the target hardware for conducting the experiments [1] [85](Details of the architecture in Section 2.1.1 in Chapter 2). One of the primary reasons for selecting the p4080 processor for these studies is the level of configurability and controllability that the embedded processor offers, which is explained in the next sections. All the experimental setup, including scripts for configuring P4080DS and corresponding code were developed using Code warrior IDE from Freescale [86].

4.3 Code Warrior TAP

The CodeWarrior TAP probe shown in Figure 4.1 allows a personal computer workstation to communicate with Freescale Power Architecture. The CodeWarrior TAP probe provides visibility control of the target system using a JTAG interface, and connects to the host computer through an Ethernet or USB interface [87]. The CodeWarrior TAP probe provides the following key benefits:

Figure 4.2: P4080 non-partitioned memory hierarchy

1. **Visibility:** Allows to view the register values of the target system. The program execution can be halted and examined.
2. **Control:** Enables to control the system by programming, modifying the registers, step through the code and use breakpoints to control program flow.

The CodeWarrior TAP probe works in tandem with the CodeWarrior debugger and provides control over the emulation functions and the target system. CodeWarrior is an IDE provided by Freescale Semiconductor to assist development, debugging and profiling of applications targeted for Freescale processors. In all the experiments, the configuration and programming of P4080 cores is done using CodeWarrior 10.4.

4.4 P4080 configuration

In the architecture of P4080, the DDR memory controller, CPC controller and the CoreNet Fabric (CCF) are shared among the eight processing cores. The test cases are meant to target each of these components individually. Hence, the P4080 device is configured as shown in Figure 4.2. Each core has its own local caches L1 and L2, while the access to L3 cache is managed by CPC controllers through CCF. The 2 GB DDR is on the farthest point in the chain and access is again made through CCF. Such a configuration is defined in the CodeWarrior project initialization script.

P4080 also provides a mechanism to partition the L3 cache and DDR. This partitioning is meant to avoid the replacement of data from a core (further referenced as the target core), by data from another

Figure 4.3: P4080 partitioned memory hierarchy

core (further referenced as the interfering core). The Figure 4.3 depicts the memory hierarchy with L3 and DDR partitioned. For creating partitions in the DDR memory, Local Access Windows (LAWs) are configured. P4080DS permits to define 32 LAWs by providing 32 sets of registers. Each LAW is defined by three registers, namely, LAW_LAWBARHn, LAW_LAWBARLn and LAW_LAWBARn. The partitions defined by LAWs, do not stop any core from using any partition. It is the responsibility of the software(application), to make sure that the target core does not allocate memory outside its assigned partition. The actual use of DDR memory partitioning using LAWs, is to partition the CPC. Using the CPC partitioning control registers, namely, CPCx_CPCPIRn, CPCx_CPCPARn and CPCx_CPCPWRn, the CPC ways are associated with the DDR memory partitions. When the allocation is done correctly, the hardware ensures that only the configured DDR partitions are allocated to the corresponding CPC ways.

The L3 cache parameters including write back/write through mode, and cache replacement policy, is configured by modifying the register CPC1_CPCCSR0 and CPC2_CPCCSR0. L1 and L2 cache parameters are configured by modifying L1CSR0, L2CSR1, L2CSR2 and L2CSR0 registers. The details on what configuration is used for which test case is described in the respective test case description.

In all the partition related experiments, the DDR memory is partitioned into 8 equal sections, each being 256 MB in size. The CPC or the L3 cache is also divided into 8 equal parts by assigning 4 ways to each DDR partition. The effective size of each CPC/L3 partition, thus becomes 128 KB. In all the experiments, branch prediction is disabled by appropriately configuring the BUCSR register. The reason for disabling branch prediction is to avoid any known variability in the execution of the program. This is because, if the branch prediction was enabled, the branch prediction mechanism would always try to

Figure 4.4: P4080DS Hardware setup

up-front predict which possible sequence of code would be executed. Program execution time varies based on the result of the prediction. This essentially means that there could arise a variability in the execution time of a program on every run.

4.5 Generic Experimental Setup

The setup, shown in Figure 4.4, has a Desktop PC (host) running Windows 7 OS with CodeWarrior IDE and connected to an ethernet hub [86]. The P4080DS is accessed using the debugger CodeWarrior Ethernet Tap via JTAG, and it communicates with the host machine using ethernet [88].

The P4080DS registers are initialized using an assembly language script, as these configurations are necessary to execute programs without the support of an operating system. The Local Access Windows (LAW) are a set of 32 registers used to define programmable regions of memory and the internal inter-connections of the chip to route accesses [89]. These regions include all the private and shared caches as well as DDR memory. Memory Management Unit (MMU) is used to translate virtual to physical memory and it is done using Translational Lookaside Buffers (TLB). TLBs are used to configure memory ranges allocated to the LAWs to be cache-able. All three levels of caches are initially enabled for all the experiments. The default mode of initialization for the caches is write through and inclusive, meaning

writing is done synchronously both to the cache and to the backing store, and all the data in local cache will also be present in the next level of cache.

For all the experiments, Core 0 is taken as the default core for measurements, while the other cores are used to measure the impact on Core 0 with interference from increasing number of cores. There are two types of measurements:

- 1) To measure the number of cycles for execution of an application on core 0 (execution time), and
- 2) To measure the number of memory requests to the shared resources by obtaining the trace of the application execution.

It is important to note that the first measurement only focuses on the application executing on Core 0, whereas the second measurement focuses on applications executing on all the cores. The term **active cores** is used in this section to identify the number of cores that are participating in any experiment. Five runs were performed for each experiment, and then the mean value of the observations is computed. A small number of runs are sufficient, because the deviation of the measured values from the mean was found to be no more than 1.5%, and in most cases it was very close to zero. As the number of cores are increased in the experiment, the contention to access the shared resource also increases. The graphs that are plotted captures the effect of increase in execution time or data traffic when the amount of contention is increased. The applications that executing on these cores are consecutive memory reads and writes and hence this can be replicated on any other embedded platform for comparative evaluation.

Curve Fitting Parameters: Measurements from the experiments are fitted with a curve using MATLAB (R2012b) curve fitting toolbox to analyze the amount of contention in each of the shared resource [90] [91]. The best fit curve is identified by closeness of Root Mean Square Error (RMSE) to zero. RMSE is the square of the mean of all residuals [92]. Confidence interval indicates how uncertain are the fitted coefficients. For the curve fitting, this value is kept at 95%. Prediction bound (PI) indicates the probability of a new observation to be contained within the lower and upper bound. This interval is kept to be at either 95% or 99% based on the RMSE of the fitted curve. Datasets can be fitted with polynomials of different order. Increasing the order of the polynomial increases the number of data points lying on the curve, thereby decreasing the error (RMSE). To select the polynomial that best fits the dataset, the order of the polynomial is increased until the RMSE is less than or atleast close to 1.5%. This is because the measurement variation is bounded by 1.5%, and any further improvement of RMSE may not increase the accuracy of the fitted curve. In most cases, the RMSE does not go below 1.5%, as the order of the fitted curve could not be increased further owing to the limited size of the dataset. All

the fitted curves have their x-axis values normalized by a mean and Standard Deviation (SD). Thus, to find the value that will be an input to the fitted curve, we must use $(x - \text{mean})/\text{SD}$.

4.6 Experiment I: Contentions in DDR controller

4.6.1 Description

To understand the contention in DDR controller, all the caches are disabled. This ensures that all the memory accesses in the application actually translate to a physical memory access and not the caches. The system interconnect does not contribute to the contention here as there is enough bandwidth for memory accesses. The system interconnect contentions occur when cache coherency protocol is active. The initialization script is executed to obtain the above requirement. The pseudo code for the test program used on the target core i.e. core 0 is as follows. Similar code is used on the interfering cores, with an infinite while loop. The infinite while loop ensures that the interfering cores are generating continuous traffic for the DDR controller.

```
DECLARE array[1024]
FOR 1024 iterations
WRITE array[i] with random number
INCREMENT i
END FOR
```

The execution time (latency) of the above code is measured in terms of CPU cycles. The test is repeated by increasing the number of active cores from 1 to 8.

4.6.2 Result and Analysis

Figure 4.5 exhibits an increase in latency while executing on core 0, as the number of interfering cores increase from 0 to 7, for the non-partitioned DDR configuration. The x-axis of the fitted curve represents the number of interfering cores and the y-axis represents execution time normalized w.r.t. the execution time on a single core, when all the interfering cores are inactive. The best fit for the measurement data is a linear curve, represented as $f(x) = p1 * x + p2$, where $p1 = 0.9911$ and $p2 = 0.9925$, when confidence bound is set to 95%. The RMSE for the curve is 0.0340.

Similarly, figure 4.6 exhibits the increase in latency on core 0, as the number of interfering cores increase from 0 to 7, for the partitioned DDR configuration. The best fit for the measurement data is a

Figure 4.5: Contentions in non-partitioned DDR

polynomial curve, represented as $f(x) = p1 * x^6 + p2 * x^5 + p3 * x^4 + p4 * x^3 + p5 * x^2 + p6 * x + p7$, where $p1 = 0.0005$, $p2 = -0.01027$, $p3 = 0.07534$, $p4 = -0.2355$, $p5 = 0.3189$, $p6 = 0.3183$ and $p7 = 1$, when confidence bound is set to 95%. The RMSE for the curve is 0.0201.

As can be seen in Figure 4.5 and 4.6, the latency increases significantly with the increase in number of interfering cores. Since, all the caches are disabled, it can be inferred that the increase in latency is due to the contentions in DDR controller only. However, an interesting observation is that for the non-partitioned DDR configuration, the increase is linear in nature while, for the partitioned DDR configuration, it is a polynomial of degree equivalent to the number of interfering cores. It appears that the difference in behavior is because of the way in which DDR traffic is handled for the respective configurations. Since the low level architectural details of P4080 device are unavailable, it is beyond the scope of this work to provide a concrete explanation for such difference in behavior. The highlight of these observations is that the degree of the fitted polynomial curve is 6, which is comparable to the number of contending cores. Hence, it can be inferred that the contention overhead increases exponentially with the number of contending cores.

Figure 4.6: Contentions in partitioned DDR

4.7 Experiment II: CPC contentions in partitioned CPC controller

4.7.1 Description

In this experiment, all local caches (i.e. L1 and L2 level caches which are private to the cores), are disabled in order to study the effect of CPC cache contentions (L3 level cache which is shared across the cores). By disabling only the local caches, it is ensured that all the memory accesses in the code translate to either an L3 access (in case of a cache hit) or the DDR memory (in case of a cache miss). The DDR memory and CPC (for L3 cache) were partitioned using LAW, CPCx CPCPIRn, CPCx CPCPARn and CPCx CPCPWRn registers. This experiment is conducted for write-through (WT) L3 cache configuration, and Psuedo-Least-Recently-Used (PLRU) replacement policy. The corresponding initialization script is executed. Figure 4.3 depicts this memory configuration, with L1 and L2 disabled by appropriately configuring the registers.

The pseudo code for the test program used on the target core 0 is as follows. The first for loop in the code is intended to populate the L3 cache with array elements. The second loop is to read the array elements. The same code is used on the interfering cores. Only the execution time of the second loop is profiled (number of CPU cycles).

```
DECLARE array[n]
FOR n iterations
WRITE array[i] with random number
INCREMENT i
END FOR

FOR n iterations
READ array[i]
INCREMENT i
END FOR
```

The test is repeated for increasing number of interfering cores from 0 to 7, while keeping the number of memory accesses constant i.e. the array size as 256.

4.7.2 Results

Figure 4.7 exhibits the change in CPC cache hit ratio, on core 0, as the number of interfering cores increase from 0 to 7. The x-axis of the fitted curve represents the number of interfering cores and, the y-axis represents the CPC cache hits normalized w.r.t. the number of cache hits when none of the interfering cores is active.

Figure 4.8 exhibits the increase in latency on core 0, as the number of interfering cores increase from 0 to 7. The x-axis of the fitted curve represents the number of interfering cores and, the y-axis represents the difference in execution time as compared to the case where none of the interfering cores is active. The best fit for the measurement data is a polynomial curve, represented as $f(x) = p1 * x^6 + p2 * x^5 + p3 * x^4 + p4 * x^3 + p5 * x^2 + p6 * x + p7$, where $p1 = 1.92$, $p2 = -28.87$, $p3 = 149.1$, $p4 = -295.6$, $p5 = 191.9$, and $p6 = -2.88$, when confidence bound is set to 95%. The RMSE for the curve is 76.0676.

As can be seen in Figure 4.8, with the increase in number of interfering cores, there is an increase in latency even when the cache hit ratio remains at unity (Figure 4.7). Since, there are no cache misses, it can be inferred that the increase in latency is due to the contentions at CPC controller only. Moreover, the degree of the fitted polynomial curve is 5, which is comparable to the number of contending cores. Hence it can be inferred that the contention overhead increases exponentially with the number of contending cores.

Figure 4.7: Contentions in partitioned CPC

Figure 4.8: Contentions in partitioned CPC

4.8 Experiment III: CPC contentions in non partitioned CPC controller

4.8.1 Description

This experiment is similar to Experiment 4.7. The only difference is that the DDR memory and L3 cache are not partitioned in this case. The corresponding initialization script is executed. Figure 4.2 depicts this memory configuration, with L1 and L2 disabled by appropriately configuring the registers required. The test program for this experiment is same as Experiment 4.7, wherein the execution time of the second loop is profiled in terms of the number of CPU cycles. The test is repeated for increasing number of interfering cores from 0 to 7, while keeping the number of memory accesses i.e. the array size as 256.

4.8.2 Results

Figure 4.9 exhibits the change in CPC cache hit ratio, on core 0, as the number of interfering cores increase from 0 to 7. The x-axis of the fitted curve represents the number of interfering cores and, the y-axis represents the CPC cache hits normalized w.r.t. the number of cache hits when none of the interfering cores is active. Figure 4.10 exhibits the increase in latency on core 0, as the number of interfering cores increase from 0 to 7. The x-axis of the fitted curve represents the number of interfering cores and, the y-axis represents the difference in execution time as compared to the case where none of the interfering cores is active. The best fit for the measurement data is a polynomial curve, represented as $f(x) = p1 * x^6 + p2 * x^5 + p3 * x^4 + p4 * x^3 + p5 * x^2 + p6 * x + p7$, where $p1 = -14.36$, $p2 = 261.6$, $p3 = -1753$, $p4 = 5377$, $p5 = -7429$, $p6 = 3610$ and $p7 = -5.356$, when confidence bound is set to 95%. The RMSE for the curve is 313.7585.

As can be seen in Figure 4.10, with the increase in number of interfering cores, there is an increase in latency even when the cache hit ratio remains at unity (Figure 4.9). Since, there are no cache misses, it can be inferred that the increase in latency is due to the contentions at CPC controller only. Moreover, the degree of the fitted polynomial curve is 6, which is comparable to the number of contending cores. Hence it can be inferred that the contention overhead increases exponentially with the number of contending cores.

Figure 4.9: Contentions in non-partitioned CPC

Figure 4.10: Contentions in non-partitioned CPC

4.9 Experiment IV: Coherency Contentions in CCF

4.9.1 Description

In order to study the effect of contentions in CCF, coherency traffic is generated by letting all active cores write to the same memory location. All the caches are enabled, in order to have the maximum amount of coherency traffic as hardware will update all the caches to have an updated entry for the targeted memory location. The initialization script is executed.

The pseudo code for the test program used on the target core i.e. core 0 is as follows. Similar code is used on the interfering cores, with a infinite while loop. The infinite while loop ensures that the interfering cores are generating continuous coherency traffic.

```
FOR 512 iterations
WRITE MEM_PTR with random number
END FOR
```

The execution time of the above code is measured in terms of CPU cycles. The test is repeated by increasing the number of interfering cores from 0 to 7.

4.9.2 Results

Figure 4.11 exhibits the increase in latency on core 0, as the number of interfering cores increase from 0 to 7. The x-axis of the fitted curve represents the number of interfering cores and the y-axis represents the difference in execution time as compared to the case where none of the interfering cores is active. The best fit for the measurement data is a polynomial curve, represented as $f(x) = p1 * x^6 + p2 * x^5 + p3 * x^4 + p4 * x^3 + p5 * x^2 + p6 * x + p7$, where $p1 = -0.1646$, $p2 = 1.539$, $p3 = 6.673$, $p4 = -108.5$, $p5 = 332.2$, $p6 = -91.88$ and $p7 = -0.9828$, when confidence bound is set to 95%. The RMSE for the curve is 57.5762.

The mathematical lower bound on the fitted curve lies on the negative y-axis, which does not have a meaning in physical sense. It is, therefore, not represented on the plot. As can be seen in Figure 4.11, the execution time increases with the increase in the number of interfering cores. The degree of the fitted polynomial curve is 6, which is comparable to the number of contending cores. Hence, it can be inferred that the contention overhead increases exponentially with the number of contending cores. Since L1 and L2 have dedicated local interconnects to their respective cores, the increased latency is

Figure 4.11: Coherency contentions in non-partitioned CPC

due to the contentions caused by the coherency traffic at CCF and CPC controller. However, with the available profiling tools for P4080, it is beyond the scope to further isolate the contentions at CCF from contentions at CPC controller. Hence, this demonstrates a cumulative increase in latency due to coherency traffic.

