TECHNISCHE UNIVERSITÄT MÜNCHEN
Lehrstuhl für Echtzeitsysteme und Robotik

Predictable and high performance multi-core architectures

Hardik Shah

Vollständiger Abdruck der von der Fakultät der Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 22.06.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 29.10.2015 angenommen.

Abstract

Multi-core architectures will provide the computational power needed to the high performance hard real-time systems. Typically, multi-core architectures employ shared resources to reduce cost by decreasing chip area and package size, and to exchange data (for example, shared memory). The interference on the shared resources makes the execution time of applications running on these architectures unpredictable and dependent on activity of co-existing cores and the employed arbitration scheme. Hence, the Worst Case Execution Time (WCET) of applications is impossible or hard to estimate. For industrial deployment, the WCET of safety critical hard real-time applications on an underlying platform must be known and must stay below an enforced limit.

Traditionally, the above mentioned problem is solved by static time partitioning or over-allocating the shared resource to the critical core. Both result in inefficient resource utilization which leads to loss of performance. This thesis provides techniques to achieve time-predictable execution of applications on multi-core architectures preserving their performance. Although modern multi-core architectures have number of shared resources, this thesis focuses on shared memory without loss of generality. Our approach minimally modifies the existing multi-core architectures and existing measurement based WCET analysis tools, preserving performance, cost and time-to-market benefits.

A shared SDRAM arbitrated under a budget based arbiter exhibits complex timing behavior for the worst case interference analysis. Conventional techniques remove this complexity by applying an abstraction. This thesis provides a detailed worst case interference analysis of the aforementioned combination resulting in much tighter WCET bounds. The tighter WCET bounds reduce over-allocation, thus, increase performance of the system indirectly.

Apart from performance and cost benefits, the timing aspect of our techniques is conceptually certifiable according to the CAST32 guidelines.

Zusammenfassung

Mehrkernarchitekturen könnten die notwendige Rechenleistung für leistungsstarke harte Echtzeitsysteme zur Verfügung stellen. Typischerweise verwenden die Mehrkernarchitekturen gemeinsame Ressourcen zur Kostenreduzierung da die Chipfläche und die Packetgröße reduziert werden können, sowie zum Datenaustausch (z.B. gemeinsamer Speicher). Die Interferenz auf den gemeinsamen Ressourcen macht die Ausführungszeit der Anwendungen unvorhersagbar und anhängig von der Aktivität der koexistierenden Kerne und dem Arbitrierungsschema. Aus diesem Grund ist die Einschätzung der Worst Case Execution Time (WCET) der Anwendungen unmöglich oder sehr schwierig. Die WCET der sicherheitskritischen harten Echtzeitanwendungen auf der zugrundeliegenden Plattform für die industriellen Entwicklung muss bekannt sein und unterhalb einer bestimmten Grenze liegen.

Traditionell wird das oben beschriebene Problem gelöst, indem statische Zeitaufteilung vorgenommen wird oder die gemeinsame Ressource dem kritischen Kern zugewiesen wird. Beide Methoden führen zu einer ineffizienten Ressourcenverwertung und folglich zu Performanzverlust. Diese Dissertation präsentiert Methoden zu zeitlich vorhersagbaren Anwendungen auf Mehrkernarchitekturen mit Aufrechterhaltung ihrer Performanz. Obwohl moderne Mehrkernarchitekturen zahlreiche Typen von gemeinsamen Ressourcen enthalten, konzentriert sich diese Dissertation ohne Beschränkung der Allgemeinheit auf den gemeinsamen Speicher. Unsere Lösung modifiziert die vorhandenen Mehrkernarchitekturen und die WCET Analyse Tools minimal und bietet Vorteile bezüglich der Performanz, der Kosten und der Produktionseinführungszeit.

Ein gemeinsamer SDRAM arbitriert durch einen Budget-basierter Vermittler zeigt ein sehr komplexes Timing-Verhalten bei der Analyse der Worstcase-Interferenz. Herkömmliche Methoden reduzieren diese Komplexität mit Hilfe einer Abstraktion. Diese Arbeit präsentiert eine detaillierte Analyse der Worstcase-Interferenz der oben

beschriebenen Kombination mit strengeren WCET-Grenzen. Die strengeren WCET-Grenzen führen indirekt zu einer Performanzsteigerung.

Abseits der Performanz- und Kostenvorteile sind die Timing-Aspekte unserer Methoden konzeptionell mit den CAST32 Richtlinien zertifizierbar.

Acknowledgements

First of all, I would like to express my deepest gratitude to my adviser Prof. Dr. Alois Knoll for giving me the opportunity to work under his guidance on a very interesting and challenging research problem and for his continuous support and inspiration. I would also like to thank Prof. Dr. Heinrich Daembkes for investing time in reviewing my thesis and acting as a second supervisor.

Many thanks go to Dr. Kai Huang and Dr. Andreas Raabe for their guidance and support whenever required. My colleagues Morteza Hashemi, Dr. Ali Nasseri, Martin Eder, Gang Chen, Biao Hu, Dr. Reinhard Lafrenz, Simon Barner and Dr. Christian Buckl encouraged and motivated me. Our staff members, Marie-Luise, Frau Gertrud Eberl, Frau Ute Lomp and Amy Bücherl helped me in organizational activities. Many thanks also to students with whom I worked during past few years.

Finally, I would like to thank my wife Riddhi for her continuous support, love and encouragement. My daughter Aangi cheered me up after long hours in the lab. I am thankful to my parents Charulata Shah and Dr. Dhanendra Shah for all their love and efforts in educating me. My sisters Meenal and Neelam also supported me in achieving quality education.

Contents

List of Figures

List of Tables

List of abbreviations

ACET	Average Case Execution Time
BCET	Best Case Execution Time
B_L	Best Case Latency
CABA	Cycle Accurate Byte Accurate
CCSP	Credit Controlled Static Priority
COTS	Commercial Off-The Shelf
DMA	Direct Memory Access
DPQ	Dynamic Priority Queue
ECU	Electronic Control Unit
FPGA	Field Programmable Gate Array
HRT	Hard Real-time
ILP	Integer Linear Programming
$\mathcal{LR}$	Latency Rate
MMU	Memory Management Unit
MOET	Maximum Observed Execution Time
MPU	Memory Protection Unit
NoC	Network On Chip
OET	Observed Execution Time
PBS	Priority Based Budget Scheduler
PD	Priority Division
RR	Round Robin
SDRAM	Synchronous Dynamic Random Access Memory
SP	Static Priority
SRAM	Static Random Access Memory
SRT	Soft Real-time
SS	Slot Size
TDMA	Time Division Multiple Access
WCET	Worst Case Execution Time
WCRT	Worst Case Response Time
W_L	Worst Case Latency

Chapter 1

Introduction

1.1 Background and Motivation

In the last century, humanity has moved from all mechanical systems to smarter, processor backed mechatronic systems due to their versatility, efficiency, controlability, usability etc. advantages. This shift is also visible in safety critical Hard Real-Time (HRT) systems such as cars and airplanes. On one hand, the safety critical HRT systems benefit from processor backed embedded systems, on the other hand the complexity of the system is ported from mechanical parts to electronics and software. Thus, the cost of a product has been increasingly driven by the contained electronics. An example is depicted in Fig. 1.1 for modern cars. Although the motivation and contribution of this thesis is valid for any HRT systems, automobiles have been used as an illustration here. Airplanes also have similar requirements, however, certification requirements (discussed in Sec. A.1 – excerpt of CAST-32 paper [3]) in airplanes are very strict.

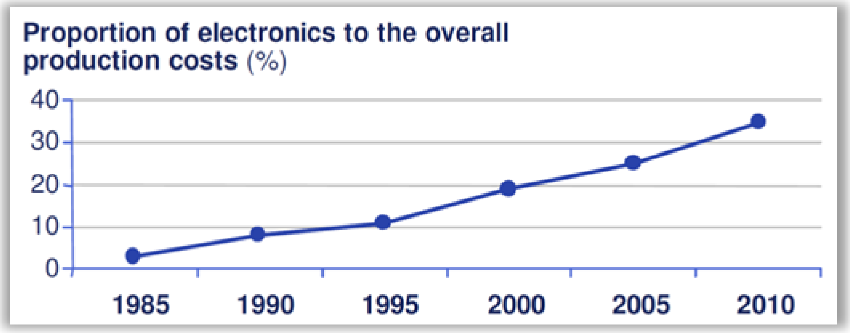
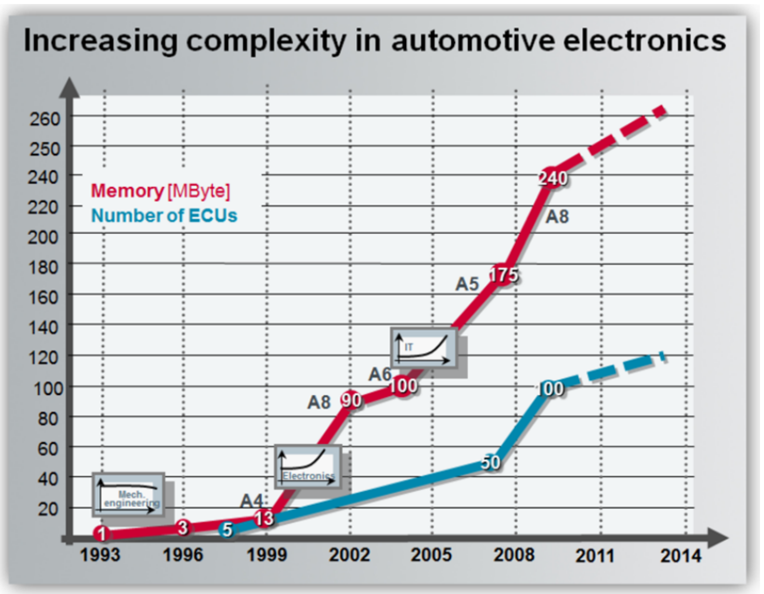


Figure 1.1: Increasing cost of Electronics in Cars. (Source: Market and Technology Study Automotive Power Electronics 2015)



Source:
Paul Milbredt, AUDI AG, EFTA 2010
Switched FlexRay: Increasing the Effective Bandwidth and Safety of FlexRay Networks

Figure 1.2: Ever increasing number of ECUs in modern cars

For each technique presented in this thesis, a short section is dedicated to explain how does it comply with the certification objectives and activities.

For almost two decades, car manufacturers have added a dedicated Electronic Control Unit (ECU) on an appropriate interconnect (aka bus) to add a functionality to the car. This has resulted in ever increasing number of ECUs in modern cars as depicted in Fig. 1.2. Moreover, the ECUs are connected to an appropriate bus (interconnect) depending on their functionality. There are number of buses operating on different communication protocols, for example, CAN [4], LIN [5], MOST [6] and FlexRay [7]. The complexity involving many ECUs, software executing on them and different communication standards limit scalability of modern cars.

The impact of the above mentioned complexity can be understood by the following example. The *engine start-stop automatic* functionality improves fuel efficiency by stopping the engine when the car is standstill. However, ≈ 30 ECUs are involved in making the decision to stop the engine[1]. This makes development of new functionalities and verifying them extremely difficult, error prone and expensive. Moreover, the additional weight of ECUs and cables (≈ 4Km long, 60 Kg weight [8]) reduces fuel economy gained by smarter engine management.

[1]G.Spiegelberg, Siemens AG

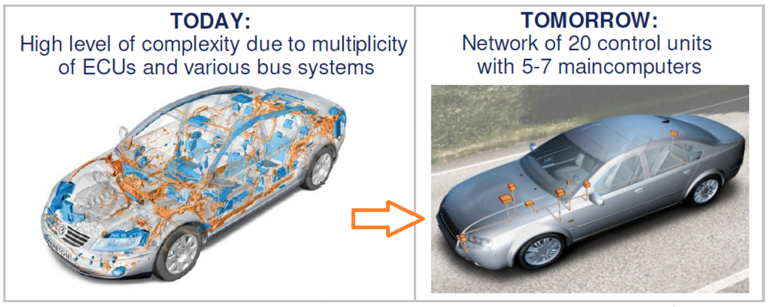

TODAY:
High level of complexity due to multiplicity of ECUs and various bus systems
TOMORROW:
Network of 20 control units with 5-7 maincomputers

Figure 1.3: Centralized architecture. (Source: Market and Technology Study Automotive Power Electronics 2015)

To tackle the above mentioned problem, a centralized architecture is suggested in literature [8] and depicted in Fig. 1.3. Here, fewer number of, $\approx$ 20, high performance ECUs are suggested to implement all functionalities of a car. These ECUs can be connected with each other using a single bus standard, for example, time triggered Ethernet. Such time triggered Ethernet communication is already deployed in industrial Ethernet standards, for example, EtherCat [9], PowerLink [10], TTEthernet [11] etc. The reduction in number ECUs and the single communication standard drastically increase scalability and simplify development and verification process.

The above discussion asserts the need of powerful ECUs in the automotive industry. In the past, achieving high computation power was rather simpler. With each new generation of the processor core the operating frequency was increasing which resulted in higher computation power and faster execution of the existing code without any modifications. However, together with the increase in operating frequency, the power consumption also increases. They are interrelated by the equation (1.1) [12].

$$P \propto C \times V^2 \times f \tag{1.1}$$

In the above equation, P is the consumed power, C is the switched capacitance, V is the respective voltage swing and f is the operating frequency. The consumed power is dissipated as heat resulting in increased temperature and local hot spots. In the last decade, researchers observed that increasing the operating frequency was not feasible anymore due to the unmanageable heat dissipation. The generated heat could damage the chip itself. This phenomenon is popularly known as *power wall*.