4.10 Experiment V: Cache evictions in non-partitioned CPC controller

4.10.1 Description

In this experiment, all local caches (i.e. L1 and L2 level caches which are private to the cores), are disabled in order to study the effect of cache evictions in the L3 cache. By disabling only the local caches, it is ensured that all the memory accesses in the code translate to either an L3 access (in case of a cache hit) or the DDR memory (in case of a cache miss). The DDR memory and CPC (for L3 cache) were partitioned using LAW, CPCx CPCPIRn, CPCx CPCPARn and CPCx CPCPWRn registers. This experiment is conducted for two different L3 cache configurations, namely write-back(WB) and write-through(WT), and two different cache replacement policies, namely First-In-First-Out (FIFO) and Psuedo-Least-Recently-Used (PLRU). When multiple cores simultaneously populate the cache with its data, the data of one of the cores will be replacing the data of an already existing core in the shared cache (L3). This leads to an inter-core data interference which results in more cache misses and in turn

increases the execution time of the program.

The pseudo code for the test program used on the target core 0 is as follows. The first for loop in the code is intended to populate the L3 cache with array elements. The second loop is to read the array elements. A for loop similar to the first loop is used on the interfering cores, to make sure that all active cores interfere in the process of populating the cache. It is the second loop which is profiled for CPC read hits. All the interfering cores are paused during the execution of the second loop on the target core.

```
DECLARE array[n]
FOR n iterations
WRITE array[i] with random number
INCREMENT i
END FOR

FOR n iterations
READ array[i]
INCREMENT i
END FOR
```

The test is repeated for increasing number of interfering cores from 0 to 7. In order to see the impact of evictions, two different array lengths have been considered, one where the data size is less than the size of the cache and one where it is equal to the size of L3 cache i.e. 2 MB. The case where data size is greater than the L3 cache size, is not considered as the latter case already creates a worst case scenario where the L3 cache contents are completely replaced by the data from interfering cores.

4.10.2 Results

Figures 4.12, 4.13, 4.14 and 4.15, exhibit the decrease in L3 cache (or CPC) hits, with the increase in number of interfering cores for 1 MB (less than the L3 cache size i.e. 2MB) array size. The x-axis of the fitted curve represents the number of interfering cores and, the y-axis represents the CPC cache hits normalized w.r.t. the number of cache hits when none of the interfering cores is active. The best fit for the corresponding measurement data, is an exponential curve, expressed as $f(x) = ae^{bx} + ce^{dx}$. The values of coefficients a, b, c and d, with 95% confidence bound, for different configurations and replacement policies are given in Figure 4.16.

Similarly, Figures 4.17, 4.18, 4.19 and 4.20, exhibit the decrease in L3 cache (or CPC) hits, with the increase in number of interfering cores for 2 MB (same as L3 cache size) array size. The best fit for the

Figure 4.12: Contentions in non-partitioned CPC for 1 MB, write through and fifo

Figure 4.13: Contentions in non-partitioned CPC for 1 MB, write back and fifo

Figure 4.14: Contentions in non-partitioned CPC for 1 MB, write through and plru

Figure 4.15: Contentions in non-partitioned CPC for 1 MB, write back and plru

Cache replacement policy	L3 cache configuration	Coefficients			
		a	b	c	d
FIFO	Write back	0.1672	0.1296	0.8463	-0.1039
PLRU	Write back	0.05471	0.2135	0.9569	-0.07087
FIFO	Write through	0.4524	0.0556	0.5638	-0.1769
PLRU	Write through	0.1811	0.1216	0.8327	-0.1035

Figure 4.16: Curve fitting coefficients for 1 MB data size

Figure 4.17: Contentions in non-partitioned CPC for 2 MB, write through and fifo

corresponding measurement data, is an exponential curve, expressed as $f(x) = ae^{bx} + ce^{dx}$. The values of coefficients a , b , c and d , with 95% confidence bound, for different configurations and replacement policies are given in Figure 4.21.

The effect is much worse for the case where data size is increased to 2 MB (i.e. the size of the shared L3 cache), as evident from Figures 4.17, 4.18, 4.19 and 4.20. The unpredictability in cache hit ratio due to asynchronous cache evictions is a well-known problem in multi-core systems, which is typically resolved by partitioning the shared cache. During asynchronous cache evictions, the data of one of the cores in the cache is replaced by the data of another core. In a partitioned configuration, each partition acts as a cache, local to the corresponding core. Thus, preventing any asynchronous cache evictions, at the cost of reduced cache space available to each core.

4.11 Experiment VI: Cache evictions in partitioned CPC controller

4.11.1 Description

This experiment is similar to Experiment 4.10. The only difference is that the DDR memory and the L3 cache are partitioned. Figure 4.2 depicts this memory configuration, with L1 and L2 disabled (by appropriately configuring the registers). The partitioning based initialization script is executed. The code for this experiment is similar to Experiment 4.10. However, the target array lengths are reduced to 256 kB (i.e. same the L3 cache partition size) and 512 kB (i.e. less than the L3 cache partition size).

Figure 4.18: Contentions in non-partitioned CPC for 2 MB, write back and fifo

Figure 4.19: Contentions in non-partitioned CPC for 2 MB, write through and plru

Figure 4.20: Contentions in non-partitioned CPC for 2 MB, write back and plru

Cache replacement policy	L3 cache configuration	Coefficients			
		a	b	c	d
FIFO	Write back	-7315	-0.4279	7316	-0.4279
PLRU	Write back	7933	-0.4112	-7932	-0.4112
FIFO	Write through	-3261	-0.4293	3262	-0.4294
PLRU	Write through	-113300	-0.4092	113300	-0.4092

Figure 4.21: Curve fitting coefficients for 2 MB data size

4.11.2 Results

Figures 4.22 demonstrates that the cache hit ratio remains unchanged for the partitioned configuration, irrespective of the number of interfering cores and the data size. The x-axis of the curve represents the number of interfering cores and, the y-axis represents the CPC cache hits normalized w.r.t. the number of cache hits when none of the interfering cores is active. This behavior is expected, as each L3 partition is local to the corresponding core. However, even if the cache hit ratio remains unchanged, the execution time still increases with the increase in number of interfering cores. This is because, as the number of cores increase, the number of partitions in the shared cache increases, which in turn would decrease the amount of data that could be cached for each core. This implies there would be more cache misses and that would result in an increased execution time.

4.12 Conclusion

Experiments discussed in this chapter bring to light the presence of contention due to shared hardware resources in COTS multi-core processors. This chapter focused on the P4080 QorIQ processor

Figure 4.22: Contentions in partitioned CPC for 256KB, write back and plru

from Freescale that has typical architectural features found in most COTS multi-cores such as shared caches, memory controllers and on-chip communication network. Time-predictability issues arising from architectural features such as branch prediction, hardware prefetching and TLB misses were ignored in this studies by disabling them, primarily because these issues are not specific to multi-cores and therefore not the focus of this work. The Section 4.4 discusses in detail the possible architectural configurations in the chosen processor. Some configurations such as write through cache policy aid in predictable execution of the tasks, and therefore chosen to be used in the experiments. Some other configurations such as cache locking and partitioning also aid predictable execution of the tasks(for e.g. see [84]), but were not considered in the experiments. Although cache partitioning eliminates data access unpredictability arising from the shared CPC, the CPC controller is still shared between the cores and the experiments in this chapter show that the impact of this contention is significant. Further, partitioning will negatively impact applications that actually share data and can be executed in parallel. Cache locking on the other hand has challenges in determining which data to lock so as to reduce the overall impact of cache misses on execution time, and similar to partitioning, it also does not address the issue of contention on the CPC controller itself.

Six experiments were performed to evaluate the impact of contention for the shared resources on a multi-core processor. These include experiments focused on the contention for DDR memory controller, CPC controller, corenet bus due to cache coherency mechanism, and data in the CPC due to evictions by cores executing in parallel. Graphs presented in the section highlight the impact on application

execution times from these contentions, and show that in general there is an exponential increase in the execution time with increasing number of contending cores. It can also be observed that the increase in execution time is prohibitively high as we scale the number of active cores, and almost negates all the benefits of parallel execution.

These experiments also highlight some interesting and unique aspects of the P4080 architecture in terms of predictable execution of the tasks. The processor has two independent CPCs, each connected to an independent DDR memory controller. There is however no configuration mechanism to control which CPC must be used for a particular request, except through static allocation of memory. Although in most of the experiments the accesses were roughly split equally between the two CPCs, there was no mechanism to control this precisely. This has an impact on timing predictability because it directly affects contention on the CPC and DDR memory controllers. It is generally believed that the bandwidth of the corenet bus on COTS multi-core architectures are quite high, and therefore would not impact execution times significantly. The experiments on coherency mechanism clearly show that this is not the case, and even with five or six active cores, the contention can increase exponentially. Another interesting outcome of the experiments is the impact of cache controller contention on execution times (e.g., the CPC controller in P4080). Although memory controller contention is widely studied, the impact from cache controllers is not that well understood. The experiments show that even cache controller contention can significantly affect WCET.

Based on the results presented in this chapter, it is evident that sharing of hardware resources leads to a significant increase in execution time of applications that execute in parallel. An obvious mechanism to eliminate this contention, as suggested by Freescale themselves (see Page 31 in [93]), is to partition all the shared resources among the cores. This means the CPC and DDR memory controllers, the CPC itself, and the corenet bus, must all be partitioned. Although the controllers and the CPC can be partitioned among the cores through configuration, the number of these controllers is typically much less than the number of cores in COTS multi-cores. The partitioning approach is therefore not scalable with the number of cores. Further, to the best of my knowledge, there is no configuration mechanism available to partition the corenet bus among the cores. One may then wonder if it is possible to deduce the contention on the corenet bus through static analysis of application code. There are several challenges to this analysis including unknown arbitration mechanism of the corenet, difficulty in statically predicting memory access patterns of application code due to control and data dependencies, and cyclic dependency between delay due to contention and memory access pattern of application code. Since the hardware cannot be fully partitioned, another mechanism to achieve timing predictability would be through changes to the software architecture of applications. By ensuring that no two applications exe-

cuting in parallel will request for access to shared resources at the same time, it is possible to eliminate the above mentioned contentions. This solution is limited in scope however, because it does not scale with the number of cores, and it is thus challenging to execute memory-intensive applications in parallel. The following chapter introduces the hybrid COTS FPGA architecture for multi-cores to overcome these challenges for its use in RTS.

Chapter 5

Time-Predictable System-On-Chip architecture for COTS multi-cores

5.1 Introduction

As discussed in the previous chapters, in order to back the increasing demand in the hardware performance for RTS, processors with multiple cores became the obvious choice due to their superior Size, Weight and Power characteristics. However, we also know from the results in Chapter 4, that the time taken by an application executing on a core generally depends on how every other application utilizes the shared resources in the multi-core. To address the WCET estimation challenge due to shared hardware resources in multi-core processors discussed in Chapter 4, in this chapter, a novel System-on-Chip (SoC) architecture based on COTS multi-cores is presented. Specifically, this chapter proposes, designs and implements an architecture which helps COTS multi-core to access its shared memory subsystem predictably. This architecture is proposed on hybrid hardware platforms comprising a COTS multi-core which is tightly coupled with Field Programmable Gate Array (FPGA), e.g., Xilinx Zynq ZC706.

Hybrid-FPGA platforms, which comprise COTS multi-cores executing in tight co-ordination with programmable hardware logic, have been introduced in recent years. Examples of such platforms include the Zynq series from Xilinx [94] and the Cyclone-V from Altera [95]. These platforms have all the benefits of COTS processors such as low cost and SWaP, but at the same time offer flexibility to customize the hardware logic via FPGAs. By re-routing all the memory requests from the COTS processor to the FPGA in the SoC, the architecture is able to support predictable execution of real-time applications.

Figure 5.1: Zynq Platform Overview

The proposed architecture clearly depicts the potential for utilizing COTS multi-cores with tightly coupled FPGA for RTA with timing requirements. This architecture can also be transformed into an Application-Specific Integrated Circuit (ASIC) to further improve SWaP characteristics, if the capability of FPGAs to be re-programmed is not required. Primarily, the goal is to investigate the feasibility of developing predictable architectures with full functionality using hybrid-FPGAs. As a first step in this direction, the feasibility of this architectural idea is tested as follows:

1. Design of an efficient architecture for re-routing memory requests from the COTS multi-core to the programmable logic using Memory Interface Generator controller which is a custom memory controller provided by Xilinx, and
2. Design a round-robin arbiter to control the arbitration of the requests from the cores on the programmable logic using a custom developed memory controller to provide a predictable access to the DDR.

Developing a cache hierarchy with predictable shared cache design on the programmable logic, using both the private caches in the COTS multi-core and integrating the custom memory controller with the shared cache, will be considered in future work.

5.2 Platform Overview

This section provides an overview of the Zynq FPGA platform, its architecture details and the tools used for the design.

ZC706 Development Platform The development platform used for design and experiments is ZC706 Evaluation Board from Xilinx. This is an evaluation board for XC7Z045 Zynq All Programmable SoC. The abstract level block diagram of this platform is shown in Figure 5.1. The key features include 1GB 32-bit wide DDR3 memory on the Processing System (PS) side, two 128Mb Quad-SPI flash memory and 1GB 64-bit wide DDR3 memory on the Programmable Logic (PL) side. The platform has got SD card slot and can be booted from SD card. This enables us to run both cores in bare-metal mode. By bare-metal, it means that there is no Operating System (OS) or a hypervisor to control the execution of the program. Instead, the program is loaded to the program memory and directly executed on the cores. The advantage of executing the program in bare-metal mode is to minimize the (operating system) overhead in the program execution and would facilitate to measure the program execution accurately. The primary reason for choosing this development platform is, the presence of 1GB DDR3 on PL side, which allows us to directly interact with memory from custom controller deployed in FPGA. The board has got USB JTAG interface for debugging software and hardware logic in FPGA. The UART interface can be used for serial console.

ZC706 platform has XC7Z054 which is Zynq-7000 series programmable SoC from Xilinx [94]. This is an attractive SoC for embedded applications which require multi-core features and custom hardware acceleration units. Zynq-7000 series include a hard Processing System (PS) and a Programmable Logic (PL). The size of FPGA varies across different devices in Zynq-7000 family. This SoC enables us to build custom hardware on FPGA and build a heterogeneous processing system using hard-core processors in PS. Figure 5.2 shows high-level architecture of Zynq-7000 SoC. The PS side has dual-core ARM-A9 processor with 32kB private Level-1 I-Cache and D-Cache. Both cores share a 512kB L2 cache. The Snoop Controller Unit (SCU) ensures cache coherency between cores. It connects the two ARM cores to the memory subsystem and also has the logic to manage the coherency of the caches. The SCU communicates with the ARM cores through a logic termed as cache coherency protocol which helps to manage the coherency of data between the L1 and L2 caches. The PL side has a Kintex-7 FPGA with 350k logic cells. The ARM cores communicate with PL using 32-bit wide general purpose(GP) AXI ports. An AXI master in FPGA can directly access the PS side DDR memory through 64-bit wide high performance slave ports. The AXI interconnects and buses establish communication link between

Figure 5.2: Xilinx Zynq Architecture(source: Xilinx Zynq TRM) [2]

all peripherals and the CPU. The ARM cores can run at maximum frequency of 800MHz and the FPGA fabric supports maximum frequency of 250MHz.

Figure 5.3 shows memory request and response path between ARM cores in PS and external memory connected to PS and PL side. The snoop controller arbitrates between two of its AXI master ports. The request to PS DDR memory is forwarded to PS DDR controller from L2 cache controller. Any request to PL DDR memory has to pass through a couple of interconnects and DDR controller in FPGA. The general purpose AXI3 master ports are used for this communication. AXI interconnects are used in FPGA for protocol conversion from AXI3 to AXI4 and clock domain crossing.

Overview of AMBA AXI Protocol The PS and PL is interfaced using AXI bus [96]. This AXI bus is based on the AMBA AXI protocol which is a burst-based protocol for high performance, high-frequency system designs. Every transaction has address and control information on address channel. The master will initiate read or write request. The data is transferred between master to slave for write request and in the reverse direction for read request. The Zynq uses AXI bus and interconnects to establish connection between CPU and its peripherals. The general purpose ports used in the request-response path between CPU and FPGA are AXI3 ports. Hence, it is necessary to have AXI3 interface for any custom hardware to connect it to GP port.

Figure 5.3: PS-PL Request-response Path

Figure 5.4: Predictable SoC architecture

Overview of Development Tools Xilinx provides variety of FPGA development tools that support programmable platforms. This work uses Xilinx Vivado Design Suite, v2014.2 [97]. Vivado is IP-centric and system-centric development environment and supports FPGA design [97]. The behavioral simulator integrated with the same tool eases the development process. This end-to-end FPGA hardware development tool makes the development process easier compared to other Xilinx tools. The software test cases are developed on Xilinx SDK, v2014.2 [98]. The SDK provides Eclipse-based development environment. It provides FPGA configuration and application debugging utility through JTAG [99].

5.3 Proposed Architecture

This section discusses the design and implementation details of the proposed architecture with respect to the Zynq 706 platform.

5.3.1 Design Overview

To mitigate the challenge of unpredictability due to shared memory subsystem that have been discussed, a novel architecture for COTS multi-cores is proposed as shown in Figure 5.4. The key idea of the proposed architecture is that, the memory requests to the DDR that are generated from the PS (ARM cores) are re-directed to the PL (FPGA) and then to the DDR which is interfaced with the PL. The main intuition behind this proposed architecture is that those components of the multi-core which have been identified to cause unpredictability (shared resources) are disabled within the multi-core and these re-directed memory requests are handled outside the core by designing specific components on the FPGA.

As a first step to build this architecture, all levels of caches along with the memory controller are disabled inside the core on the PS using the internal register settings. The high level design of the proposed architecture is shown in the Figure 5.4. The ARM A9 cores with their own bus architecture is interfaced with the Master Interconnect inside the Zynq706. Also, the caches and controllers in the ARM A9 are disabled using register settings. The master interconnect has the ability to multiplex between multiple ports of Zynq706 SoC. In the proposed architecture, the master interconnect is interfaced with the master general purpose (GP) port. This connection follows the AXI protocol. In the initial stages of the design, an existing custom memory controller IP (MIG) provided by Xilinx was used. Since the arbitration of this controller cannot be customized, an external arbiter (round-robin scheme) is used to arbitrate and send the request to the MIG controller. The MIG controller will handle only the generation

Figure 5.5: Arbiter Controller Framework

of the DDR commands and sending the memory request to the DDR3 memory.