Employing multi-cores to increase the computation power of the ECUs is a feasible approach. General purpose systems are using multi-cores since many years now [13]. Although HRT systems may greatly benefit from multi-cores, they are not used in safety critical systems due to the following *research problem.* The interference on shared resources in multi-core architectures prolongs execution time of applications executing on them unpredictably. The Worst Case Execution Time (WCET) of applications must be known and stay below an enforced limit (dead line) in order to achieve certification for commercial use. The shared resource interference in multi-core architectures make the WCET computation impossible or hard to determine weakening the suitability of multi-core architectures in HRT systems.

The above mentioned problem is well-known in academic and industrial community. The academical approach to solve the problem proposes to re-design multi-cores to achieve time predictable execution. Typically, radical new designs are proposed, for example, PRET [14], MERASA [15], CoMPSoC [16], ACROSS [17], RECOMP [18] and T-CREST [19] architectures. These radical new designs may be too expensive for HRT systems due to the comparatively small market size and huge costs involved in building these architectures. Additionally, the legacy application code must be modified to benefit from these architectures. The modifications results in significant development and re-certification costs.

Industry approaches the problem either by suspends all cores except the core on which an HRT task is scheduled or by considering overly pessimistic WCET of the HRT application[1]. While the industrial approach is cost effective, both industrial and academical approaches have tremendous performance penalty due to the inefficient resource utilization.

This thesis approaches the problem in a different way. We suggest none-to-minimal changes in the existing multi-core architectures to achieve time predictable execution of applications. Our approach does not enforce any modification in legacy code or existing single core timing analysis tools. Thus, our approach produces a cost effective, certification friendly and short time-to-market solution. Moreover, since we propose none-to-minimal modification in existing multi-core architectures, the high performance benefit of multi-core is preserved leading to truly *Predictable and High Performance Multi-core Architectures.*

[1] WCET of applications is often used for schedulability analysis

1.2 Goals of this thesis

The previous section has highlighted the need of a methodology to develop predictable and high performance multi-core architectures for Hrt and mixed critical systems[1]. This section summarizes main goals of the thesis. As stated in the title, the goal of the thesis is to provide predictable and high performance multi-core architectures. Since predictability in single-core architectures is well understood problem and methods [20] exist for Wcet analysis, this thesis exclusively focuses on multi-core architectures.

Our strategy of building a predictable architecture follows the definitions of predictability presented by Thiele *et al* [21] and Kirner *et al* [22]. By these definitions, we set our goals for a predictable architecture with the following virtues.

1. The upper bound on the produced Wcet of applications executing on the architecture should be close to the actual Wcet.

2. The required overall analysis efforts should be as minimum as possible.

3. The performance penalty of predictable architecture should be as minimum as possible. This has also been suggested by Schoeberl *et al* [23].

We extend the above defined virtues in [21, 22, 23] by the following additional virtues.

4. The design costs of such architectures should be as low as possible.

5. Wcet analysis of applications executing on these architectures should be simple and inexpensive.

6. The timing aspect of the combined system (hardware and software) should be certifiable[2] for the safety critical applications.

Our goal is to develop a multi-core architecture with all these virtues leading to a predictable, high performance and cost effectively certifiable solution.

[1]Systems executing applications with mixed levels of criticality.

[2]In this thesis, we consider certification as a formal proof of functional safety. Since the thesis focuses on the timing aspect, we provide, analysis, arguments and proofs which are acceptable by certification authorities for deterministic execution time.

1.3 Main contributions of the thesis

The contributions of this thesis are in building predictable multi-core architectures and analysis techniques surrounding the virtues presented in the previous section. The detailed contributions are as follows.

1. Invention of a novel arbitration scheme for shared resources in multi-core architectures which produces low WCET and increases resource utilization resulting in a faster execution in the average case. Additionally, the arbiter enables incremental certification (analysis in isolation) leading to a reduction in development and certification efforts.

2. Algorithms to estimate the worst case interference on a shared SDRAM under complex budget based arbiters. SDRAM promises inexpensive large data storage at high data rate which is essential in multi-core architectures. The presented algorithms are more precise than the existing techniques resulting in tighter bound on WCET (higher predictability).

3. Presentation of a technique to measure the WCET of applications executing on multi-core architectures. The technique does not enforce any modifications in production chips and single-core timing analysis tools. Thus, the technique leads to a low cost certifiable solution preserving the performance of multi-core architectures. Additionally, the technique allows chip vendors to hide details of their arbiter. This helps them keep their competitive edge.

4. Identification of a few caveats in building predictable multi-core architectures. Here, the concerns originate from low level micro-architectural details and apply only to multi-core architectures. Ignorance of these concerns may result in a wrong WCET estimation.

5. Proof of concept of the techniques on a real hardware using benchmark applications as well as use case demonstrations in real-world-like situations.

1.4 Structure of the thesis

The thesis is composed of eight chapters. The next chapter describes the related work in the area and distinguishes our work from the existing work.

The Chapter 3 presents a novel arbitration scheme, called Priority Division (PD), for multi-core architectures. The arbitration scheme has equal worst case performance as the TDMA arbitration, however, it has much better resource utilization resulting in *faster* execution in the average case. The chapter also proposes a single HRT capable variant of the PD arbiter which outperforms the highest priority master of *fixed priority* scheduling in the produced WCET.

The Chapter 4 presents timing analysis algorithms for complex arbiters under a complex shared memory (SDRAM). The presented algorithms improve precision of the produced bounds over existing techniques. Additionally, the chapter also compares unconventional and conventional arbiters in terms of predictability, performance and occupied chip area. The PD arbiter is found to be the clear winner in our tests.

The Chapter 5 presents a novel technique to measure WCET of applications executing on multi-core architectures using a single-core measurement based tools. The technique does not enforce any modifications to the existing production chips or tools making it an inexpensive, performance preserving and certifiable solution.

The Chapter 6 raises few points which must be taken care of while building a predictable multi-core architecture. These concerns are related only to multi-core architectures. Typically, they arise from very low level micro-architecture details which are generally abstracted by WCET analyzers. However, ignoring these concerns in multi-core architectures may lead to optimistic WCET estimation.

The chapter 7 presents three demonstrators built during this thesis work. The demonstrators highlight the usability of the techniques presented in above mentioned chapters in real-world-like situations.

Finally, the chapter 8 concludes our work with notes on future work.

Chapter 2

Related work

Achieving deadlines in Hrt systems is as important as producing a correct result after application execution. Hence, knowing the Wcet of applications on corresponding platform is of utmost important. Several methods exist for single-core architectures [20]. The shared resource interference is the biggest challenge for Wcet analysis in multi-core architectures [24]. Several micro-architecture modifications as well as analysis techniques are presented in literature. This chapter discusses the related work in existing micro-architecture components and analysis techniques.

2.1 Shared resource arbiters

Wcet analysis is yet immature on multi-core architectures and we believe that the industrial adoption will be incremental. In other words, from single-core, to dual-core, to quad-core and to many cores. Although our techniques are valid for Network on Chip (NoC) architectures, we present our analysis and tests on shared-bus-like architecture for the sake of simplicity. Sec. 3.1.2 describes how our techniques and analysis are applicable to a NoC.

Multi-core systems share system resources, e.g. main memory, to decrease package size and thereby, decrease cost of the product. An arbiter is employed to resolve conflicts among simultaneously arriving shared memory[1] access requests. Thus, the arbiter plays an important role in determining the maximum and the minimum latencies to access the shared memory as well as the shared memory utilization. Typically, low worst case latencies and memory utilization are mutually exclusive, although simultaneously desired, properties. Number of

[1]Throughout the thesis, we take shared memory as an example of shared resource since it is the most common shared resource and interference on it has significant impact on the produced Wcet.

arbitration schemes have been proposed to bring these desired properties together up to some extent. In this section, those schemes are discussed.

2.1.1 Close to traditional arbiters

Static priority (Sp– aka *Fixed priority*), Time Division Multiple Access (Tdma) and Round Robin (Rr) are simple and widely used conventional (traditional) arbiters. Their worst case latency and memory utilization analysis are presented in Sec. 3.1.5, Sec. 3.1.6 and Sec. 3.1.7, respectively. From the analysis it is clear that under Sp, only the highest priority master is guaranteed to have low worst case latencies while the lower priority masters have infinite worst case latency. However, memory utilization is ideal under Sp. Similar to Sp, Rr provides high memory utilization, however, at the cost of worst case latencies to all masters in the system. Tdma is the most predictable and provides low worst case latency to all masters, however, at the cost of low memory utilization. As stated in the previous section, low worst case latency and high memory utilization are simultaneously desired properties. Hence, new arbitration schemes are derived by mixing the traditional arbiters to achieve benefits of both the worlds.

Richardson *et al* [25] propose dTDMA arbitration scheme. Under dTDMA, slots of only active masters are inserted in the arbiter schedule. Hence, wastage of unused slots is prevented and memory utilization increases. However, when a master sends an access request, it is unaware of current slot schedule (due to dynamism). Hence, in the worst case, it has to assume that its request will be served in the last slot. This results in equally high worst case latencies as Rr.

Interference Aware Bus Arbitration (IABA) scheme is proposed by Paolieri *et al* [26]. We consider that the IABA arbitration scheme is a mix of Sp and Rr since Hrt masters have higher priority in accessing the shared memory. However, conflicting access requests from multiple Hrt are resolved by Rr. Here again, the worst case latency for multi-Hrt systems is equal to Rr.

The *Slot reservation* arbitration scheme is proposed by Poletti *et al* [27]. Under slot reservation, the critical master is assigned a reserved slot while other slots are dynamically arbitrated under the Rr scheme. Hence, we consider that the slot reservation arbitration scheme is a mix of Tdma and Rr. Here, if the reserved slot is not used then poor memory utilization is resulted, like Tdma. Additionally, only one critical master is supported and high worst case latency is expected in slots arbitrated under Rr.

We proposed Priority Division (Pd) [28, 1] as another "close to traditional" arbiter. We noticed that the Tdma is the most predictable and Sp is the most resource efficient. Additionally, low worst latency can be achieved for a selected master under the Sp. Hence, we chose Sp as a secondary arbitration to reuse the unused slots under the TDMA. If all masters utilize their slots, the PD and the TDMA behave exactly the same.

Thus, all arbiters derived from traditional arbiters behave exactly as their parent arbiter in corner cases. Recently, with specific goal in mind, complex arbiters are presented. They include randomized arbiters (for probabilistic WCET analysis) and budget based arbiters (for decoupling of latency and allocated rate).

2.1.2 Randomized arbiters

Lahiri *et al* [29] and Jalle *et al* [30] propose randomized arbiters. Under randomized arbiters, each master has certain probability of being granted an access when they send a request. Hence, for every access a probabilistic latency distribution can be achieved. For probabilistic WCET analysis, this distribution is required instead of the absolute worst case latency. RT_Lottery [31] (*Real-time Lottery*) mixes TDMA and lottery arbiters.

2.1.3 Budget based arbiters

Under traditional arbiters (except SP) as well as their derivatives (except PD in `hl` mode – Sec. 3.2.2), the low worst case latency can only be achieved by allocating high bandwidth. To remove this coupling, budget based arbiters are proposed. As the name suggests, instead of dividing memory bandwidth in number of slots, the masters are assigned a fixed budget of number of accesses in a unit time. The budgeting distributes the available shared memory bandwidth according to masters' memory requirements. Credit Controlled Static Priority (CCSP– Akesson *et al* [32]), Priority based Budget Scheduler (PBS– Stein *et al* [33]) and Multi-Bandwidth Bus Arbiter (MBBA – Bourgade *et al* [34]) are examples of budget based arbiters. Under these arbiters, the conflicts on shared memory are resolved by priorities. Here, a critical master is assigned high priority to achieve low latency access. Thus, the coupling between latency and allocated bandwidth is removed. Shreedhar *et al* propose to use budgets under RR arbiter in deficit round robin [35].

The maximum and minimum latency analysis under the budget based arbiters is complex. Here, latency to the current access depends not only on activity of co-existing masters, but also on the history of accesses.

The original PBS design in [33] had only two priority levels: high and low. In our previous work [36], we improved the design of the PBS arbiter by adding multiple priority levels and provided worst case interference analysis for each level. Additionally, we provide analysis for corner cases at replenishment period boundary. Tomlow *et al* [37] gives higher flexibility to the highest priority master and applies PBS scheduling to operating system. Here, processor is a shared resource.

2.1.4 Latency analysis under budget based arbiters

The budget based arbiters are complex arbiters for the worst case latency analysis since the worst case latency depends on the history of accesses done by the test master as well as the co-existing masters. To abstract details, Akesson *et al* [32] employ $\mathcal{LR}$ (Latency Rate) models [38] and Stein *et al* [33] employ dataflow models. These models model memory accesses in fluid manner. They assume that memory traffic can be divided into infinitesimally small units. However, shared memory accesses are served in a burst fashion. Hence, these models produce highly pessimistic results. The improvement is presented in Staschulat el al [39], Siyoum *et al* [40] and our previous works [36, 41]. These works take into account the bursty behavior of accesses. Lele *et al* [42] present a new dataflow modeling for TDMA arbiter which performs better than the $\mathcal{LR}$ model.

The above mentioned works, except [36, 41], have the following modeling limitations. i) They treat accesses from a core (master) independent of each other. In reality, cache misses are blocking[1] in nature. Here, in the worst case, service latency to one cache miss delays the occurrence of the next cache miss by the same amount. This delay must be incorporated in their models to correctly estimate worst case latencies. ii) The type of access is ignored. In reality, a read access to a shared memory takes longer to finish than a write access (Fig. 4.2). If the type of an access is ignored, in the worst case, it must be assumed that each access is of read type. This significantly increases the WCET.