The overall functioning of the design can be explained as follows. As observed in the Figure 5.4, the requests to the memory (DDR3 memory on PL) from the multi-core bypasses the SCU, the caches (as they are disabled) and arrives at the central interconnect of Zynq706. The ARM cores are address mapped to the GP port and hence any memory request which arrives at the central interconnect is re-directed through this GP port into the FPGA. It is then received using the AXI protocol by the FIFO buffers and are stored in them. Each core has its own set of FIFO buffers on the FPGA to store their respective requests. The buffer to be stored is identified using a core ID value in the address of the memory request. This is then followed by an arbiter which performs a round robin arbitration between the FIFO buffers and then directs the requests to the custom memory controller. The arbiter is a simple logic that alternates between the two FIFO queues based on a predefined delay and forwards the request to the memory controller. This memory controller generates the required memory commands and then sends the corresponding commands through a physical layer interface into the DDR memory.

5.3.2 Arbiter-Controller Framework

The DDR memory connected to the PL side of the architecture must be accessed only through a memory controller which is instantiated on the PL side. As a first step, the Xilinx provided custom memory controller (MIG) is used to design the architecture [94]. However, these controllers have a protected arbitration policy and signal routing mechanism which will create unpredictability in the architecture.

Figure 5.6: Arbiter with AXI interfaces

The design in Figure 5.5 explains the implemented arbiter mechanism in the proposed architecture. This design with the MIG controller by Xilinx will help to generate a PHY (physical layer interface) which will help to interface to the DDR [100]. The PHY can be re-used while designing the custom memory controller design.

In this initial design, the memory requests from the two ARM A9 cores bypass the internal multi-core components and arrive at the PL in a single port. The processing system components present in the memory path from cores to the FPGA are SCU and master interconnect as can be seen from Figure 5.3. Once, these requests reach the GP port on the PL, the AXI slave interface will handle these requests.

The components designed in the PL are the arbiter and the DDR memory controller. The arbiter module is designed to have slave and master AXI3 interface. The slave port of the arbiter interfaces with GP port via an AXI interconnect [101]. The request reordering mechanism is disabled as it introduces unpredictability in the access to the DDR memory for every request. In the instances where processors are being used for non-real time systems, performance of the processor is a key metric. Request reordering helps in improving the average case performance as it helps improving the bandwidth utilization. However, for real-time systems, the request reordering mechanism is disabled as the focus is on predictable execution. The front-end of arbiter module is an AXI slave interface as shown in Figure 5.6. The requests are buffered in the respective request FIFOs of each core. Front-end uses AXI IDs to distinguish between cores. The AXI ID generally represents the core ID from which the requests are generated. The number of outstanding requests per core depends on size (depth) of the FIFO buffer

Figure 5.7: Custom DDR Controller

and is configurable during design. The response path from the DDR memory is an independent path which interfaces back to the GP port. If the FIFO is non-empty, it indicates the presence of a memory request to the arbiter. The arbiter module, in every clock cycle looks for the FIFO non-empty signal and performs arbitration between the FIFOs and forwards the memory request. The current architecture of the arbiter uses a round-robin policy between the cores (in every clock cycle, the arbiter module switches between the two request buffers). The back-end of this arbiter is the MIG memory controller which receives the memory request and generates the commands to the DDR based on the DDR timing policy. The next step in this architecture design is to replace this Xilinx MIG controller with the custom predictable controller design and interface it with the DDR. Every slave and master interconnect core has its corresponding ACLK input, as does the underlying crossbar core. The AXI crossbar core has its own aclk input. When the clock source that is driving the aclk pin of the slave and master interconnect is different than the one that is driving the internal crossbar, an AXI clock converter core is automatically instantiated in this path by the tool. This will handle all clock domain crossing within the design [101].

5.3.3 DDR controller Design

The custom controller shown in Figure 5.7 is divided into a front-end, central core and back-end which provides a modular mechanism for designing. The arbiter itself forms a part of the front-end of the controller along with an AXI interface and memory buffer to store the incoming memory request based on the core.

The core of the controller is responsible for maintaining the DDR timing and generating the commands to the banks of the DDR. It is extremely critical as every timing constraint imposed by JEDEC standard need to be satisfied for the command to be accepted or processed by a DDR. The functions inside the controller core have been implemented using Finite State Machines (FSM) [102]. The bank FSM is responsible for handling requests to designated bank and making sure that intra-bank timings are always met. The number of banks in the design can be configured using a configurable parameter. The bank controller FSM is responsible for receiving requests from the front-end (arbiter) and forwards them to the respective bank FSM thereby making sure that inter-bank timing constraints are met. This FSM also handles the scheduling of the refresh commands by coordinating with the refresh control logic module.

The back-end of the controller is the component that controls the interface to the DDR. The most important module at the back-end is the command generator. This module decodes the memory requests to the banks into its DDR memory commands and asserts the logic levels on the bus as per DDR standards. It also makes sure that there are no operation commands sent to the DDR when there are no requests to the specific bank (as per JEDEC standards). The initialization logic is responsible for power-on initialization of the entire DDR. It is also based on the JEDEC standards.

5.3.3.1 Salient Features of DDR controller

Some of the most important features of this DDR memory controller which is responsible for its predictable execution are as follows (the details of each feature are explained in the subsequent sections).

1. Closed Row Policy: According to this policy, every row of the DDR is closed (precharged) after every access to the row is completed.
2. Posted CAS Operation: According to this policy, a read or a write command can be asserted immediately after every activate command to the DDR.
3. Bank interleaved Memory Mapping: When closed row policy is used for DDR operation, the

Figure 5.8: Bank Controller FSM

efficiency is increased using bank interleaved memory mapping based on JEDEC standards.

4. Bank level Parallelism: The inter-bank timing constraints are much smaller compared to intra-bank timing constraints. Hence, it is logical to have a bank level parallelism to send the commands to the DDR.

5.3.3.2 DDR controller State Machines

The main components of the DDR controller are its state machines. Every operation in the DDR takes specific amount of time depending on speed grade of the DDR module and its corresponding JEDEC standards. The read and write operations to the DDR involves charging and discharging storage capacitor which is not an instantaneous process. Hence, successive commands should be spaced well

enough before the start of each process. The controller FSMs schedule sequence of commands with minimum time separation to handle any read and write request. This design involves two type of FSMs that operate at different levels of granularity as described below.

Bank Controller FSM Every bank of the DDR has its own row buffer. Hence, multiple banks can be activated independent of each other. However, in doing so, there exists a minimum amount of time between the activation of any two banks specified by JEDEC standard. The delay between two ACT commands to two different banks is significantly lower when compared to the delay between two ACT commands to the same bank. Thus, if there are pending requests targeting different banks, they can be processed without waiting for the completion of previous command.

The bank controller is designed to take advantage of this nature of DRAM structure to improve the throughput. The state machine of bank controller is shown in Figure 5.8. It receives memory requests from request handler in the front-end and assigns them to respective bank FSMs. While doing so, it makes sure that the timing constraints between activate commands to different banks are not violated. A time delay of t_{RRD} clock cycles is given between successive bank activation. The bank controller handles refresh command requests from refresh control logic. The FSM accepts refresh command only if it is critical or no other memory requests are pending. The criticality of refresh commands is discussed later in this chapter.

Bank Specific FSM Serving read and write requests requires issuing sequence of commands to the DDR module. The target row has to be activated by sending ACT command (also known as Row Access Strobe) before reading from or writing into the row. The RD/WR command (also known as Column Access Strobe) follows ACT command which actually does read/write operation. The opened row is closed by sending precharge command once the read/write operation is complete. All these sequence of commands must be sent adhering to the timing constraints as per JEDEC standards.

The bank specific FSMs are dedicated to handle memory requests to their designated bank. The design of a bank FSM can be better explained with the help of Figure 5.9. It receives memory request from the bank controller and activates the target row of its bank. Read/write command is sent after RAS to CAS delay. The data handler is signaled to start the data transfer. The activated row is closed by sending precharge command (closed row policy). The bank FSM exerts back pressure on bank controller while it is handling any request by asserting a busy signal. This implies that when the bank FSM is busy, the arbiter would be still be buffering the requests in the FIFO without sending them into the bank FSM for generating the DDR specific commands. The number of bank FSMs is a configurable parameter and

Figure 5.9: Bank Specific FSM

it should be equal to the number of banks present in a DDR. All bank FSMs function independent of each other (Each FSM is a separate hardware implementation which executes in parallel).

Posted CAS As mentioned previously, the bank FSMs do not interact with each other. There will be a possibility that two bank FSMs are trying to send different commands to their respective banks at any given cycle. Consider an instance where one FSM is sending ACT and another is sending RD/WR command to their respective banks. However, the command bus can send a command only on one edge of the clock unlike the data bus. Thus, there is a collision on the command bus. The bank FSMs use posted CAS mechanism to overcome this scenario. An instance of the command bus conflict is shown in upper half of Figure 5.10. In a posted CAS operation, a CAS command (RD/WR) can be sent in the next clock cycle after RAS command (ACT). The CAS command is held by the DDR and executed after the additive latency. (DDR3 allows the read or write to be issued immediately after the ACT command. However, the read/write command is internally held by the device before actually issuing it. This time

Figure 5.10: Posted CAS and Normal CAS [3]

period is termed as additive latency.) This improves the command and data bus efficiency, and also simplifies the controller design.

5.3.3.3 Controller Policies

Refresh Control Strategy Every row of the DDR has to be refreshed once in 64ms for the charge in the cells to be retained which is termed as static refresh interval [3]. To refresh all rows within the allotted refresh period, the controller must issue auto refresh command at an average periodic refresh interval (t_{REFI}) as defined by the JEDEC standards [12]. The DDR will keep track of refreshed rows and this lowers the burden on controller. The controller's responsibility is to send refresh commands to DDR at every t_{REFI} . The average refresh interval is the static refresh interval divided by the total number of rows in the DDR.

For static refresh interval of 64ms, a DDR with 8192 rows has t_{REFI} of 7.81 μ s. Thus, the controller has to issue 1 refresh command in 7.81 μ s. However, to provide improved efficiency in scheduling, the JEDEC DDR3 standard allows postponing of maximum 8 refresh commands. The accumulated refresh commands can be issued successively.

Figure 5.11: Refresh Control Logic

The refresh strategy of this controller uses postponed refresh commands. Figure 5.11 shows the logic used in refresh policy. A modulo counter will keep track of total refresh intervals expired (pending refresh commands). The refresh control logic sends refresh request to the bank controller if there are any pending refresh commands. The request is accepted by the bank controller if there are no memory requests. Thus the bank controller can hold the refresh commands until the pending refresh commands reach 8. Once it reaches 8, the refresh logic asserts critical refresh signal and the bank controller must accept the refresh request. Thus the refresh control logic avoids interference from refresh cycles with majority of memory requests. However, this interference should be considered for worst case latency estimation.

Memory Mapping The memory address is divided into bank, row and column address (assuming single rank DDR module). The position of bits corresponding to these addresses decides memory mapping strategy. Figure 5.12 illustrates sequential and bank interleaved memory mapping. In the interleaved addressing scheme, sequential addresses are distributed across all the banks available in the DDR. As discussed previously, successive ACT commands to different banks can be issued with smaller delay.

Figure 5.12: Memory Mapping

Hence, interleaved memory mapping improves average sequential access time. This controller design uses bank interleaved memory mapping to improve sequential access performance. Memory map unit is responsible for decoding memory address and mapping it to the bank, row and column addresses.

Command Generation Figure 5.13 shows command generation flow. The command generation module receives DDR commands and asserts specified logic levels on the command bus. The commands and their signaling are tabulated in Figure 2.3. Command generator uses a lookup table to store these commands. Additionally, it drives address bus while sending RAS and CAS commands. Whenever there are no active requests, it sends NOP commands.

Data Handling The control path and data path are handled separately in this design. Data handler is responsible for reading write data from the front-end buffers and driving data bus while handling write request. Bank FSMs synchronize with CAS commands and data transfer using handshake signals to the data handler. The read data is sampled on data strobe signals and written into front-end response data buffer. The data handler notifies bank FSMs about completion of the data transfer so that they can issue precharge command.

Figure 5.13: Command Generation

Figure 5.14: Controller Front-end Implementation

Figure 5.15: Controller Core Implementation

5.4 Implementation Details

The architecture proposed in Section 5.3 is implemented using verilog HDL language and Xilinx Vivado Toolset [97]. The memory controller front-end, core and the back-end implementation block diagrams are shown in Figures 5.14, 5.15 and 5.16. As shown in Figure 5.14, the memory controller exposes the AXI3 interface to the PS, which in-turn is connected using the Xilinx Vivado toolset [97]. The front-end has 3 separate modules which are the arbiter (performing round-robin arbitration), the AXI slave module and the memory request handler unit. The AXI slave module implements the AXI protocol as specified by the AXI AMBA standards. Each requester has a read/write request buffer but the response buffer is shared. The request handler and memory map module is a combinational unit. The incoming DDR address is partitioned based on the DDR parameters and forwarded to the bank controller. The bank controller's busy parameter makes sure that the request is asserted from the front-end only when the busy signal is low.

The controller core implementation details are shown in Figure 5.15. The bank controller is a FSM that handles memory and refresh requests. Memory requests are always prioritized over refresh unless

Figure 5.16: Controller Back-end Implementation

Figure 5.17: Read Request from the Core to the DDR

the refresh is critical. The FSM holds the busy signal high until the completion of commands in the bank FSM.

The controller back-end implementation details are shown in Figure 5.16. This end implements the interface to the DDR. The command generator obtains its commands from the bank FSMs connected to it. The command generator multiplexes across all the banks present in the architecture. This in-turn drives the command and address lines of the DDR interface. The initialization logic is responsible to power up the DDR for the first time.

This entire controller unit is interfaced with a Xilinx PHY which is the physical interface to the DDR. On the other end, the AXI slave from the front-end connects to the GP port of the Zynq Platform to complete the proposed architecture implementation.

Figure 5.18: Memory Controller serving Read Request

Memory Read and Write Based on the architecture implementation, it is evident that a read or a write request from the core to the DDR requires it to adhere to the AXI protocol as the PS and PL portions of the architecture are connected through the AXI-GP port [96]. The timing diagram in Figure 5.17 shows the read request from the core to the DDR. A read is requested by asserting ARVALID signal with a valid address (ADDR) by the master. The slave acknowledges the request by asserting ARREADY for one cycle. After the ARREADY signal is asserted by the slave, the read request is transferred from the request buffer to the memory controller by the arbiter. The timing diagram in Figure 5.18 shows how the memory controller handles the read request internally. A Simple read operation with burst length (BL) 4 is shown in Figure 5.18. The memory mapping unit splits the valid address into row, column and bank addresses. The implemented controller will first send ACT command to the specific bank and row. The controller waits for t_{RCD} before sending the read command. The address bus holds valid column address during this command interval. The controller has to wait for t_{RL} for the first data to arrive. The data is synchronous with data strobe signal as can be seen from the waveform. The data arrives on both clock edges till the burst finishes. The controller then sends the precharge command upon receiving full burst of data. After the required number of data burst is received, the slave responds with data and asserts RLAST as shown in the Figure 5.17. The timing diagrams for a write request from the core to the DDR is shown in Figure 5.19 and Figure 5.20. The control signal using the AXI protocol is similar to that of the read request explained above except for a fact that the core places both the address and the data to be written on the AXI bus and waits for the last data to be written (WLAST) as its acknowledgment. The memory controller internally handles write request similar to read but with different predefined latencies as shown in Figure 5.20.

5.5 Analysis of worst-case latency of the implemented controller

This section analyzes the worst-case latencies (for a read and write request) of the implemented designs using the timing diagrams and the FSMs explained in the previous sections. Since, the DDR controller is implemented on the FPGA, it is possible to obtain a cycle accurate execution of the logic

Figure 5.19: Write Request from the Core to the DDR

Figure 5.20: Memory Controller serving Write Request

and also a bound on the maximum time that a request can take to complete. The worst-case execution time for this memory controller can be bounded using the JEDEC timing constraints and the design policy adopted and implemented for the memory controller. For analyzing a request from a particular core, the interference from all possible sources which include other cores (N cores) and refresh logic is taken into account. Refresh cycles are always considered in the worst-case latency computation for any request. The read-to-write and write-to-read switching times are subsumed by precharge time because closed row policy is used. The data transfer time which is referred to as (t_{BUS}) is subsumed by read-to-precharge delay. These equations for worst-case latencies assume a scenario where all the requests are targeted to the same bank so that the bank controller cannot exploit any bank level parallelism. However, bank level parallelism improves only the average case performance of the controller. The latency bound for a request when it arrives at the controller's slave end to the point in time it would be written into the DDR can be bounded as explained below. Let us define L_{write}^{wc} as the maximum time taken by the write request to be served from the point it enters the PL to the time the response is sent to the core. The different parameters that affect the latency of a write request are as follows.

1. The maximum time taken by the implemented memory controller to serve the memory request.

The maximum time taken by the implemented memory controller can be obtained from the write request timing diagram shown in Figure 5.20 using JEDEC standards. It is the summation of the time to serve the ACT command (t_{RCD}), WR command (t_{WL}), time to write the data burst (t_{BUS}), time to precharge (close row policy implemented) (t_{RP}) and the time to serve one refresh command in the worst-case (t_{RFC}).

$$t_{RCD} + t_{WL} + t_{BUS} + t_{WR} + t_{RP} + t_{RFC}$$

2. The time taken by the arbiter and the request buffer.

T_{buff} and T_{arb} are the fixed latency incurred in buffering the request in the front-end buffer and time taken to schedule the request in the arbiter respectively.