2.1.5 Arbiter comparisons

Although a shared resource arbiter has a huge impact on the execution time predictability and resource utilization of a particular system, the impact of arbitration policy on WCET and resource utilization is a relatively less explored area. Pitter *et al* [43] and Kopetz *et al* [44] concluded that the TDMA arbiter is the most predictable arbiter among the traditional arbiters. After we introduced the PD arbiter in [28], Kelter *et al* [45] compared PD against RR and TDMA and concluded that the PD arbiter is an attractive candidate for mixed critical systems. Jalle *et al* [46] compared TDMA and RR arbitration scheme and concluded that the RR arbitration

[1]An out-or-order processor can execute instructions until a dependent instruction blocks the execution.

scheme is more predictable than the TDMA if the exact arrival time of a shared resource access is unknown.

In Chapter 4, we compare traditional arbiters, budget based arbiters and PD. This is the first work to compare different class of arbiters to study their impact on WCET of applications.

2.2 Shared resource interference compensated WCET analysis

Several interference aware (compensated) techniques are presented which can be roughly segregated in static analysis, measurement based hybrid analysis, co-existing application aware analysis and measurement based analysis in the presence of stress patterns. This section discusses related work in all these categories.

2.2.1 Static WCET analysis

The static WCET analysis is a widely used technique and number of commercial and academic tools are available for single core architectures, for example aIT[1], Otawa [47], Chronos [48]. In static WCET analysis technique, abstract models of application and underlying micro-architecture components (e.g. cache, pipeline etc) are created. Later, Integer Linear Programming (ILP) is applied with an objective to identify the maximum execution time. The technique is called abstract interpretation and Implicit Path Enumeration Technique (IPET).

The main advantage of the static analysis is, it gives formal guarantees of the worst case execution time and it is able to analyze timing anomalies and domino effects [49, 50]. However, the abstraction and the objective of maximizing execution time (ILP) produce significantly high WCET. The abstract modeling of micro-architectural components requires significant effort. This is particularly important since details of underlying hardware is vaguely disclosed by manufacturers to protect their competitive edge. This enforces the abstraction further, and results in increased WCET.

Our approach, as explained in Chapter 5, is based on measurement based technique where the targeted platform itself is used for measurements. Hence, we do not compare our approach to the static timing analysis. An interested reader is referred to Chattopadhyay *et al.* [51], Kelter *et al.* [52], Ding *et al.* [53] for the latest development in static WCET analysis for multi-core architectures.

[1]http://www.absint.com/ait/

2.2.2 Measurement based hybrid WCET analysis

The hybrid measurement based WCET analysis technique is also an industry standard technique and employed in commercial tools (for example RapiTime[1]). The approach was first presented by Kirner *et al* [54]. Here, execution traces are recorded while an instrumented[2] test-application is executing on the target platform. On the host machine (RapiTime), these traces are analyzed and the worst case path is constructed considering the Maximum Observed Execution Time (MOET) of individual basic blocks.

Due to the on target testing, WCET of applications executing on a reasonably complex platform can be analyzed. Additionally, due to the absence of abstract modeling and the ILP, tight WCET bound is produced. In its existing form, as explained in Sec. 5.1.1, the technique cannot be applied to multi-core architectures. In this thesis, we extend the technique to applications executing on multi-core architectures.

2.2.3 Holistic WCET analysis

In this analysis technique, complete knowledge of co-existing applications is considered to be known. Using this knowledge, maximum delay due to interference on the shared memory is predicted. One such approach using Abstract interpretation, Timed Automata (TA) and model checking is presented by Lv *et al* [55]. At first, abstract interpretation is used to determine occurrence time of each cache miss. This information is then used to build TA models of concurrently executing applications. The TA models of arbiter and applications are analyzed using a model checker to determine maximum interference and corresponding WCET. Pellizzoni *et al* [56] propose to use real-time calculus to determine the maximum interference between I/O traffic and the test-application. The arrival curve (memory access intensity) for I/Os is controlled and the arrival curve of application is achieved from a cache activity trace.

The holistic approach relies on absolute knowledge of simultaneously executing applications to tighten interference bound (and there by WCET). This approach is restrictive. A minor bug fix in a co-existing application enforce re-analysis of the whole system. In our approach, we analyze applications in isolation considering the worst interference from the co-existing applications. Thus, our approach yields, irrespective of co-existing applications, an absolute WCET.

[1]http://www.rapitasystems.com/products/rapitime

[2]Light weight instrumentation points are added at the beginning and end of each Basic Block – a linear code segment with single entry and single exit points. The instrumentation points time stamp the execution of basic blocks.

2.2.4 Measurements in the presence of stress patterns

As the name suggests, this approach measures execution time of a test-application in the presence of artificially generated interference. The artificial interference is typically generated by executing synthetic applications on co-existing masters. The synthetic application maintains a big data structure (much bigger than the cache) and accesses the data structure in strides bigger than the cache-line size. This makes sure that every access to the structure results in a cache miss. Thus, uninterrupted accesses (cache misses) towards memory is created by co-existing masters.

This technique does not need any change in hardware and hybrid Wcet analysis technique (Sec. 2.2.2) for single core architectures can be used in its existing form. However, as presented in Sec. 6.1.4 and Sec. 6.1.5, this approach produces unsafe Wcet bound.

2.2.5 Internal monitoring aided analysis

In order to estimate interference aware Wcet, internal monitoring approach is presented. Here, internal events, such as cache misses, are monitored during task execution. The worst case memory latency is then added to the observed execution time. The technique is presented by Nowotsch *et al* [57], Bin *et al* [58] and in our previous work [2]. The technique presented in [57] and [58] are coarse grain and produces higher Wcet bound. Additionally, as mentioned by Alhammmad [59], the technique presented in [57] needs synchronized task execution on different cores as well as knowledge of resource utilization and execution time of each co-existing application. The technique presented in [58] uses stressing benchmarks together with internal monitoring.

Compared to these techniques, our technique presented in [2] and Sec. 5 produces absolute Wcet and analyzes applications independent from co-existing applications. Additionally, our technique allows the chip manufacturers to hide internal specifications and still provide for the worst case interference compensation. This is highly attractive to both, chip vendors and OEMs.

2.3 Customized predictable architectures

Predictable execution time in the presence of caches and pipelines is difficult to achieve. The shared resource interference in mult-core architectures makes the problem even harder. There are two approaches to solve the problem [60]: i) Composability ii) Predictability.

According to Akesson *et al* [60], time composability means that an application path always takes the same amount of time, irrespective of co-existing applications. On the other hand, predictability means that execution time of an application path depends on the activity of co-existing applications, however, an *upper bound* on the execution time can be provided.

PRET [14] and CoMPSoC [16] are examples of composable (repeatable time) machines. PRET approach uses Scratch Pad Memory (SPM), TDMA arbiter and thread interleaved pipelines to achieve repeatable execution time [61]. Reineke *et al* [62] presented how the PRET approach can also be used in the presence of a shared SDRAM. A version of PRET processor executing ARM instruction set is also presented by Liu [63]. In summary, the PRET approach removes all the interference from the system. Instead of avoiding interference from the system altogether, the CoMPSoC [64] approach artificially delays the access to the shared memory by the theoretical worst case delay using delay blocks [65]. Goossens *et al* [66] present a technique to achieve composable memory response without using the delay blocks.

MERASA [15], parMERASA [67], ACROSS [17], RECOMP [18] and T-CREST [19] are predictable architectures. Unlike composable architectures, they do not focus on the repeatability of execution time. Instead, they allow upper bounded interference on shared resources. The upper bounded interference is achieved in MERASA architecture using either RR [68] or IABA arbitration [26] scheme. For WCET analysis, the arbiters can be configured in *wcet analysis* mode. In this mode, the arbiter artificially delays the access to the shared memory by the worst case. The parMERASA approach presents time predictable on-chip ring architectures to achieve composable WCET on a many core architecture [69].

The RECOMP project uses IDAMC NoC [70] to support mixed critical applications. At the router, two traffic classes are distinguished, according to their criticality (GT - Guaranteed Throughput and BE - Best Effort (latency sensitive)). The GT traffic is arbitrated under RR while the BE traffic is arbitrated by *winner-takes-all* arbitration (arbiter lock mechanism – Sec. 6.2.1). To manage smooth flow of packets in the NoC, back suction technique is used [71].

The ACROSS project employs time triggered NoC for time predictable access to the shared memory. Similar to ACROSS, the T-CREST project aims to build a time predictable NoC architecture [72, 73]. Hence, time-triggered static scheduling for routing NoC packets is employed [74, 75]. The static time-triggered schedule is expected to deliver poor resource utilization.

Apart from the above mentioned academic architectures, XMOS [76] and Propeller chip [77] are commercial predictable multi-core processors. The XMOS chip achieves predictable execution by removing all performance enhancing micro architectural components, for example, caches and pipeline. This significantly reduce the performance of XMOS chip. As stated in their overview [78], with 16 logical cores the peak performance is only 1000 MIPS. All cores share a main memory, however, the arbitration scheme is undisclosed. The propeller chip contains 8 cores and yields 160 MIPS. Although the processor cores are not detailed, we assume they do not contain caches and pipelines. An on-chip *hub* arbitrates shared resources in TDMA fashion.

To build a specialized chip only to achieve repeatable or predictable execution is very expensive, especially due to the small market size of automobile and avionics industry. Additionally, they severely degrade performance. In contrast, our techniques require tiny modification to existing multi-cores. Additionally, pin mapping is not at all affected, hence, the existing PCBs can be used without any modifications. Compared to customized architectures, our approach is significantly cost efficient and yields higher performance.

2.4 Probabilistic WCET analysis – pWCET

Typically, hardware components used in safety critical systems must give probabilistic guarantee of correct functionality, for example 1 failure per 10^9 hours of operation – failing probability 10^{-9} per hour. This means that the hardware may fail, however, the probability of the failure is very low.

The same argument is used by probabilistic WCET analysis approach. Here, instead of giving an absolute WCET value, probabilistic distribution of execution time is provided. From this distribution a cut-off is selected – execution time may exceed the cut-off by probability of 10^{-9} per XYZ number of executions. The approach was first presented by Edgar *et al* [79] and Bernat *et al* [80] and recently investigated by PROARTIS [81] and PROXIMA [82] (for multi-core architectures) projects. To achieve history independent random behavior, randomized cache [83] and randomized shared bus arbiter [30] are used. Since the focus of our work is in achieving the absolute WCET[1], we do not compare our technique with the probabilistic approach.

[1]The absolute WCET is the only *certifiable* technique right now.

Arbiter	SP	TDMA	RR	dTDMA	Slot reservation	Budget based	Random	**PD**
Hardware complexity	++	++	++	++	+	--	--	++
Analysis complexity	++	++	++	++	++	--	--	++
Absolute WCET	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
WCET Quality	++$^{1)}$	++	--	--	++$^{1)}$	++$^{1)}$	+$^{3)}$	++
Resource utilization	++	--	++	++	+	+	N/A	+
Certifiable	++$^{1,2)}$	++	+$^{2)}$	+$^{2)}$	++$^{1)}$	--	--	++

Table 2.1: Comparison of arbiters. *1) Only for the highest priority or reserved master. 2) The technique presented in Sec. 5 is required. 3) Probabilistic* WCET

2.5 Summary

This section summarizes the chapter and compares our approach with others on high level. Tables 2.1 and 2.2 list factors contributing to the virtues of predictable and high performance multi-core architectures presented in Sec. 1.2. It is clear that a single solution cannot be the best solution in all factors, however, our techniques perform better than the existing techniques in over all comparison.

Table 2.1 summarizes comparison between our PD arbiter and other arbiters. The PD has a small resource utilization penalty compared to *work conserving* arbiters (SP, RR and dTDMA). The working principal and latency/resource utilization of the PD arbiter is presented in Sec. 3.2.

Compared to existing latency analysis under budget based arbiters, our analysis presented in chapter 4 improves the analysis in the following ways. i) We precisely model each arbiter for the worst case behavior without using any abstraction. This increases the *one time* modeling efforts, however, significantly improves the precision. ii) We remove the unrealistic assumption of treating cache misses independently. Instead, we take into account the effect of service latency to one cache miss on the subsequent cache misses. iii) We model each shared resource access by its *type* for the worst case latency analysis to further improve the precision. iv) Instead of coarsely modeling the traffic towards shared resource, our detailed model captures burstyness and presents analysis on corner cases, for example, at replenishment period boundary. Thus, our approach significantly improves precision of the produced WCET compared to existing approaches.

Table 2.2 compares our technique presented in chapter 5 with the existing approaches. Our technique needs a minor change in the *test chips* of the commercial products. The modification is far less than the prohibitively expensive modifications required in custom architectures. Since our technique enables WCET analysis in isolation, it assumes the worst interference from the co-existing applications for each cache miss. Currently, this is the only inexpensive and certifiable approach, however, it produces larger WCET compared to the holistic approach and custom architectures. Although, our technique is certifiable, being measurement based analysis, it cannot analyze timing anomalies. It is clear from the table that benefits of our technique outweigh its limitations by far.

In summary, our techniques deliver predictable multi-core architecture at minimal hardware change, at minimal certification costs and at minimal reduction in performance compared to the existing approaches. Our techniques are in compliance with the relevant CAST-32 [3] paper guidelines for certification[1].

[1]Although the guidelines are currently valid only for a dual-core processor, this is the best information available at the moment. Additionally, we do not expect a huge change when the guidelines are released for multi-core architectures.