3. The time taken to serve N-1 requestors due to TDM arbitration.

In the worst-case scenario, there can be atleast one request pending from all N requestors and the time taken to serve these N-1 requestors can be bounded as follows.

$$(N - 1) * [t_{RCD} + t_{WL} + t_{BUS} + t_{WR} + t_{RP}]$$

This is the product of (N-1) number of requestors with the time taken to serve the pending request from each of the (N-1) requestor. The latency to write is independent for every request as the controller employs a closed-row policy. Note that based on the refresh implementation policy explained in Section 5.3.3.3, it is evident that there can be only one critical refresh which could occur in the worst-case while serving atleast one memory request from each of the N requestors (The critical refresh period is in micro-seconds and a single read or write is in nano seconds. Hence, many number of reads/writes can be completed within this critical refresh period. N needs to be an extremely large number for it to cross the critical refresh period which is not practical). Hence, t_{RFC} (refresh time) need not be taken into account for this computation.

Thus the worst-case latency for a write request to be served in the implemented architecture is the sum of the three enumerated points above.

Thus,

$$L_{write}^{wc} = (N - 1) * [t_{RCD} + t_{WL} + t_{BUS} + t_{WR} + t_{RP}] + T_{buff} + T_{arb} + t_{RCD} + t_{WL} + t_{BUS} + t_{WR} + t_{RP} + t_{RFC}$$

This can be computed in a similar mechanism for a read request. Let us define L_{read}^{wc} as the maximum time taken by the read request to be served from the point it enters the PL to the time the response is

sent to the core. The different parameters that affect the latency of a read request are as follows.

1. The maximum time taken by the implemented memory controller to serve the memory request.

The maximum time taken by the implemented memory controller can be obtained from the read request timing diagram shown in Figure 5.18 using JEDEC standards. It is the summation of the time to serve the ACT command, RD command, time to read the data burst, time to precharge (close row policy implemented) and the time to serve one refresh command in the worst-case.

$$t_{RCD} + t_{RL} + t_{RTP} + t_{RP} + t_{RFC}$$

2. The time taken by the arbiter and the request buffer.

T_{buff} and T_{arb} are the fixed latency incurred in buffering the request in the front-end buffer and time taken to schedule the request in the arbiter respectively.

3. The time taken to serve N-1 requestors due to TDM arbitration.

In the worst-case scenario, there can be atleast one request pending from all N requestors and the time taken to serve these N-1 requestors can be bounded as follows.

$$(N - 1) * [t_{RCD} + t_{WL} + t_{BUS} + t_{WR} + t_{RP}]$$

This is the product of (N-1) number of requestors with the time taken to serve the pending request from each of the (N-1) requestor. The latency to serve a write request is always greater than the latency to serve a read request (based on JEDEC standards [12]). Also, the latency to write is independent for every request as the controller employs a closed-row policy. Hence, it is assumed here that every pending request from the N-1 requestors is a write request to compute the worst-case latency. Note that based on the refresh implementation policy explained in Section 5.3.3.3, it is evident that there can be only one critical refresh which could occur in the worst-case while serving atleast one memory request from each of the N requestors. Hence, t_{RFC} (refresh time) need not be taken into account for this computation.

Thus the worst-case latency for a read request to be served in the implemented architecture is the sum of the three enumerated points above.

Thus,

$$L_{read}^{wc} = (N-1)*[t_{RCD}+t_{WL}+t_{BUS}+t_{WR}+t_{RP}]+T_{buff}+T_{arb}+t_{RCD}+t_{RL}+t_{RTP}+t_{RP}+t_{RFC}$$

Since the PL section of the architecture can be bounded accurately, the overall latency of the architecture will only depend on the time the request takes to reach the PL from each core which can

Figure 5.21: Experiment Setup

be bounded for the SoC on which the design is implemented, which provides a predictable execution architecture.

5.6 Test results and challenges in testing the architecture

Experiment Setup The primary aim of the experiments conducted is to validate the predictable execution of an application on the proposed architecture. To prove this, the best possible methodology is to repeatedly measure the time taken for individual read and write requests and show the consistency of the results for both single and dual core cases. This should be repeated for accessing the DDR in both the PS side and the PL side and measuring the time taken for the same. The memory requests to the PS DDR will serve as reference against which the proposed architecture will be compared.

The experiment setup is as shown in Figure 5.21. To test any design with a single ARM A9 core on Zynq 706, it can be seen from the Figure 5.21, that the zynq 706 can be connected to a host system which runs the Xilinx Vivado Design Suite and the SDK and further interface this host system to the zynq development board using a JTAG debugging interface. This debugger helps to access all the registers inside the ARM A9 core and further assist in accurately profiling every application executing on the single ARM A9 core.

However, it was discovered only during the course of the project that the Zynq platform does not permit the dual core experiments to be directly conducted using a setup similar to the single core. This implies that the Xilinx provided tool set allows tracing and debugging of only single core events. To

perform multi-core experiments, Xilinx provides a round-about mechanism [103]. The Zynq development board has an SD-card slot, which can store the boot loader and the multi-core application. As soon as the development board is powered ON, the boot loader executes the application as shown in the Figure 5.21. Thus, an in depth profiling and tracing of events is not possible with the Zynq development boards and hence, the application used for multi-core has timers which measure the time taken for individual memory requests.

Experiment: The application used for both single and dual core experiments is to send 100 memory requests (read and write requests separately) and measure the time taken (in clock cycles) for each memory request. For the dual core experiments, each core simultaneously sends 100 memory requests to the DDR and one of the cores (core 0) is profiled to obtain the time taken for each memory request. The obtained data is then plotted as a histogram to visualize the distribution of the measured execution times. The different experiments conducted on the architecture (shown in Figure 5.22) are to measure the time for individual read requests, write requests in both single and dual core mode to the PL DDR and compare them to the time for individual read and write requests to the PS DDR. The blue arrow represents the memory request to the DDR on the PS side (which represents a COTS multi-core) and the red arrow represents the memory request to the DDR on the PL side (which represents the proposed architecture).

Results: The results of the experiments are presented as histograms. For plotting the histogram, the bin size is kept as 10 cycles (+/- 5 cycles as a threshold for measurement error).

Single Core Read Requests:

The Figures 5.23, 5.24 represent the histogram for 100 read requests to the PS DDR (reference architecture) and 100 read requests to the PL DDR (proposed architecture). From the Histogram 5.23, it can be seen that the time to perform a single core read request varies between 74 clock cycles to 144 clock cycles. The time taken for a single read varies significantly for the PS DDR. On the other hand, the Histogram 5.28 presents the data for 100 read requests to the proposed architecture (PL DDR). It can be seen that the proposed architecture performs consistently by having most of its reads between 181-191 clock cycles. The outlying points represent the refresh related delays for the architecture (1 refresh for every 10 read requests approximately). Although a read to the proposed architecture takes twice the time as the reference architecture, it can be seen that it is predictable.

Figure 5.22: Experiments for the architecture

Figure 5.23: Memory Read Request to PS DDR - Single Core

Figure 5.24: Memory Read Request to PL DDR - Single Core

Single Core Write Requests:

The Figures 5.25, 5.26 represent the histogram for 100 write requests to the PS DDR (reference architecture) and 100 write requests to the PL DDR (proposed architecture). As seen in the histograms and as explained for single core read experiment, it is evident that the write request to the proposed architecture (PL DDR) is slower compared to the reference architecture (PS DDR). However, the writes to the proposed architecture is predictable. Of the 100 writes, it is seen that 91 of them reside between 95-105 cycles. The outliers are clearly the refresh related delays (1 refresh for every 10 write requests approximately).

Figure 5.25: Memory Write Request to PS DDR - Single Core

Figure 5.26: Memory Write Request to PL DDR - Single Core

Observations from Single core experiments It can be seen that the memory read/write requests to the proposed architecture (PL DDR) is predictable, meaning the time measured for the read and the write requests can be quantified. However, the memory read/write requests to the reference architecture (PS DDR) has a range of values which cannot be easily comprehended.

Another observation from these experiments is that, the memory requests to the PS DDR is approx-

imately twice as faster when compared to the reference architecture. This can be easily explained due to the lower frequency clock on the FPGA (200 MHz). This latency can be drastically improved as the frequency at which the FPGA is clocked improves.

Dual Core Read and Write Requests:

The Figures 5.27, 5.28, 5.29 and 5.30 represent the histogram for 100 read/write requests of the core 0 to the PS DDR(reference architecture) and 100 read/write requests of the core 1 to the PL DDR (proposed architecture) in the dual core setting where each core sends 100 memory requests to the DDRs simultaneously. As mentioned previously, the dual core experiments were conducted only using SD card and a boot loader. It can be observed that time taken for each memory request in both reference and the proposed architecture lies across a range of clock cycles. Although the time taken for the proposed architecture is more consistent than the reference architecture, the memory request time in dual core mode could not be measured precisely (using SD Card) which will be explained subsequently.

Figure 5.27: Memory Read Request to PS DDR - Dual Core

Figure 5.28: Memory Read Request to PL DDR - Dual Core

Figure 5.29: Memory Write Request to PS DDR - Dual Core

Figure 5.30: Memory Write Request to PL DDR - Dual Core

The reasons for imprecise measurements in dual core setting are as follows.

1. The interrupts from the peripherals which are connected to the PS and PL cannot be controlled while executing the platform in dual-core mode through SD card. Hence, there is always a possibility of an interrupt to occur while the experiment is executing which can affect the measurement.
2. The boot loader mechanism works in such a way that it executes the initial script on the core 0, eventually starts the core 1 and then uses core 0 as a master to activate the core 1 [103]. Hence, in any of the experiments, it is not possible to control the time in which both cores are active and also there exists no mechanism to halt the cores while they are executing using the SD card.
3. When the experiment is run using SD card, there is no direct visibility to all the register set (in real-time) inside the ARM cores which can be otherwise directly monitored while the experiment is performed using the SDK tools and Xilinx JTAG debuggers.
4. According to the document [103], it can be seen that the program in the SD card controls the functioning of the CPU cores. Specifically, core 0 is set as a master core. Hence, it is unknown how the core 0 affects the requests from cores 0 and 1 to be sent to the DDR.
5. Finally, the overhead of the control program executed by the boot loader in the SD card mode itself cannot be contained and hinders the measurement of precise read/write request timings.

The above points make it practically infeasible for any dual-core experiments to be conducted in the setting which is explicitly required for the case (2 cores executing in parallel and generating memory requests to the DDR which can cause contention). Although, the feasibility of the architecture could be proved by implementing it in the platform and also showing by means of equations that the PL side of the implementation is predictable, it could not be verified experimentally. However, the advantage of building designs on FPGA is that it is modular, portable and can be re-used. This means that the same architecture can be replicated when future versions of the platform with dual-core capability is released. Thus, the architecture proposed is generic and can be replicated on any hybrid COTS-FPGA platform with a DDR on its PL.

5.7 Conclusion

This chapter thus establishes the feasibility of an architecture which has COTS multi-core (ARM A9) and at the same time having its major challenge of unpredictability in the shared memory subsystem addressed with the proposed solution. The problem of shared memory subsystem has been addressed in a novel way by routing all the memory requests to the FPGA and designing the techniques to handle these memory requests in a predictable manner on the FPGA. Although the feasibility of the architecture is shown, techniques are required to benchmark and experimentally prove the predictability of the architecture. This can be done as the FPGA industry produces more advanced SoC architectures in the future. Also, the current SoC architectures are severely limited both on FPGA resources and the frequency at which they are clocked. This affects the throughput of the entire architecture. However, as the subsequent generations of the SoC are made available, the frequency of the design on the FPGA will definitely improve which will eventually improve the throughput drastically.

Chapter 6

Bank Allocation Algorithm for requestors

6.1 Motivation

The previous chapter had shown the feasibility of an architecture with COTS multi-core supported by an FPGA to be predictable by implementing the shared memory subsystem outside the multi-core. This chapter further focuses on one of the components of the memory sub-system namely the DDR memory. The DDR memory is the most commonly used memory type in multi-cores as it not only provides large amount of storage space at low cost per bit but also at a very efficient rate [104]. The architecture of the DDR is explained under Section 2.1.1 in Chapter 2.

Each core is a memory requestor which requests data of a specific size (request size) from the DDR. When multiple requestors share a DDR, the worst-case execution time for each requestor depends on the manner in which the DDR banks are accessed. Two methods have been specifically focused. First, when each requestor has banks allocated exclusively to itself (private bank allocation). Second, when all the banks are shared across the requestors (shared bank allocation).

The advantage of using private bank allocation is that it avoids interference between requestors. However, each requestor requires a separate set of banks which reduces DDR utilization. On the other hand shared bank allocation leads to interference between the requestors.

This chapter considers the problem of allocating banks of DDR to requestors such that their worst-case execution time requirement is satisfied. The main contributions are as follows: 1) Obtaining the latency of a requestor when its banks are privatized and an optimal algorithm to allocate banks to each requestor (optimal for worst-case latency). 2) An optimal bank allocation strategy when requestors share banks with one another (optimal for worst-case latency).

6.2 Background

In this section, a background of the existing work is presented based on which these algorithms have been derived.

6.2.1 DDR Memory Mapping

A background on DDR memory has already been presented under Section 2.1.1 in Chapter 2. A Double Data Rate memory (DDR) chip comprises a set of banks, each containing memory elements arranged as rows and columns (all DDR3 memories have 8 banks). The DDR and the processor are interfaced with a command, address, and data bus. The DDR works on the following set of memory commands - activate (ACT), read/write (R/W), precharge (PRE) and refresh (REF), each of which takes a predefined number of clock cycles as specified by JEDEC [105]. On every clock cycle one memory command is transferred to the DDR, while two data words are transferred from the DDR. Commands are issued to the DDR based on the timing constraints provided by the JEDEC standards [105].

A memory map decides the accurate place of the data in the memory and specifies how a request is split over banks. It is controlled by two parameters: Burst Count (BC) Bank Interleaving number (BI). Bank Interleaving specifies the number of banks that a requestor accesses. Burst Count is the number of R/W commands to be sent for each bank. For every request size, the value of burst count and bank interleaving are fixed and its product is the request size in memory words. It is to be noted that every requestor can have a different request size. The BI and BC of a requestor must be decided as a part of the system configuration whereas the BL value remains a constant for all requestors.

6.2.2 Memory controller Architecture

For the bank allocation algorithms the memory controller architecture provided by Yonghui is used *et al.* [5]. This controller is divided into *front-end* and *back-end* as shown in Figure 6.1. The former is responsible for handling requestor arbitration and the latter for memory command arbitration.

Front-end The front-end (see Figure 6.1) of the memory controller receives requests from the requestors. Each requestor has a separate request buffer and is assumed to have at most one pending request (the next request cannot be sent until the response of the current request is received). There is one response buffer per requestor which receives the data read from the memory. The front-end has a

Notation	Description
i	The number of a requestor arriving at the back-end. $i \geq 0$
R_i	The i^{th} requestor at back-end
RS_i	Size of requestor R_i
j	The first bank number for current requestor R_i
BI_i, BC_i	Bank Interleaving number and Burst Count of requestor R_i
b_j	Bank number of the j^{th} bank access
ACT_j	ACT command of j^{th} bank
$t(ACT_j)$	Time of scheduling ACT command to bank j
c_j	Command collision between ACT and higher priority command
$t(RW_j^k)$	Time of scheduling the RW command
PRE_j	Precharge command to bank j
$t(PRE_j)$	Time of scheduling the PRE command to bank j
$t_s(R_i)$	Starting time of requestor R_i
$t_f(R_i)$	Finishing time of requestor R_i
l	Index the banks of a request R_i and $\forall k \in [0, BI_i - 1]$
k	Index the burst count of bank for R_i and $\forall k \in [0, BC_i - 1]$

Table 6.1: Basic Notations

The back-end of the memory controller receives requests from the front-end. When a requestor R_i arrives at the back-end, the controller finds the memory map information in the lookup table and inserts it in the parameter queue. The command generator generates the memory commands and inserts into the command buffers corresponding to each bank. Next, the timers to track the JEDEC constraints are started. The commands at the head of the command buffers can be issued after these timers expire. Multiple command buffers can have valid commands. However, only one valid command is selected and issued by the command arbiter based on a dynamic command scheduling algorithm.

The command scheduling algorithm has the following properties. The current requestor's valid commands have higher priority than the subsequent requestors (in-order processing of requests). The valid R/W commands have higher priority than the valid ACT commands. Also, the banks with lower number has higher priority. The command scheduler sends a notification to the front-end when all the ACT commands of a current requestor has been issued. This notification is used by the front-end to send the subsequent request to the back-end. The controller follows a closed-row policy, meaning the DDR row is precharged after every access.

Figure 6.2: Timing Dependencies in Dynamic Scheduling

6.2.3 Formalization and Notations

Based on the background provided about the memory controller and the DDR command scheduling policies, this section also introduces the notations and definitions already defined in [5] to further extend the results for the algorithms. As previously mentioned, every memory request is converted into a set of bank interleaving number of bank accesses and every access to the bank in turn is further broken down to a set of burst count number of read or write accesses with the last access tagged with an auto precharge. The same notations are used as provided in [5] to extend the results for this work as shown in Table 6.1.