Arbiter	Static analysis	Custom Architectures	Probabilistic Analysis	Holistic Analysis	Measurements under stress patterns	Internal Monitoring	**Our Approach**
Change in hardware	++$^{1)}$	–	–	++	++	++	+$^{2)}$
Analysis complexity	–	++	–	-	++	-$^{3)}$	++
Absolute WCET	Yes	Yes	pWCET	Yes$^{4)}$	No	Yes	Yes
WCET Quality	–	++	pWCET	++$^{4)}$	N/A$^{5)}$	+$^{6)}$	+$^{7)}$
Certifiable	++	++	N/A	–	–	+$^{8)}$	++
Change in application code	++	–	–	++	++	–$^{8)}$	++
Certification costs	–	++	N/A	++	–$^{5)}$	–$^{9)}$	++
Performance impact	–$^{10)}$	–	-	++	++	–$^{11)}$	++
Analysis in isolation	++	++	++	–	++	+$^{3)}$	++
Change in WCET analysis tool	–	++	–	-	++	++	++
Timing anomalies analyzable?	++	++$^{12)}$	++$^{12)}$	–	–	–	–

Table 2.2: Comparison of interference aware WCET analysis approaches. *1) Complex architectures are not analyzable. 2) Minor change in test chip is required. 3) All cores must be synchronized, finishing time and resource usage of co-existing applications must be known. 4) Only valid for a particular set of co-existing applications. 5) No true* WCET*. 6) Overheads due to monitoring and suspension mechanism. 7) Penalty due to assumption of always worst interference. 8) All co-existing applications must implement monitoring and suspension mechanism. 9) Co-existing applications must also be certified for monitoring and suspension mechanism. 10) Due to high* WCET *values, inefficient scheduling is expected. 11) Resource wastage due to the suspension mechanism. 12) Timing anomalies not present.*

Chapter 3

Priority division arbiter

This chapter presents the Priority Division (PD) arbiter for sharing on-chip resources. The arbiter aims to provide latency guarantees equal to the TDMA arbiter and shared resource utilization more than the TDMA arbiter. The chapter also compares the performance of our arbiter with commonly used starvation free arbiters – Time Division Multiple Access (TDMA) and Round Robin (RR). The chapter also compares the performance of PD with the Static Priority (SP) arbiter[1].

3.1 Background

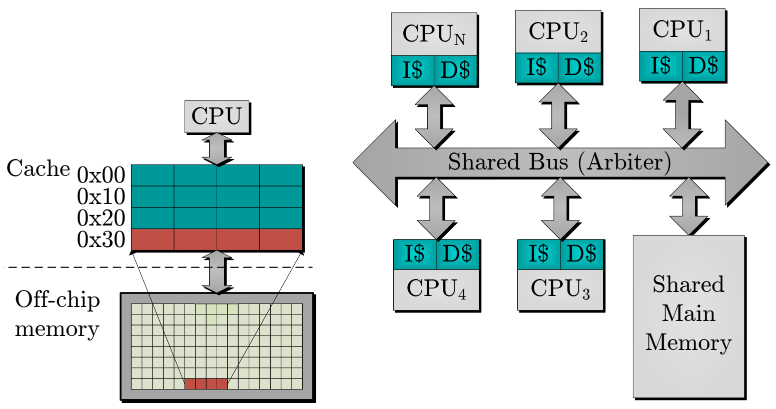
This section provides the necessary background information to facilitate the discussion presented in the chapter.

3.1.1 Shared resources in multi-cores

Shared on-chip resources are often employed in COTS based multi-core architectures to reduce number of components and smaller chip/package size. The smaller size drives the cost down. A basic multi-core architecture is depicted in Fig. 3.1(b) [1]. The figure depicts a shared main memory which is one of the most commonly used shared resource.

The figure shows N number of CPU cores connected via a shared bus to the main memory. The cores employ instruction and data caches. The cores usually access data and instructions from caches. If the required instruction or data is not present in caches (cache miss), an access to the shared main memory is requested. As depicted in Fig. 3.1(a), data within a cache is

[1]The chapter includes contents from our previous work [1]



(a) Cache-line Mapping (b) Basic Multi-core Architecture

Figure 3.1: Cache-line Mapping and Basic Multi-core Architecture.

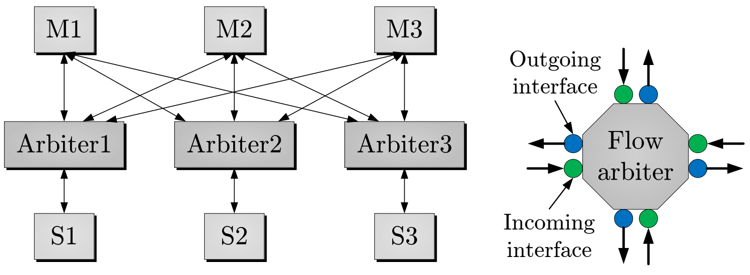
stored in cache-lines. The incoming data from main memory replaces one of the cache-lines (eviction). The evicted cache-line is written back to the main memory (write back policy). Accesses to the main memory are issued in burst fashion to speed-up the operation. Thus, the application being executed on the core, the core's cache configuration and eviction policy, all, influence the main memory access pattern of any core.

3.1.2 Employment of arbiters in a NoC

The Fig. 3.1(b) depicts a shared-bus-like architecture for the sake of simplicity. In reality, our test-architecture is essentially a NoC with a Crossbar [84] topology as presented in Fig. 3.2(a). This architecture is also called *slave side arbitration* [85] since each slave is equipped with its own arbiter. Here, masters (M1, M2, ...) can communicate with individual slaves (S1, S2,...) in parallel which significantly improves interconnect throughput. However, if multiple masters want to communicate with a unique slave, the respective arbiter resolves conflicts.

A multi-core architecture, generally, contains only one shared main memory and interference on it has the maximum impact on the WCET. Hence, in order to absolutely focus on this interference, we consider that the shared memory is the only slave in the system. This assumption leads to an identical (shared-bus-like) architecture as presented in the Fig. 3.1(b).

A router node of a NoC is presented in Fig. 3.2(b). Here, the router is responsible for avoiding conflicts if multiple incoming flows want to use the same outgoing interface. The outgoing interface is now the shared resource, hence, analyses and techniques developed in this thesis are applicable.



(a)
Crossbar NoC
(Our true implementation)

(b)
Router node
of a NoC

Figure 3.2: Employment of arbiters in NoC

3.1.3 The shared resource access latency and utilization

The single bus-slave interface of Fig. 3.1(b) can serve only one master in a particular clock cycle. Typically, an arbiter guards the shared memory and grants only one master to access the memory. If another master wants to access the memory at the same time, it has to wait. Thus, latency to access the shared memory is influenced by: **i)** Operating speed of the memory itself – the slower the memory the longer it takes to access it, **ii)** The policy by which the shared memory is arbitrated, and **iii)** The access patterns of co-existing cores when a request to the shared memory is raised.