The timing dependencies for a dynamic command scheduling for the memory controller for a requestor is depicted in Figure 6.2. [5] analyzes the time for execution of a requestor by calculating the scheduling time of the DDR commands. Based on the Figure 6.2, it can be seen that the ACT command of a request to a bank can be delayed by a previously scheduled ACT command (t_{RRD}) or by the precharge command of the same bank (t_{RP}) or by the four bank activation timing constraint (t_{FAW}). Similarly, every other constraint can be derived from the Figure 6.2. Also, every ACT command can be blocked by a read or a write command with higher priority (as explained by the command scheduling policies in Section 6.2.2) which can postpone the ACT command by one cycle (depicted by a blue circle in Figure 6.2).

Let $t_a(R_i)$ be defined as the time at which R_i arrives at the back-end. When a request arrives at the back-end, it is executed by scheduling the commands starting from bank b_j . j denotes the number of the first bank access of R_i and it is a function of i ($j(i) = \sum_{k=1}^{i-1} BI_k$). The finishing time of R_i is defined

as the time at which the last RD or WR of the request is scheduled. The starting time of R_i is defined as the earliest when the commands are scheduled by the arbiter. This could be either two clock cycles immediately after the arrival time or one cycle immediately after the finish time.

Thus using the Figure 6.2 and the DDR command scheduling policies, the scheduling time of ACT, RW and PRE commands can be defined as follows.

ACT Command: The scheduling of the ACT command depends on three JEDEC constraints which are depicted in Figure 6.2 (t_{RRD} , t_{FAW} , t_{RP}). Also, as explained previously in the dynamic command scheduling, the ACT command can incur a command collision due to the higher priority commands which will be represented as $c(j+l)$. Note that $c(j+l)$ can take either 0 or 1 clock cycle based on the existence of a command collision.

$$t(ACT_{j+l}) = \max\{t(ACT_{j+l-1}) + t_{RRD}, t(PRE_m) + t_{RP}, t(ACT_{j+l-4}) + t_{FAW}, t_a(R_i) + 2\} + c(j+l) \quad (6.1)$$

where, $t(ACT_{j+l-1})$ represents the scheduling time of an ACT command scheduled before the current ACT command and $t(ACT_{j+l-4})$ represents the scheduling time of the fourth ACT command prior to the one currently scheduled.

RW Command: The RW command scheduling of the current requestor depends on its previous RW command or the ACT command to the same bank as shown in Figure 6.2.

$$t(RW_{j+l}^0) = \max(t(RW_{j+l-1}^{BC_i-1}) + t_{SWITCH}, t(ACT_{j+l}) + t_{RCD}) \quad (6.2)$$

where $t(RW_{j+l}^0)$ represents the first RW command of the current requestor.

PRE command: The PRE command is scheduled based on the ACT command of the same bank or the last RW command to the bank.

$$t(PRE_{j+l}) = \max(t(ACT_{j+l}) + t_{RAS}, t(RW_{j+l}^{BC_i-1}) + t_{RWTP}) \quad (6.3)$$

Using these equations and As Late As Possible (ALAP) scheduling mechanism for the commands of the requestor which was scheduled prior to the current requestor, is used to derive the worst-case latency for the current requestor when the requestors share banks (Reference [5] has derived this case). The

intuition behind using ALAP scheduling for a requestor prior to the current requestor is that it provides the latest time at which the DDR commands of the current requestor can be scheduled. It is computed by calculating backwards from the last read/write for the requestor scheduled prior to the current and assigning the minimum possible time between the DDR commands while going back in time from the last read/write. This implies that the time between scheduling the DDR commands must be minimum while also satisfying the JEDEC constraints. Let us understand this with a simple illustration. As it has already been stated in Table 2.3, the minimum time between two read or write commands is t_{CCD} . As we know that the read/write commands to a single bank are always sent sequentially, the minimum time between the first read/write commands to the successive banks would be $BC_{i-1} \times t_{CCD}$. We also know from Table 2.3, the minimum time between an ACT and RD/WR command is t_{RCD} . Further, the minimum time between two ACT commands to different banks is t_{RRD} . Thus, calculating backwards the minimum time interval between two success ACT commands would be $\max(t_{RRD}, BC_{i-1} \times t_{CCD})$.

Also, it has already been proved in [5] that the first ACT of the current requestor does not depend on the Four Bank Activation Window (t_{FAW}) constraint ([105] and lemma 3 in reference [5]). The idea behind this proof is that the first ACT of the current requestor is farther away than the t_{FAW} timing when the requestor prior to the current one is scheduled using ALAP scheduling. This implies that the JEDEC constraints between the ACT command of the current requestor to the previous ACT command in the requestor scheduled prior to the current is always greater than t_{FAW} as the memory controller follows a closed row policy. Using the worst-case starting time for the current requestor, the time at which the ACT, RD/WR and PRE commands of the current requestor is scheduled is computed. This equation is then substituted in the worst-case finishing time equation of the current requestor which is defined as follows.

$$t_f(R_i) = \max_{0 \leq l \leq BI_i - 1} (t_f(R_{i-1}) + t_{SWITCH} + [(BI_i \times BC_i) - 1] \times t_{CCD}, t(ACT_{j+l}) + t_{RCD} + [((BI_i - l)BC_i) - 1]t_{CCD}) \quad (6.4)$$

Upon closer inspection of the equation 6.4, it can be understood that the equation represents two possible execution latency time periods which is evident from the Figure 6.2. The first part of the max term represents the scenario where there is a switch between the requestor type (read to write or write to read) between the current and the previous requestor. In this case, as shown in Figure 6.2 after the last read/write of the requestor scheduled prior to the current requestor is sent to the DDR, the t_{SWITCH} timing delay takes effect and it then leads to the first read/write of the current requestor.

The second part of the max term represents the scenario where the latency of the current requestor depends on the first ACT of the current requestor (ACT_j). Substituting the worst-case starting time of the ACT command into this equation and iterating the equation obtained from the ALAP scheduling and expanding it for the number of banks for the current requestor (0 to BI_{i-1}) will give the worst-case latency for the current requestor. As a simple illustration, let us use the Table 2.3 to compute the values in the equation 6.4. The value of t_{SWITCH} would be maximum when there is read that follows a write request ($t_{SWITCH} = t_{WL} + BL/2 + t_{WTR}$). Based on JEDEC standards, it is a known fact that when the memory bus switches from a write to a read request, it incurs the maximum delay for the subsequent request. Thus, substituting the values from the JEDEC standards provided in Table 2.3 in equation 6.4, the finishing time of a requestor is obtained to be the maximum of $[(t_f(R_{i-1}) + 22 + 4((BI_i \times BC_i) - 1), t(ACT_{j+l}) + 8 + 4((BI_i \times BC_i) - 4l(BC_i) - 4)]$. If the value of l was 0, then we know for a fact that the first ACT of current requestor would be scheduled before the last RD/WR of the previous requestor. Hence, it can be safely assumed that $t(ACT_{j+0})$ to be less than $t_f(R_{i-1})$. Thus, the first term would be the maximum of the two terms as the value of t_{SWITCH} is 22. If there was no switching between the reads and writes, then either of the two terms could be the maximum.

6.3 Model for Requestors, DDR and Bank Allocation

Requestors. Let $R = \{R_1, R_2, \dots, R_n\}$ represent a set of requestors with a worst-case execution time requirement L_{wc} . Each requestor R_i is represented by the tuple (BI_i, BC_i, RS_i, l_i) , where

- BI_i represents the bank interleaving number of a requestor,
- BC_i represents the burst count for the requestor,
- RS_i represents the request size and
- l_i represents the latency of the requestor.

The worst-case latency (l_{wc}) is defined as the sum of the latencies of all the requestors ($l_{wc} = \sum_{i=1}^n l_i$).

DDR Model. Let BI_{max} represent the maximum available banks and BC_{max} represent the maximum row size of a bank. In all the discussions henceforth, it is assumed that the minimum value of BI_{max} is at least 8 [105]. The request size RS_i is defined as the product of BI_i, BC_i, BL and IW , where BL and IW represent the burst length and the interface width respectively. The BL and IW are

constants for a given DDR. In all the following discussions, it is assumed that $RS_i = BI_i \times BC_i$. It is also assumed that all RS_i values are powers of 2 (i.e., 1, 2, 4, 8 ..). This ensures that all the request sizes right from $RS_i = 1$ have been considered. Any value of $BL \times IW$ will increment the value of all RS_i by a constant.

Bank Allocation. Bank allocation to a requestor can be performed in two ways, either private or shared. For private, the bank allocated to one requestor cannot be reallocated to another requestor. For shared banks, at least one of the banks of the requestor is shared with its previous requestor.

The problem focused in this chapter is the allocation of BI_i and BC_i to each requestor such that the resulting latencies satisfy the requirement $l_{wc} \leq L_{wc}$.

6.4 Private bank allocation

This section discusses the latency equation derivation when the banks are privatized and proposes an optimal algorithm to allocate banks to requestors.

Problem Statement Given a set of requestors $\{R_i\}$ and the required JEDEC constraints, we need to allocate BI_i such that $\sum_{i=1}^n BI_i \leq BI_{max}$ and the worst-case latency requirement L_{wc} of the requestors is satisfied. Also, for any two requestors, R_i and R'_i , $B_i \cap B'_i = \emptyset$ when the banks are privatized.

6.4.1 Latency Equation for private bank allocation

To derive the latency (l_i) of a requestor in the back-end of a memory controller when the banks of a requestor are privatized, the derivation in [5] is closely followed with minor modifications which is explained below.

Privatized banks When the banks of R_i are privatized, the set of banks allocated are used only by R_i . As mentioned earlier, the first ACT command of R_i depends on 3 factors (t_{RRD} , t_{RP} , t_{FAW}). When the banks are private to R_i , the bank b_i will not be used by any other requestor. Hence, there will not be any existing PRE command to delay the issuing of ACT (refer to the Figure 6.2). Also, the first ACT command does not depend on the four bank activation window (t_{FAW}) based on the JEDEC parameters as explained under Section 6.2. Hence, the first ACT command of requestor R_i depends only on its previous ACT (t_{RRD} cycles). This is the difference between the private bank requestor latency and the

shared bank requestor latency explained in the Section 6.2 and derived in [5].

Based on the properties of the command scheduling mechanism explained in Section 6.2, a valid R/W command always has a higher priority over ACT. This causes the ACT command collision. Let $c(j+l)$ represent this command collision. The time at which an ACT command is scheduled $t(AC T_{j+l})$ which was explained in Equation 6.1, now reduces to Equation 6.5 for privatized banks.

$$t(AC T_{j+l}) = \max\{t(AC T_{j+l-1}) + t_{RRD}\} + c(j+l) \quad (6.5)$$

Finishing Time The finishing time of a request is the maximum of two scenarios as explained in the Equation 6.4.

$$\begin{aligned} t_f(R_i) = \max_{0 \leq l \leq BI_i - 1} & (t_f(R_{i-1}) + t_{SWITCH} + [(BI_i \cdot BC_i) - 1] \\ & \times t_{CCD}, t(AC T_{j+l}) + t_{RCD} + [(BI_i - l)BC_i - 1]t_{CCD}) \end{aligned} \quad (6.6)$$

We know that the equation (6.5) depends only on $t(AC T_{j+l-1})$ and t_{RRD} . By iteratively using the equation (6.5) to find the scheduling time of all the ACT commands, we obtain $t(AC T_{j+l})$ as follows.

$$\begin{aligned} t(AC T_{j+l}) = & \{t(AC T_{j-1}) + (l+1) \times t_{RRD} \\ & + \sum_{h=0}^l c(j+h)\} \end{aligned} \quad (6.7)$$

The scheduling time of $t(AC T_{j-1})$ is determined as follows. $t(AC T_{j-1})$ represents the last ACT of the requestor R_{i-1} . If $t_s(R_i)$ is the starting time of requestor R_i , then going backwards in the command scheduling time-line by scheduling each command as late as possible, $t(AC T_{j-1})$ should have been scheduled at least $t_{RCD} + ((BC_{i-1} - 1) \times t_{CCD})$ cycles before $t_s(R_i)$ (refer to the explanation of ALAP scheduling in Section 6.2).

$$t(AC T_{j-1}) = t_s(R_i) - 1 - t_{RCD} - (BC_{i-1} - 1) \times t_{CCD} \quad (6.8)$$

Latency (l_i) To derive l_i , the equation derived for the scheduling time for the ACT commands should be maximized based on equation (6.6) and assuming a collision for every ACT command. Substituting equation (6.8) and (6.7) in (6.6). We know that $t_f(R_{i-1}) = t_s(R_i) - 1$. Also, keeping the approach conservative, it is assumed that every ACT command will face a command collision from every other

command, $c(j + h) = 1$ and hence $\sum_{h=0}^l c(j + h) = l + 1$.

Substituting the above in equation 6.6.

$$\begin{aligned}
 t_f(R_i) = & \max_{0 \leq l \leq BI_i - 1} (t_s(R_i) - 1 + t_{SWITCH} + \\
 & [(BI_i \times BC_i) - 1] \times t_{CCD}, (t_s(R_i) - 1 - (BC_{i-1} - 1) \times \\
 & t_{CCD} + (l + 1) \times t_{RRD} + [(BI_i - l) \times BC_i - 1] \\
 & \times t_{CCD} + l + 1)
 \end{aligned} \tag{6.9}$$

We know that $l_i = t_f(R_i) - t_s(R_i) + 1$. From the above equation, moving $t_s(R_i) - 1$ to the left hand side, and then substituting the limits $l = 0$ and $l = BI_i - 1$ and further reducing the latency equation is obtained as follows.

$$\begin{aligned}
 l_i = & \max(t_{SWITCH} + (BI_i \times BC_i - 1) \times t_{CCD}, \\
 & (BI_i \times BC_i - BC_{i-1}) \times t_{CCD} + (BI_i - 1) \times \\
 & (t_{RRD} + 1 - BC_i \times t_{CCD}) + t_{RRD} + 1)
 \end{aligned} \tag{6.10}$$

6.4.2 Algorithm to Allocate Private banks to Requestors

To perform private bank allocation, given a set of requestors, the latency equation (6.10) is used. The latency equation comprises two terms which will henceforth be referred as Term 1 ($T1$) and Term 2 ($T2$). The parameters in the terms are BI_i , BC_i , BC_{i-1} and JEDEC constraints. The terms BI_i and BC_i are related as follows $RS_i = BI_i \times BC_i$. l_i is represented as follows.

$$l_i = \max(T1, T2) \tag{6.11}$$

where,

$$T1 = (t_{RRD} + 1)BI_i + t_{CCD}(BC_i - BC_{i-1}),$$

$$T2 = t_{SWITCH} + (t_{CCD} \times (BI_i \times BC_i - 1)).$$

Observations:

The following are the observations on $T1$ and $T2$.

1. For a given request size RS_i and JEDEC parameters, $T2$ is a straight line. This is because $BI_i \times BC_i$, t_{SWITCH} and t_{CCD} are constants for a given DDR and its JEDEC standards.

2. $\forall BI_i < RS_i$ and $\forall BC_{i-1}$, $T1$ is always less than $T2$.

As a next step, two lemmas will be stated. The aim of lemmas 1 and 2 is to show that for any RS_i , $T2$ is the least achievable latency for any requestor. For any requestor R_i , the maximum number of banks that can be allocated will be when $BC_i = 1$, which implies $BI_i = RS_i$. Observing $T2$, it is evident that for a given request size RS_i and for a set of JEDEC parameters, $T2$ is always a straight line. This implies that the least latency that can be achieved for any given request size will be equal to $T2$, meaning the latency cannot be reduced beyond $T2$. For a given request size RS_i , the maximum number of banks that can be allocated is equal to RS_i (when $BC_i = 1$). Hence, using lemma 1, it is shown that $T1$ and $T2$ can intersect at most once and using lemma 2, it is shown that if the terms $T1$ and $T2$ intersect, it happens between $RS_i/2$ and RS_i . Thus, using these lemmas it will be shown that $T2 > T1$ in the region $BI_i \in [1, RS_i)$.

Lemma 1 $\forall BC_{i-1}$ & $\forall BI_i$, $T1$ and $T2$ can intersect at most once.

Proof: To show that $T1$ and $T2$ can intersect at most once, the two terms are equated and substituted in $BC_i = RS_i/BI_i$ to find the intersection points as follows.

$$(t_{RRD} + 1)BI_i^2 + (t_{CCD} - t_{CCD} \times RS_i - t_{CCD} \times BC_{i-1} - t_{SWITCH})BI_i + t_{CCD} \times RS_i = 0 \quad (6.12)$$

Solving for BI_i , we have

$$BI_i = [-(b) \pm \sqrt{(b)^2 - 4(t_{RRD} + 1)RS_i}]/2(t_{RRD} + 1).$$

$$\text{where, } b = t_{CCD} - (t_{CCD} \times RS_i) - (t_{CCD} \times BC_{i-1}) - t_{SWITCH}.$$

Let,

$$Root1 = [-(b) - \sqrt{(b)^2 - 4(t_{RRD} + 1)RS_i}]/2(t_{RRD} + 1),$$

$$Root2 = [-(b) + \sqrt{(b)^2 - 4(t_{RRD} + 1)RS_i}]/2(t_{RRD} + 1).$$

Next step is to show that one of the intersection points ($Root1$) is always less than 1 by contradiction. For the intersection point to be greater than or equal to 1, numerator($Root1$) $\geq$ denominator. We also know that $t_{CCD} = 4$, $4 \leq t_{RRD} \leq 7$ and $t_{SWITCH} \geq 11$ according to JEDEC standards [105].

Assumption: Let, numerator($Root1$) $\geq$ denominator

$$\Rightarrow -(b) - \sqrt{(b)^2 - 4(t_{RRD} + 1)RS_i} \geq 2(t_{RRD} + 1)$$

$$\Rightarrow BC_{i-1} \leq (5 + t_{RRD} - t_{SWITCH})/4$$

For the assumption to be true given the values of t_{CCD} , t_{RRD} and t_{SWITCH} , $BC_{i-1} < 1$ is required. On the contrary it is known that $BC_{i-1} \geq 1$. Hence, it can be concluded that Root1 is always less than 1.

Thus, $\forall BC_{i-1}$ & $\forall BI_i$, T1 and T2 can intersect at most once and if it intersects the intersection point is Root2. ■

The next lemma will show that the root which exists, lie between RS_i and $RS_i/2$.

Lemma 2 $\forall BI_i < RS_i$ and $\forall BC_{i-1}$, $T1 < T2$.

Proof: To prove that $\forall BI_i < RS_i$ and $\forall BC_{i-1}$, $T1 < T2$, it will be first shown that $T2 > T1$ for BI_i value which is immediately less than RS_i .

Let us now consider the value of BI_i immediately less than RS_i , which is $BI_i = RS_i/2$ and the corresponding $BC_i = 2$. This is because the request sizes (RS_i) are assumed to be in powers of 2. Thus, if the largest $BI_i = RS_i$, then the subsequent request size will be $BI_i = RS_i/2$.

Substituting $BI_i = RS_i/2$, $BC_i = 2$ in equation 6.11,

$$T1 = (t_{RRD} + 1)RS_i/2 + 8 - 4BC_{i-1},$$

$$T2 = t_{SWITCH} - 4 + 4RS_i$$

Given $t_{CCD} = 4$, $4 \leq t_{RRD} \leq 7$ and $t_{SWITCH} \geq 11$, for $T1$ to be greater than $T2$, BC_{i-1} should be a negative value which is not feasible. Thus, $\forall BI_i < RS_i$, $T2 > T1$.

Also from Lemma 1, we know that T1 and T2 can intersect at most once. If at $BI_i = RS_i/2$, $T2 > T1$, then the intersection point (Root2) must be between $RS_i/2$ and RS_i . ■

Algorithm Description Given the inputs R , L_{wc} , BI_{max} , BC_{max} and JEDEC standards, the Algorithm 6.4.1 allocates private banks to requestors such that the worst-case latency (l_{wc}) is minimum.

The Line 3 of algorithm 6.4.1 initially computes the minimum number of banks required by each requestor (minimum BI_i) based on the formula $\lceil RS_i/BC_{max} \rceil$. In Line 4, the algorithm computes the sum of the minimum number of banks required by all the requestors. For any allocation to be feasible, this sum should be less than the total number of banks available for allocation. In line 5, the $T2$ term of the latency equation 6.11 is computed for each requestor. For any allocation to be feasible, $l_{wc} \leq L_{wc}$.

Algorithm 6.4.1 Private bank allocation strategy

```

1 Input:  $R, L_{wc}$ , JEDEC parameters
2 Output:  $\{BI_i\}$ 
3 foreach  $R_i \in R$  do
  | compute  $BI_i = \lceil RS_i / BC_{max} \rceil$ 
  end
4 If  $\sum_{i=1}^n BI_i > BI_{max}$ , declare FAILURE
5 foreach  $R_i \in R$  do
  | compute  $l_i = t_{SWITCH} - 4 + 4 \times RS_i$ 
  end
6 Compute  $l_{wc} = \sum_{i=1}^n l_i$ 
7 If  $l_{wc} > L_{wc}$ , then declare FAILURE
8 return  $\{BI_i\}$ 

```

The algorithm returns the computed $\{BI_i\}$ values if the condition in Line 4 is not violated and the condition in Line 7 is passed.

Algorithm Correctness The algorithm computes the minimum number of banks required by each requestor in Line 3. It returns a failure when the minimum number of banks required for all the requestors exceeds BI_{max} . It is a fact that when banks are private, the maximum number of banks that can be allocated should be less than the available number of banks in the DDR model considered.

From Lemmas 1 and 2 we know that $T2$ is the minimum achievable latency for any requestor and the minimum value of BI_i computed in Line 3 of the algorithm itself returns a latency equal to $T2$. This implies that the worst-case latency $l_{wc} = \sum_{i=1}^n l_i$ will also be the minimum worst-case latency. Hence, if $l_{wc} > L_{wc}$, there cannot be any feasible allocation. Thus, the algorithm does not return any BI_i allocation if the above mentioned condition is true.

Algorithm Complexity The algorithm 6.4.1 has a linear complexity. The Line 3 of the Algorithm 6.4.1 has a complexity of $O(n)$ in the number of requestors. Similarly the Lines 4 and 5 also have a complexity of $O(n)$ in the number of requestors. Hence, the algorithm has a linear complexity.

The optimality of Algorithm 6.4.1 will be proven in the following theorem. The aim of proposing this algorithm is to find a bank allocation, if there exists one. Thus, the proposed algorithm is optimal, meaning the algorithm always returns feasible bank allocation with minimum worst-case latency.

Theorem 1 *The bank allocation strategy in Algorithm 6.4.1 is optimal for worst-case latency.*

Proof: To prove the optimality of the Algorithm 6.4.1, three separate cases are considered. In the

first case, it is shown by contradiction that, if algorithm 6.4.1 does not return a feasible assignment, then no other algorithm can provide an assignment. In the second case it is shown that if any algorithm provides a feasible assignment, then algorithm 6.4.1 also returns an assignment. In the final case it is shown that the algorithm 6.4.1 when providing an assignment, always returns the minimum latency.

Case 1: Assume that the Algorithm 6.4.1 does not return a feasible assignment and another algorithm A returns a feasible assignment.

From Line 3 of Algorithm 6.4.1, we know that the algorithm always assigns the lowest BI_i value to each requestor. This implies that the algorithm A should have returned a BI_i value which is higher than the one provided by algorithm 6.4.1. From Lemmas 1 and 2, l_i is equal to $T2$ when BI_i is allocated based on Line 3. This will return a minimum l_{wc} value. If there is no feasible allocation for this value of BI_i , then there will not be a feasible assignment for any higher value of BI_i . Hence, the assumption that the other algorithm provides an assignment is contradictory.

Case 2: If an Algorithm A can provide a feasible bank allocation, then Algorithm 1 can also find a feasible allocation.

If algorithm A provides a feasible bank assignment, it implies that the algorithm must have assigned a BI_i value higher than algorithm 6.4.1. From Lemma 1, it is clear that Algorithm 1 returns the minimum BI_i value for every requestor. From Lemmas 1 and 2, it is clear that assigning any higher BI_i value provides a latency equal to $T2$, then all the feasible values of BI_i which are lower than the assignment must also provide the same latency. Hence, Algorithm 6.4.1 will also return a feasible bank assignment.

Case 3: Algorithm returns the minimum achievable latency.

From Lemmas 1 and 2 we know that $T2$ is the minimum achievable latency for any requestor and the minimum value of BI_i computed in Line 3 of the algorithm itself returns a latency equal to $T2$. This implies that the worst-case latency will be the minimum achievable latency.

Hence, the algorithm 6.4.1 is optimal for worst-case latency. ■

6.5 Shared bank allocation

This section will discuss the latency equation when at least one of the banks of the requestor is shared with its predecessor requestor and then propose an optimal algorithm to allocate banks to requestors.

Problem Statement Given a set of requestors $\{R_i\}$ and the required JEDEC constraints, we need to allocate BI_i such that $BI_i \leq BI_{max}$ and the worst-case execution time requirement L_{wc} of the requestors is satisfied.

6.5.1 Latency Equation for shared bank allocation

The latency (l_i) of a requestor in the back-end of a memory controller when at least one of its allocated bank is shared with its previous requestor is provided in Theorem 1 under Section 9 in reference [5]. This result has been derived in [5] by extending the derivation for shared bank allocation without the assumptions that were made for private banks explained under Section 6.4.

This latency equation is a maximum of 6 terms.

$$l_i = \max(T1, T2, T3, T4, T5, T6) \quad (6.13)$$

where,

$$T1 = (BC_i - BC_{i-1}) \times t_{CCD} + BI_i \times (t_{RRD} + 1)$$

$$T2 = t_{RWTP} + t_{RP} + t_{RCD} + [BI_i \times BC_i - 1 - (\min(BI_{i-1}, BI_i) - 1) \times BC_{i-1}] \times t_{CCD} + 1$$

$$T3 = t_{RWTP} + t_{RP} + t_{RCD} + [(BI_i - (\min(BI_{i-1}, BI_i) - 1)) \times BC_i - 1] \times t_{CCD} + 1$$

$$T4 = t_{RWTP} + t_{RP} + t_{RCD} + (BI_i - 1) \times (t_{RRD} + 1) + 1 + [BC_i - 1 - (\min(BI_{i-1}, BI_i) - 1) \times BC_{i-1}] \times t_{CCD}$$

$$T5 = t_{RWTP} + t_{RP} + t_{RCD} + (BC_i - 1) \times t_{CCD} + [BI_i - \min(BI_{i-1}, BI_i)] \times (t_{RRD} + 1) + 1$$

$$T6 = t_{SWITCH} + (BI_i \times BC_i - 1) \times t_{CCD}$$

The first and the last terms (T1 and T6) represent a possible dynamic command scheduling scenario where the DDR commands are perfectly pipelined without any precharge delay between the requests. By perfectly pipelined, it means that the DDR are commands of the previous and the current requestor are sent without incurring any additional delay other than the waiting time for the JEDEC constraints. This implies that the current requestor and the previous requestor might not have any banks in common, which could avoid the current requestor from waiting for its bank to get precharged (opened and used by the previous requestor). Specifically, the term T1 denotes the latency of a request when the DDR commands of successive requests do not share any bank and the request type is also the same (meaning

the requests could be successive reads or writes). On the other hand, the term T6 denotes the latency of a request when the DDR commands of successive requests do not share any bank and the request type varies between the successive requests (meaning a read followed by a write or a write followed by a read). In this case, the switching time of the bus is taken into account. The terms T2 to T5 represent the possible command scheduling scenarios where the successive requests share banks (common banks). Thus these 4 terms have the time to precharge and the time to activate accounted ($t_{RWTTP} + t_{RP} + t_{RCD}$).

6.5.2 Shared bank Allocation

Given a set of requestors and DDR model with JEDEC parameters, the shared bank allocation strategy is given in Algorithm 6.5.1.

Algorithm 6.5.1 Shared bank allocation strategy

```

1 Input: R,  $L_{wc}$ , JEDEC parameters
2 Output:  $\{BI_i\}$ 
3 Sort R in descending order of the  $RS_i$ .
4 foreach  $R_i \in R$  do
    | If  $RS_i \geq 8$ ,  $BI_i = \min(RS_i/2, BI_{max})$ 
    | If  $RS_i = 4$  &  $RS_{i-1} > 4$ ,  $BI_i = 4$ 
    | If  $RS_i = 4$  &  $RS_{i-1} \leq 4$ ,  $BI_i = 2$ 
    | If  $RS_i = 2$  &  $RS_{i-1} \geq 4$ ,  $BI_i = 2$ 
    | If  $RS_i = 2$  &  $RS_{i-1} < 2$ ,  $BI_i = 1$ 
    | If  $RS_i = 2$  &  $RS_{i-1} = 2$ ,  $BI_i = BI_{i-1}$ 
    | If  $RS_i = 1$   $BI_i = 1$ 
    end
5 Compute  $l_{wc} = \sum_{i=1}^n l_i$ 
6 if  $l_{wc} > L_{wc}$  then
    | declare FAILURE
    else
    | return  $\{BI_i\}$ 
    end

```

Algorithm Description The algorithm inputs a set of requestors, JEDEC parameters of the DDR and the worst-case latency requirement (L_{wc}). Given these inputs, Algorithm 6.5.1 allocates banks to requestors such that the worst-case latency (l_{wc}) is minimum. Line 3 of the Algorithm 6.5.1 sorts the given requestors based on the request size (RS_i). In line 4, the algorithm allocates the number of banks based on RS_i and RS_{i-1} . In Line 5, the algorithm computes l_{wc} . If $l_{wc} < L_{wc}$, then $\{BI_i\}$ allocation is returned.

Algorithm Correctness We know that the value of BI_{max} is at least 8.

- For request size greater than 8, Line 4 of the algorithm allocates a $\min(RS_i/2, BI_{max})$.
- For request size less than 8, Line 4 allocates BI_i values which are less than 8.

Hence, only a feasible BI_i is allocated.

In Line 5, the Algorithm 6.5.1 computes l_{wc} . In Line 6, the algorithm checks if $l_{wc} < L_{wc}$ and returns an allocation only if it is true. This ensures that the algorithm always returns a correct bank allocation.

Algorithm Complexity The complexity of Algorithm 6.5.1 is $O(n(\log(n)))$. This is because of the following reasons.

- The sorting of requestors in Line 3 can be performed in $n(\log(n))$ time.
- The remaining computations can be performed in linear time.

Hence, the complexity of the algorithm is $n(\log(n))$.

Overview to Prove the Optimality The optimality of the algorithm is proved in 2 steps. By optimal it means that the algorithm always returns an allocation (if it exists) which minimizes the worst-case latency. Since the algorithm sorts the requestors in descending order, let us define an inversion between two requestors as the following condition: $RS_i > RS_{i-1}$. For example in a set of 6 requestors, if requestor 2 has request size 8 and requestor 3 has request size 16, this condition is defined as an inversion.

Step 1: As a first step it is shown that for any given requestor R_i of size RS_i , if the predecessor request size RS_{i-1} is greater than or equal to RS_i (i.e., $RS_{i-1} \geq RS_i$), then its latency l_i is lower compared to if RS_{i-1} is less than RS_i . This helps to show that, given a set of requestors with multiple *inversions* (defined as the predecessor request size strictly lower than current request size), any swap between such inverted requestors would either result in latency reduction for requestor RS_i or remains the same. This will also result in the requestors being sorted in the descending order of their request size.

Step 2: If the requestors are sorted in the descending order, the first requestor (R_1) will have a predecessor (R_n) with a smaller¹ request size (i.e., $RS_1 > RS_n$). Next it is shown that, any further

¹If the first and the last requestor have the same request size, it implies that there are requestors with only one request size. This case is trivial to prove and hence we only consider the case where $RS_1 > RS_n$.

y/x	≥ 8	4	2	1
≥ 8	$12+4y$	23+4y when $RS_{i-1}=x$ 27+4y when $RS_{i-1}=x/2$	31+4y when $RS_{i-1}=x$ 35+4y when $RS_{i-1}=x/2$	$35+4y$
4	39	39 when $RS_{i-1}=x$ 43 when $RS_{i-1}=x/2$	47 when $RS_{i-1}=x$ 51 when $RS_{i-1}=x/2$	51
2	39	39 when $RS_{i-1}=x$ 42 when $RS_{i-1}=x/2$	42 when $RS_{i-1}=x$ 43 when $RS_{i-1}=x/2$	46
1	39	39	39	39

Table 6.2: Latency of requestors under different scenarios

swap in the requestors to reduce the latency of the first requestor will result in an equivalent or a higher worst-case latency. This implies that sorting the requestors in descending order will result in the least worst-case latency.

To prove steps 1 and 2, the latency of current requestor for all possible predecessor request size is computed as shown in Table 6.2. Requestors are grouped into four categories of request size: $RS_i \geq 8$, $RS_i = 4$, $RS_i = 2$ and $RS_i = 1$. Let ‘y’ represent the request size RS_i (current request size) and ‘x’ represent the request size RS_{i-1} (predecessor request size) in the table. The table is obtained by substituting RS_i and RS_{i-1} values along with the allocation of B_i and B_{i-1} based on the algorithm 6.5.1, in the latency equation 6.13.

In the following lemma, step 1 is proved.

Lemma 3 Given a set of requestors R and $\forall R_i \in R$, the latency of the current requestor R_i reduces or remains the same when $RS_i \leq RS_{i-1}$ when compared to $RS_i > RS_{i-1}$.

Proof: Let R_i represent the current requestor (request size RS_i) and R_{i-1} is its predecessor (request size RS_{i-1}). Let R'_{i-1} be the new predecessor requestor (request size RS'_{i-1}). Let l_i and l'_i be the latency of R_i when its predecessor is R_{i-1} and R'_{i-1} respectively.

Let us refer to the latencies derived for each requestor presented in the Table 6.2. For a requestor R_i with size RS_i (denoted by y), its predecessor R_{i-1} can take one of the RS_{i-1} values (denoted by x). For instance, a requestor with size 4 ($y=4$), can have either one of the requestors with size greater than or equal to 8, or equal to 4, or 2, or 1 as its predecessor (value in row x).

From the Table 6.2, it can be seen that for each requestor in column y, the latency reduces when the predecessor request size increases, i.e., l_i for $RS_{i-1} = 1$ is greater than l_i for $RS_{i-1} = 2$. This holds true for any request size RS_i given in the Table 6.2.

Thus it can be concluded based on the Table 6.2 that,

$$l'_i \leq l_i \text{ when } RS'_{i-1} < RS_i. \quad \blacksquare$$

Based on the Lemma 3, given a set of requestors with multiple inversions, each inversion is swapped such that the predecessor of the current requestor has larger request size. This process is continued until there are no more inversions other than the requestor R_1 and R_n ($RS_1 > RS_n$). By this, the latency of each requestor with the exception of R_1 is reduced. The requestor R_1 will have maximum latency as its predecessor R_n has least request size. To show that the worst-case latency cannot be reduced further, the lemma 4 is proposed as follows.

Lemma 4 *For the requestor R_1 , the predecessor is R_n and $RS_n \leq RS_1$. In this case, any further reduction in the latency of R_1 is at most the increase in the latency for R_n .*

Proof: Let us refer to the latencies derived for each requestor presented in the Table 6.2 and let us consider two separate cases for this proof. Also, let R_1 and R_n be the first and last requestors with request sizes RS_1 and RS_n respectively. Let R'_n be some R_i represent the new last requestor to reduce the latency of R_1 , where $RS'_n > RS_n$. This gives us two separate cases to be analyzed. In the first case, it is assumed that there are multiple requestors of size RS_i . Hence, making this requestor R_i as R'_n changes the latency only for R'_n . In the second case, it is assumed that there is only one requestor of size RS_i . Hence, making this requestor R_i as R'_n changes not only the latency of R'_n but also for R_{i+1} .

Case 1: As a first case, it is assumed that there are multiple requestors of size RS_i .

To prove this case, two specific examples are considered and then generalized for all cases.

Example 1: When the request size $RS_1 \geq 8$, the maximum latency that it can have is when $RS_n = 1$ (from Table 6.2). To reduce this latency, the predecessor request size RS_n must be increased as evident from the Table 6.2. Let a requestor of size $RS_i = 4$ be made as RS'_n . This reduces the latency of the requestor R_1 by 12 ($23 + 4y - 35 - 4y = -12$). However, the latency of $RS'_n = 4$ changes as its predecessor changes. In the first instance, the requestor of size $RS_i = 4$ has its predecessor to be another requestor of size $RS_{i-1} = 4$. Now, the new predecessor for RS'_n is $RS_n = 1$. From the latencies computed in Table 6.2, the change in the latency for the requestor of size 4 can be computed as ($39 - 51 = +12$). Hence, the effective change in the worst-case latency is $-12 + 12 = 0$.

Example 2: When the requestor $RS_1 \geq 8$, the maximum latency that it can have is when $RS_n = 1$ (from Table 6.2). To reduce this latency, the predecessor requestor RS'_n must be equal to 2. This reduces the latency of RS_1 to ($31 + 4y - 35 - 4y = -4$). However, the latency of $RS'_n = 2$ increases as its predecessor changes from request size 2 to 1. The maximum difference in latency is ($46 - 39 = +7$). Hence, the net difference is an increase of +3.

Thus, using the Table 6.2, it can be established that any reduction in the latency for requestor R_1 (for any request size) after sorting the requestors in their descending order, will be at most equal to the increase in the latency for R'_n .

This can be generalized for the Table 6.2 as, $l_1 + l_n \leq l'_1 + l'_n$, where $R_1 > R_n$ and $R'_n > R_n$

Case 2: For this case it is assumed that there is only one requestor of size RS_i .

To prove this case, one specific example is explained and then a proof is generalized. When requestor $R_1 \geq 8$, the maximum latency that it can have is when $R_n = 1$ (from Table 6.2). To reduce this latency, the predecessor request size RS_n must be increased as evident from the Table 6.2. Let a requestor of size $RS_i = 4$ be made as RS'_n . This reduces the latency of the requestor R_1 by 12. However, the latency of $R'_n = 4$ changes as its predecessor changes. Also, since there is only one requestor of size $RS_i = 4$, the latency of requestor of size $RS_i = 2$ changes. From the table, the net change in the latency is zero. Thus, it can be seen that any change in the requestor ordering to decrease the latency of R_1 is compensated by an equal increase in the latency of the other requestors whose ordering has been changed. Likewise, all the other sub-cases can be evaluated from the Table 6.2 and generalized as following,

$$l_1 + l_n \leq l'_1 + l'_n + l_{i+1}, \text{ where } R_1 > R_n \text{ and } R'_n > R_n \text{ and } R_{i+1} < R'_n$$

Thus, it can be concluded that any decrease in the latency for the requestor R_1 , is at most equal to the increase in the latency of the other requestors. ■

Using the Lemmas 3 and 4, the optimality of the algorithm 6.5.1 is proved in the following theorem. The proposed algorithm is optimal, meaning the algorithm always returns feasible bank allocation with minimum worst-case latency.

Theorem 2 Algorithm 2 is an optimal shared bank allocation strategy for obtaining minimum worst-case latency.

Proof: For algorithm 6.5.1 to be optimal for worst-case latency, the bank allocation returned by the algorithm must result in the minimum worst-case latency. When all requestors have $RS_i \geq 8$, the algorithm 6.5.1 assigns $\min(RS_i/2, BI_{max})$ number of banks for each requestor. This is because when all the requestors have $RS_i \geq 8$, the latency of each requestor is dependent only on the current request size and is independent of its predecessor for the allocation (we know that the term $T6$ of the equation 6.13 dominates all the other terms and is the least achievable latency when $RS_i \geq 8$ for all requestors as seen in the cell 1 (latency = $12+4y$) of the Table 6.2).

When requestors have size $RS_i < 8$ and $RS_i \geq 8$, the bank allocation for each requestor is given

by lines 5 - 11 of the algorithm 6.5.1. This allocation is based on the computed latencies provided in the Table 6.2. The algorithm first sorts all the requestors in the descending order of their request sizes RS_i in the line 5. Based on the lemma 3, it can be seen that sorting of the requestors in descending order reduces the latency for each requestor except R_1 . This is because for requestor R_1 , the predecessor would be R_n , where $RS_n < RS_1$. Hence, for requestor R_1 the latency would be higher if its predecessor has a smaller request size than itself, as seen in the Table 6.2. But, based on lemma 4, it can be seen that any further effort to reduce the latency of requestor R_1 results in the increase in the latency to an equivalent or greater amount for other requestors. This implies that the sum latency is the least when the requestors are sorted in the descending order of their request sizes.

Thus, the allocation provided by algorithm 6.5.1 always results in minimum l_{wc} which proves the optimality of the algorithm 6.5.1 with respect to least worst-case latency.

■

6.6 Conclusion

Thus, in this chapter two optimal algorithms for allocating banks to cores under two specific scenarios has been presented. First is when the banks of a DDR are private to the cores and second is when the banks of the DDR are shared across the cores. This solves the problem of optimally allocating the DDR memory resource (banks) to the cores. This contribution is significant as the algorithms presented in this chapter are optimal, meaning if there exists a bank allocation, the algorithms presented would be able to find it. If the algorithms return a failure in allocation, then there exist no solution for the same. Also, optimally allocating the banks of a DDR to its cores helps to maintain the utilization of the DDR as no core will be over allocated.

Chapter 7

Conclusions and Future Work

In order to use multi-core processors in real-time systems, it is important that the WCET of the application for the hardware on which the application is being executed should be determinable with minimum pessimism. In order for the WCET to be estimated, the architectural features of hardware on which the application is executed must be known. The current hardware commercially available for the use in real-time systems are the multi-cores as the chip industry has overwhelmingly moved towards this technology for the want of processing power and SWaP benefits. Hence, it is essential that there are solutions as to how to use these multi-cores in real-time systems. This chapter summarizes the contributions of this dissertation and also provides an overview on the possible future directions to use multi-cores predictably in real-time systems.

7.1 Summary of Contributions

The focus of this dissertation is to devise a path to obtain possible solutions to make multi-core processors predictable. This has been achieved in several ways. As a first step, this work focuses on an extensive literature study which is to clearly state the challenges facing the multi-core processors in RTS. The result of this extensive study is the identification of three distinct areas under which every existing solutions can be categorized. This has been identified in Chapters 1 and 3. Although this work focuses specifically on the modifications to the hardware, a brief overview on all the contributions to the different categories is presented. It also critically identifies the major challenges that hamper the precise estimation of WCET. These are the shared resources on the multi-cores which are the shared caches and its controller, shared memory and its controller and the shared system bus. There are secondary

components on the multi-core such as system interrupts, power saving strategies and others which also contribute to the unpredictability in the multi-core. However, only the primary challenges which are the shared resources are the main focus of this dissertation as they have a major influence in the WCET estimation.

Having these challenges identified, the dissertation proceeds to Chapter 4 to study the different challenges in detail using a commercially available multi-core processor, namely the P4080. A detailed architecture description of the Freescale P4080 multi-core processor is provided. Further, the architectural features that can be modified using the internal registers of the P4080 is analyzed, to possibly minimize those features which directly contribute to the unpredictability in the execution of the application. From this analyses, it has been identified that the features such as out-of-order execution and branch prediction hinder the execution of the code and hence have been disabled. Two different configurations of the P4080 was created by using the internal registers and configuring the local access windows. The first configuration is where the memory subsystem is shared across the cores. In the second configuration, the memory subsystem is partitioned such that each core has its space in the DDR memory and correspondingly in the shared L3 cache. Using these 2 configurations, multiple experiments under different configurations of the internal registers were performed. Experiments on the contention in DDR controller, cache controller, interference in L3 shared cache and also the traffic in the corenet fabric were conducted. In each of the cases, the overhead in the execution of an observed core is quantified, when the number of simultaneously executing cores are increased. It is known that the number of memory subsystem controllers cannot be scaled in accordance with the increase in the number of cores. Hence, these experiments show that the latency in the execution of an application on an observed core increases when the number of simultaneously executing cores increases. It also quantifies this increase in the latency. This increase in the latency is irrespective of the configuration of the shared resources. Hence, an important conclusion that can be arrived is that partitioning of the shared resources does not ensure a predictable execution time. This can be attributed to the fact that the arbitration and command generation patterns inside the memory subsystem controllers are unknown. This leads us to the next contribution of this dissertation, which is the proposal of a novel SoC architecture for COTS multi-cores.

With a sound understanding of the challenges in the shared memory subsystem of the COTS multi-core as studied in Chapter 4, the dissertation proceeds to propose a novel SoC COTS multi-core architecture on the Xilinx Zynq706 platform in Chapter 5. The motivation behind proposing this platform is to solve the unpredictability in the multi-cores arising due to shared memory subsystem as well as benefit from utilizing as much COTS components as possible. Thus, the hybrid COTS-FPGA architecture was

proposed on the Zynq706 platform wherein the dual-core ARM A9 processors were connected to a custom built memory subsystem on the FPGA. The memory subsystem included a predictable arbitration mechanism (round-robin) and a latency bound memory controller designed as per JEDEC standards. By analyses it is shown that the latency to execute an application on an ARM A9 core can be bounded. This architecture leads us to study the memory subsystem (specifically the DDR) in much more detail and address another specific problem related to the use of DDR by the applications executing in the cores.

As a final contribution, in Chapter 6 the bank allocation policy of a DDR memory is studied under two different scenarios. The first scenario is when the banks of the DDR are shared among all the cores in the multi-core and in the second scenario the banks are partitioned. In either case an optimal allocation policy is obtained. To perform a perfect bank allocation is always critical while building real-time systems because of several factors. The DDR memory is a limited resource on the multi-core processor and is being shared by all the cores. When there are multiple cores with different applications executing simultaneously, it is important that the Worst Case Execution Time requirement for every application is satisfied. This is directly influenced by the manner in which the DDR memory is accessed. Hence, allocating the optimal number of banks for is crucial for achieving the worst-case execution time requirement for each application.

Thus, the three technical contributions in this dissertation help address the unpredictability in the memory subsystem with the focus being the DDR memory and its controller.

7.2 Future Work

Based on the contributions of this dissertation, the possible future work directions can be listed as follows.

7.2.1 Further analyses of Freescale P4080

The current analyses of the P4080DS focused only on the architectural aspects of the multi-core that contributed to the unpredictability in the execution of the application. The current testbed is a bare-metal implementation, meaning that the application is directly executed on the core itself. This can provide control to the internal features of the multi-core only to some extent. However, the P4080DS allows users to further control the internal features of the architecture using a hypervisor.

The hypervisor is a layer of computer software or firmware that lies below the operating system and

Figure 7.1: Predictable COTS-FPGA Architecture

has exclusive privilege on the underlying hardware components. The user application and the guest operating systems execute on a level above this hypervisor. P4080 allows to control the resource usage of each partition using the hypervisor. The hypervisor although incurring an overhead, provides a bit more control and visibility on the internal registers. This might help in further analyses of the internal features of the architecture which could also have an effect on the execution time such as the Memory Management Unit (MMU) and Peripheral Access Management Unit (PAMU). These units although may not be the primary sources of unpredictability, can have some effect on the program execution on the cores when the system is set up with an RTOS and its peripherals. Hence, it is important that even the smallest architectural feature inside a COTS multi-core is analyzed and its impact on program execution is studied and quantified.

7.2.2 Improvement to the SoC-COTS architecture

The architecture proposed in Chapter 5, has not been tested on a hardware platform due to the limitations of the platform chosen for implementation as explained under Section 5.6. There are multiple future directions with this current architecture implementation which are listed as follows.

1. Porting of the current architecture that has been implemented for Xilinx Zynq706, on other com-

patible, newer state-of-the-art SoCs which are due to be released in the near future to provide a hardware level testing of the implementation. The key requirement of these future SoCs is that the PL side of the SoC must be interfaced to a DDR memory and multi-core debugging feature through JTAG should be available. The advantage of the current implementation is that it is modular follows the fundamental communication protocol, i.e., AXI and the interface to the external DDR follows the micron memory model and the JEDEC standards. This implies that the implementation can be easily ported to any SoC with ARM cores and PL with DDR memory. The porting of the architecture should help in its complete hardware verification and validation.

2. The current architecture is completely focused on the design of a predictable memory controller where the latency from the PL side to the instant the data is read or written is bounded. This provides a basic building block for the memory subsystem to be completely implemented in a predictable manner on the PL. The current architecture should be extrapolated as shown in Figure 7.1 to complete the entire memory subsystem on the FPGA. This will ensure that the memory subsystem can be bounded to a cycle accurate latency as it is implemented on the FPGA. The private caches local to the ARM A9 cores can be enabled as a first level of caching mechanism. To improve the time for memory accesses on the PL and to provide efficient sharing of data between the cores, a shared cache architecture needs to be implemented on the FPGA which can then be interfaced to the custom memory controller as shown in Figure 7.1. The advantage of having a shared cache on the FPGA would be that the cache can be implemented based on the AXI communication mechanism. Hence, it is important to develop a predictable and an efficient data sharing mechanism in the shared cache as a part of future work.

7.2.3 Optimal Bank Allocation Algorithm when banks can be either private or shared between requestors

The bank allocation policy in the Chapter 6 has been implemented for two scenarios. First, is when atleast one of banks allocated to a requestor is shared among other requestors and second, is when the banks allocated to a requestor is completely privatized. In either of the cases, an optimal bank allocation algorithm has been proposed.

However, it is important that within a given framework, there must be flexibility to allocate banks either shared or private based on the requirements. For instance, in a system with a set of applications which may include hard, firm and soft real-time, there could be a requirement for all hard real-time applications to have private bank allocation whereas the firm and soft real-time can share DDR banks

as long as they can meet their execution time requirements. In order to satisfy such scenarios, as a part of future plan, work presented in Chapter 6 will be improved as follows. The current derivation for determining the latency of a requestor has been presented for two cases. One is when the banks of the requestors are strictly private and second is when atleast one of the banks of the requestors are shared. As a potential future direction this work can be extended by determining the latency of the requestor when it can be either shared or private. Using these latency equations, an optimal algorithm to allocate banks to requestors which can be either private or shared can be derived. This completes all the possible cases of bank allocation.

References

- [1] “P4080 Development System User’s Guide,” <https://www.nxp.com/docs/en/user-guide/P4080DSUG.pdf>, 2010, [Online; accessed 15-June-2016].
- [2] Zynq SoC. <http://www.xilinx.com/products/silicon-devices/soc.html>.
- [3] M. T. note, “Designing for 1Gb DDR SDRAM.”
- [4] B. Prince and D. B. Prince, *High performance memories: new architecture DRAMs and SRAMs-evolution and function*. Wiley, 1999.
- [5] Y. Li, B. Akesson, and K. Goossens, “Architecture and analysis of a dynamically-scheduled real-time memory controller,” *Real-Time Systems*, vol. 52, no. 5, pp. 675–729, 2016.
- [6] H. Kopetz, *Real-time systems: design principles for distributed embedded applications*. Springer Science & Business Media, 2011.
- [7] M. J. S. Smith, *Application-specific integrated circuits*. Addison-Wesley Reading, MA, 1997, vol. 7.
- [8] “Certification Authorities Software Team (CAST),” https://www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/, 2014, [Online; accessed 15-June-2016].
- [9] A. Binstock. (2012) Multi-core processor architecture explained. <https://software.intel.com/en-us/articles/frequently-asked-questions-intel-multi-core-processor-architecture>. [Online; accessed 15-June-2016].
- [10] S. Podlipnig and L. Böszörményi, “A survey of web cache replacement strategies,” *ACM Computing Surveys (CSUR)*, vol. 35, no. 4, pp. 374–398, 2003.
- [11] M. S. Papamarcos and J. H. Patel, “A low-overhead coherence solution for multiprocessors with private cache memories,” in *ACM SIGARCH Computer Architecture News*, vol. 12, no. 3. ACM, 1984, pp. 348–354.
- [12] J. Standard, “Ddr3 sdram standard,” *JESD79-3, Jun*, 2007.

REFERENCES

- [13] “Future Advances in Body Electronics,” http://cache.freescale.com/files/automotive/doc/white_paper/BODYDELECTRWP.pdf, 2013, [Online; accessed 15-June-2016].
- [14] “Rapitime Analysis Tool,” <http://www.rapitasystems.com/products/rapitime>, 2004, [Online; accessed 15-June-2016].
- [15] M. Shekhar, A. Sarkar, H. Ramaprasad, and F. Mueller, “Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems,” in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2012, pp. 331–340.
- [16] H. Shah, K. Huang, and A. Knoll, “Timing anomalies in multi-core architectures due to the interference on the shared resources,” in *Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 708–713.
- [17] D. Dasari and V. Nelis, “An analysis of the impact of bus contention on the wcet in multicores,” in *International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems (HPCC-ICESS)*. IEEE, 2012, pp. 1450–1457.
- [18] R. Pellizzoni and M. Caccamo, “Impact of peripheral-processor interference on wcet analysis of real-time embedded systems,” *IEEE Transactions on Computers*, vol. 59, no. 3, pp. 400–415, 2010.
- [19] C. Cullmann, C. Ferdinand, G. Gebhard, D. Grund, C. Maiza, J. Reineke, B. Triquet, and R. Wilhelm, “Predictability considerations in the design of multi-core embedded systems,” *Proceedings of Embedded Real Time Software and Systems*, pp. 36–42, 2010.
- [20] H. Shah, A. Raabe, and A. Knoll, “Challenges of WCET analysis in COTS multi-core due to different levels of abstraction,” in *High-performance and Real-time Embedded Systems (HiRES 2013)*, Berlin, 2013.
- [21] P. Radojković, S. Girbal, A. Grasset, E. Quinones, S. Yehia, and F. J. Cazorla, “On the evaluation of the impact of shared resources in multithreaded cots processors in time-critical environments,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, p. 34, 2012.
- [22] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, “Worst case delay analysis for memory interference in multicore systems,” in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 741–746.
- [23] I. Bate, P. Conmy, T. Kelly, and J. McDermid, “Use of modern processors in safety-critical applications,” *The Computer Journal*, vol. 44, no. 6, pp. 531–543, 2001.

REFERENCES

- [24] G. Gracioli, A. Alhammad, R. Mancuso, A. A. Fröhlich, and R. Pellizzoni, “A survey on cache management mechanisms for real-time embedded systems,” *ACM Computing Surveys (CSUR)*, vol. 48, no. 2, p. 32, 2015.
- [25] D. Abts, N. D. Enright Jerger, J. Kim, D. Gibson, and M. H. Lipasti, “Achieving predictable performance through better memory controller placement in many-core cmps,” in *Proceedings of the 36th Annual International Symposium on Computer Architecture*. ACM, 2009, pp. 451–461.
- [26] M. Paolieri, J. Mische, S. Metzlauff, M. Gerdes, E. Quinones, S. Uhrig, T. Ungerer, and F. J. Cazorla, “A hard real-time capable multi-core smt processor,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 12, no. 3, p. 79, 2013.
- [27] A. Hansson, K. Goossens, M. Bekooij, and J. Huiskens, “Compsoc: A template for composable and predictable multi-processor system on chips,” *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 14, no. 1, p. 2, 2009.
- [28] “Multi-Core Execution of Parallelised Hard Real-Time Applications Supporting Analysability,” <http://www.parmerasa.eu/>, 2012, [Online; accessed 15-June-2016].
- [29] Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra, “Timing analysis of concurrent programs running on shared cache multi-cores,” *Real-Time Systems*, vol. 48, no. 6, pp. 638–680, 2012.
- [30] D. H. Puaut, “Wcet analysis of multi-level set-associative instruction caches,” *arXiv preprint arXiv:0807.0993*, 2008.
- [31] D. Hardy, T. Piquet, and I. Puaut, “Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches,” in *Real-Time Systems Symposium*. IEEE, 2009, pp. 68–77.
- [32] C. Ferdinand and R. Wilhelm, “Efficient and precise cache behavior prediction for real-time systems,” *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, 1999.
- [33] T. Kelter, H. Falk, P. Marwedel, S. Chattopadhyay, and A. Roychoudhury, “Bus-aware multi-core wcet analysis through tdma offset bounds,” in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2011, pp. 3–12.
- [34] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, “Chronos: A timing analyzer for embedded software,” *Science of Computer Programming*, vol. 69, no. 1, pp. 56–67, 2007.
- [35] J. Nowotsch, M. Paulitsch, D. Buhler, H. Theiling, S. Wegener, and M. Schmidt, “Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement,” in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2014, pp. 109–118.

REFERENCES

- [36] M. Lv, W. Yi, N. Guan, and G. Yu, “Combining abstract interpretation with model checking for timing analysis of multicore software,” in *Real-Time Systems Symposium (RTSS)*. IEEE, 2010, pp. 339–349.
- [37] D. Dasari, B. Andersson, V. Nelis, S. M. Petters, A. Easwaran, and J. Lee, “Response time analysis of cots-based multicores considering the contention on the shared memory bus,” in *International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2011, pp. 1068–1075.
- [38] H. Kim, D. de Niz, B. Andersson, M. Klein, O. Mutlu, and R. Rajkumar, “Bounding memory interference delay in cots-based multi-core systems,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 145–154.
- [39] H. Yun, R. Pellizzoni, and P. K. Valsan, “Parallelism-aware memory interference delay analysis for cots multicore systems,” in *Real-Time Systems (ECRTS), 2015 27th Euromicro Conference on*. IEEE, 2015, pp. 184–195.
- [40] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, “Worst case delay analysis for memory interference in multicore systems,” in *Proceedings of the Conference on Design, Automation and Test in Europe*. European Design and Automation Association, 2010, pp. 741–746.
- [41] K. Bletsas *et al.*, “Sporadic multiprocessor scheduling with few preemptions,” in *Euromicro Conference on Real-Time Systems*. IEEE, 2008, pp. 243–252.
- [42] V. Suhendra and T. Mitra, “Exploring locking & partitioning for predictable shared caches on multi-cores,” in *Proceedings of the 45th annual Design Automation Conference*. ACM, 2008, pp. 300–303.
- [43] A. Alhammad and R. Pellizzoni, “Time-predictable execution of multithreaded applications on multicore systems,” in *Proceedings of the conference on Design, Automation & Test in Europe*. European Design and Automation Association, 2014, p. 29.
- [44] H. Kim, A. Kandhalu, and R. Rajkumar, “A coordinated approach for practical os-level cache management in multi-core real-time systems,” in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2013, pp. 80–89.
- [45] B. C. Ward, J. L. Herman, C. J. Kenna, and J. H. Anderson, “Making shared caches more predictable on multicore platforms,” in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2013, pp. 157–167.
- [46] H. Yun, R. Mancuso, Z.-P. Wu, and R. Pellizzoni, “Palloc: Dram bank-aware memory allocator for performance isolation on multicore platforms,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2014, pp. 155–166.

REFERENCES

- [47] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, “Memguard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, pp. 55–64.
- [48] R. Mancuso, R. Dudko, E. Betti, M. Cesati, M. Caccamo, and R. Pellizzoni, “Real-time cache management framework for multi-core architectures,” in *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2013, pp. 45–54.
- [49] L. Sha, M. Caccamo, R. Mancuso, J.-E. Kim, M.-K. Yoon, R. Pellizzoni, H. Yun, R. Kegley, D. Perlman, G. Arundale *et al.*, “Single core equivalent virtual machines for hard real—time computing on multicore processors,” 2014.
- [50] L. Kosmidis, J. Abella, E. Quinones, and F. J. Cazorla, “A cache design for probabilistically analysable real-time systems,” in *Design, Automation Test in Europe Conference Exhibition (DATE)*, March 2013, pp. 513–518.
- [51] Y. Huangfu and W. Zhang, “Peg-c: Performance enhancement guaranteed cache for hard real-time systems,” *IEEE Embedded Systems Letters*, vol. 6, no. 2, pp. 17–20, June 2014.
- [52] J. Reineke, “Randomized caches considered harmful in hard real-time systems,” *Leibniz Transactions on Embedded Systems (LITES)*, vol. 1, no. 1, pp. 03:1–03:13, 2014.
- [53] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen, “Wcet centric data allocation to scratchpad memory,” in *Real-Time Systems Symposium*. IEEE, 2005, pp. 10–pp.
- [54] T. Chen, T. Mitra, A. Roychoudhury, and V. Suhendra, “Exploiting branch constraints without exhaustive path enumeration,” in *OASICS-OpenAccess Series in Informatics*, vol. 1. Schloss Dagstuhl-Leibniz-Zentrum fr Informatik, 2007.
- [55] H. Takase, H. Tomiyama, and H. Takada, “Partitioning and allocation of scratch-pad memory for priority-based preemptive multi-task systems,” in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2010, pp. 1124–1129.
- [56] S. Metzlaß, I. Guliashvili, S. Uhrig, and T. Ungerer, “A dynamic instruction scratchpad memory for embedded processors managed by hardware,” in *Architecture of Computing Systems-ARCS 2011*. Springer, 2011, pp. 122–134.
- [57] S. Wasly and R. Pellizzoni, “A dynamic scratchpad memory unit for predictable real-time embedded systems,” in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2013, pp. 183–192.
- [58] Y. Liu and W. Zhang, “Exploiting multi-level scratchpad memories for time-predictable multicore computing,” in *International Conference on Computer Design (ICCD)*. IEEE, 2012, pp. 61–66.

REFERENCES

- [59] Y. Kim, D. Broman, J. Cai, and A. Shrivastava, “Wcet-aware dynamic code management on scratchpads for software-managed multicores,” in *Real-Time and Embedded Technology and Application Symposium (RTAS)*. IEEE, 2014.
- [60] M.-F. Kuo, P. Roop, S. Andalam, and N. Patel, “Precision timed embedded systems using tickpad memory,” in *International Conference on Application of Concurrency to System Design (ACSD)*. IEEE, 2013, pp. 206–215.
- [61] G. Berry and G. Gonthier, “The estereel synchronous programming language: Design, semantics, implementation,” *Science of computer programming*, vol. 19, no. 2, pp. 87–152, 1992.
- [62] P. Roop, S. Andalam, and A. Girault, “Pret-c: A new language for programming precision timed architectures,” *INRIA Grenoble Rhone-Alpes*, 2009.
- [63] B. Akesson, K. Goossens, and M. Ringhofer, “Predator: a predictable sdram memory controller,” in *Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*. ACM, 2007, pp. 251–256.
- [64] J. Reineke, I. Liu, H. D. Patel, S. Kim, and E. A. Lee, “Pret dram controller: Bank privatization for predictability and temporal isolation,” in *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*. ACM, 2011, pp. 99–108.
- [65] D. Wang, B. Ganesh, N. Tuaycharoen, K. Baynes, A. Jaleel, and B. Jacob, “Dramsim: a memory system simulator,” *ACM SIGARCH Computer Architecture News*, vol. 33, no. 4, pp. 100–107, 2005.
- [66] S. Goossens, J. Kuijsten, B. Akesson, and K. Goossens, “A reconfigurable real-time sdram controller for mixed time-criticality systems,” in *International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE, 2013, pp. 1–10.
- [67] L. Ecco, S. Tobuschat, S. Saidi, and R. Ernst, “A mixed critical memory controller using bank privatization and fixed priority scheduling,” in *2014 IEEE 20th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE, 2014, pp. 1–10.
- [68] M. D. Gomony, B. Akesson, and K. Goossens, “Architecture and optimal configuration of a real-time multi-channel memory controller,” in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE ’13, 2013, pp. 1307–1312.
- [69] Y. Krishnapillai, Z. P. Wu, and R. Pellizzoni, “A rank-switching, open-row dram controller for time-predictable systems,” in *Euromicro Conference on Real-Time Systems (ECRTS)*. IEEE, 2014, pp. 27–38.

REFERENCES

- [70] H. Kim, D. Broman, E. A. Lee, M. Zimmer, A. Shrivastava, and J. Oh, “A predictable and command-level priority-based dram controller for mixed-criticality systems,” in *21st IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, 2015, pp. 317–326.
- [71] M. Paolieri, E. Quiñones, F. Cazorla, and M. Valero, “An analyzable memory controller for hard real-time cmps,” *Embedded Systems Letters, IEEE*, vol. 1, no. 4, pp. 86–90, Dec 2009.
- [72] M. Paolieri, E. Quinones, F. J. Cazorla, and M. Valero, “An analyzable memory controller for hard real-time cmps,” *IEEE Embedded Systems Letters*, vol. 1, no. 4, pp. 86–90, 2009.
- [73] B. Rumpler, “Complexity management for composable real-time systems,” in *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC’06)*. IEEE, 2006, pp. 9–pp.
- [74] J. Jalle, L. Kosmidis, J. Abella, E. Quiñones, and F. J. Cazorla, “Bus designs for time-probabilistic multicore processors,” in *Design, Automation and Test in Europe Conference and Exhibition (DATE)*. IEEE, 2014, pp. 1–6.
- [75] K. Goossens and A. Hansson, “The aethereal network on chip after ten years: Goals, evolution, lessons, and future,” in *Design Automation Conference (DAC)*. IEEE, 2010, pp. 306–311.
- [76] M. Schoeberl, F. Brandner, J. Sparsø, and E. Kasapaki, “A statically scheduled time-division-multiplexed network-on-chip for real-time systems,” in *Sixth IEEE/ACM International Symposium on Networks on Chip (NoCS)*. IEEE, 2012, pp. 152–160.
- [77] M. Schoeberl, D. V. Chong, W. Puffitsch, and J. Sparsø, “A time-predictable memory network-on-chip,” in *OASICS-OpenAccess Series in Informatics*, vol. 39. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2014.
- [78] R. Wilhelm and J. Reineke, “Embedded Systems: Many Cores; Many Problems,” in *IEEE Symposium on Industrial Embedded Systems (SIES)*, 2012, pp. 176–180.
- [79] A. Abel, F. Benz, J. Doerfert, B. Dörr, S. Hahn, F. Hauptenthal, M. Jacobs, A. H. Moin, J. Reineke, B. Schommer, and R. Wilhelm, “Impact of Resource Sharing on Performance and Performance Prediction: A Survey,” in *International Conference on Concurrency Theory (CONCUR)*, 2013, pp. 25–43.
- [80] D. Dasari, B. Akesson, V. Nelis, M. Awan, and S. Petters, “Identifying the Sources of Unpredictability in COTS-based Multicore Systems,” in *IEEE Symposium on Industrial Embedded Systems (SIES)*, 2013, pp. 39–48.
- [81] J. Nowotsch and M. Paulitsch, “Leveraging Multi-core Computing Architectures in Avionics,” in *IEEE Dependable Computing Conference (EDCC)*, 2012, pp. 132–143.

REFERENCES

- [82] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele, "Memory Interference Delay Estimation for Multicore Systems," Technical report, 2009. Available at <http://netfiles.uiuc.edu/rpelliz2/www>, Tech. Rep.
- [83] H. Yun, G. Yao, R. Pellizzoni, M. Caccamo, and L. Sha, "Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality," in *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, July 2012, pp. 299–308.
- [84] V. Suhendra and T. Mitra, "Exploring Locking & Partitioning for Predictable Shared Caches on Multi-Cores," in *Design Automation Conference (DAC)*, 2008, pp. 300–303.
- [85] "P4080 Development System," <https://www.nxp.com/docs/en/fact-sheet/P4080DSFS.pdf>, 2010, [Online; accessed 15-June-2016].
- [86] (2013) CodeWarriorSoftware Development. https://www.nxp.com/design/software/development-software/codewarrior-development-tools:CW_HOME.
- [87] (2020) CodeWarrior TAP Probe. https://www.nxp.com/design/software/development-software/codewarrior-development-tools/run-control-devices/codewarrior-tap:CW_TAP.
- [88] (2013) CodeWarrior TAP Probe User Guide. http://cache.freescale.com/files/soft_dev_tools/doc/user_guide/CWTAPUG.pdf.
- [89] "QorIQ P4080 Communications Processor Product Brief," http://cache.freescale.com/files/32bit/doc/prod_brief/P4080PB.pdf, 2010, [Online; accessed 15-June-2016].
- [90] MATLAB Curve Fitting Toolbox. <http://www.mathworks.com/help/curvefit/index.html>.
- [91] J. H. Mathews, K. D. Fink *et al.*, *Numerical methods using MATLAB*. Pearson prentice hall Upper Saddle River, NJ, 2004, vol. 4.
- [92] N. Levinson, "The wiener (root mean square) error criterion in filter design and prediction," *Journal of Mathematics and Physics*, vol. 25, no. 1-4, pp. 261–278, 1946.
- [93] (2013) Hardware Support for Robust Partitioning in Freescale QorIQ Multicore SoCs. cache.freescale.com/files/32bit/doc/white_paper/QORIQHSRPWP.pdf.
- [94] Zynq-7000 SoC and 7 Series Devices Memory Interface Solutions. <http://www.xilinx.com/products/intellectual-property/mig.html%##documentation>.
- [95] Altera Cyclone V SoC. <https://www.altera.com/products/soc/portfolio/cyclone-v-soc/overview.html>.
- [96] Introduction to AXI Protocol: Understanding the AXI interface. <https://community.arm.com/developer/ip-products/system/b/soc-design-blog/posts/introduction-to-axi-protocol-understanding-the-axi-interface>.

REFERENCES

- [97] Xilinx Software Development Kit (XSDK). <https://www.xilinx.com/products/design-tools/vivado.html>.
- [98] Vivado Design Suite - HLx Editions. <https://www.xilinx.com/products/design-tools/embedded-software/sdk.html>.
- [99] T. Zadigian, J. Stroud, and M. Moriarty, "Jtag-based programming and debug," Oct. 7 2014, uS Patent 8,856,600.
- [100] Interface IP DDR3 PHY. <https://www.rambus.com/interface-ip/ddrn-phys/ddr3-phy/>.
- [101] "AXI Interconnect," https://www.xilinx.com/products/intellectual-property/axi_interconnect.html, 2010, [Online; accessed 15-June-2016].
- [102] V. A. Pedroni, *Finite State Machines in Hardware: Theory and Design (with VHDL and SystemVerilog)*. The MIT Press, 2013.
- [103] "Simple AMP: Bare-Metal System Running on Both Cortex-A9 Processors," https://www.xilinx.com/support/documentation/application_notes/xapp1079-amp-bare-metal-cortex-a9.pdf, 2014.
- [104] DDR Memories Comparison and overview. <https://www.nxp.com/docs/en/supporting-information/BeyondBits2article17.pdf>.
- [105] J. Standard, "Ddr3 sdram standard," *JESD79-3, JEDEC*, 2012.