The speed of the memory and the employed arbitration policy is known, however, the access patterns of co-existing cores is cumbersome to predict. Hence, the best case and worst case scenarios are analyzed to achieve the best case latency (B_L) and the worst case latency (W_L). In the best case, the access-under-investigation is not interfered while in the worst case, it is interfered maximally.

Note that the execution on a core experiencing a cache miss is suspended until the required data is fetched from the main memory. Thus, the B_L and W_L parameters must be considered while determining the Best Case Execution Time (BCET) and the Worst Case Execution Time (WCET), respectively. During the BCET estimation, all favorable scenarios are assumed: no interference at all. If an arbiter wastes clock cycles in all favorable scenario then inefficient shared resource utilization is resulted. Thus, B_L is higher in case of non-work-conserving arbiters compared to work-conserving[1] arbiters. A low BCET indicates high shared resource utilization, Ψ.

[1] A work conserving arbiter grants the shared resource to the requesting master immediately if the shared resource is idle.

From the above discussion, we define the shared resource utilization, under all favorable conditions, as in Equation (3.1). Here, m is the master (or core) on which the test-application is executing. Under the assumption of all favorable conditions, only the test master is backlogged ($\beta_m = 1$), all co-existing masters are idle ($\beta_{m' \neq m} = 0$).

$$(\beta_m = 1) \wedge (\beta_{m' \neq m} = 0) \ \mid \ \Psi = \frac{\eta_{busy} \times 100}{\eta_{busy} + \eta_{idle}} \% \tag{3.1}$$

In the above equation, the right side of "|" is only valid, if the condition on the left side of "|" is true. Here, η_{busy} denotes the number of clock cycles utilized to serve the access request from m while η_{idle} denotes the number of unused clock cycles after the access request is issued. Considering a constant shared resource access pattern (cache miss pattern) of the test-application and a constant memory bandwidth, B_L, W_L and Ψ are only influenced by the arbitration policy. This chapter focuses on the effect of the arbitration policy on B_L, W_L and Ψ.

Asymmetric multiprocessing: In order to focus exclusively on the shared resource interference, we assume asymmetric multi-processing[1]. For symmetric multi-processing, our analysis must be augmented by task dependency analysis which is left for future work.

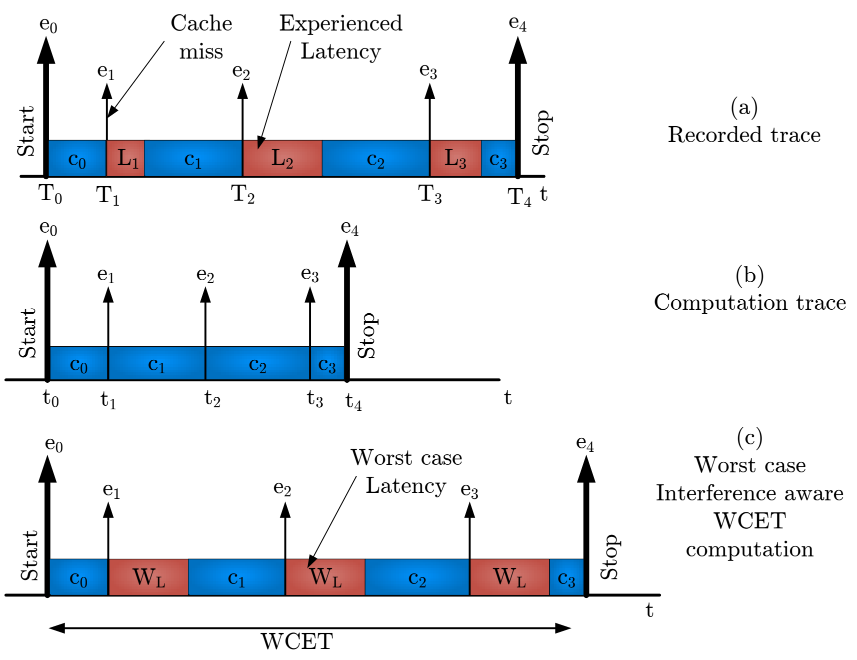
3.1.4 Computation trace

Before we study the arbitration policies in detail, this subsection explains the computation trace as described in our previous works [86, 1].

As explained in Sec. 3.1.1 a shared main memory is, unless informed explicitly, accessed only if a cache miss occurs. In asymmetric multi-processing, interference on the shared resource is the only way co-existing applications impacts the execution time of each other.

A computation trace is a convenient way to estimate either WCET, ACET– Average Case Execution Time or BCET of a particular application path taking into account the worst case, the average case or the best case interference (no interference at all), respectively. In computation trace, cache misses on the application path are denoted by timeless events ($e_0, e_1, ...$ in Fig. 3.3 [86, 1]). These timeless events are separated by the time during which the core executing from registers, caches etc. These times are denoted by computation times, $c_0, c_1, ...,$ in the figure. It is now fairly convenient to append each event by either B_L, A_L (Average case latency) or W_L to achieve BCET, ACET or WCET, respectively. Again note that the BCET,

[1]In asymmetric multi-processing, cores execute independent applications. Although, the physical memory device is shared, the data on the device is unshared.



W_L = Worst case latency, L_x = Measured latency, c_x = Computation time,
e_x = x^{th} event, T_x = Time in recorded trace, t_x = Time in computation trace

Figure 3.3: WCET estimation using the computation trace

ACET and WCET estimated using computation trace are particular to that application path and considering the shared memory interference, only.

Estimating BCET, ACET or WCET is now a simple three step process. i) The test-application path is executed on the target platform (Chapter 5) or simulation platform and cache miss trace together with the experienced latencies are recorded – recorded trace. ii) All cache miss place holders $(e_0, e_1, ...)$ are shifted to the left by removing experienced latencies $(L_0, L_1, ...)$. iii) Each cache miss event $(e_0, e_1, ...)$ is appended by either B_L, A_L or W_L, depending on the estimation goal, shifting all subsequent events to the right. The estimated BCET, ACET or WCET have now correct impact of interference, respectively.

Especially for WCET estimation, we must show that insertion of W_L for each cache miss is a conservative assumption also in the case of out-of-order execution. For example, an event occurs at time t in a computation trace. Instead of blocking immediately, the execution moves forwards for ϕ number of clock cycles until a dependent instruction stalls the execution. Had the cache miss experienced the worst case latency, the effective execution blocking time is $W_L - \phi$. Instead, using computation trace, we assume that the blocking time is $W_L > (W_L - \phi)$. If

$\phi \geq W_L$, the processor has sufficient instructions in pipe-line so execution will not be blocked at all. Thus, inserting W_L for each cache miss event and assuming suspension of execution for W_L is a conservative assumption.

It is clear that the WCET estimated in this fashion is the WCET of that particular path only. Extensive testing and measurement based WCET analysis tool, as presented in Chapter 5, must be used to find the critical path and the WCET of application.

3.1.5 Analysis of the Static Priority Arbiter

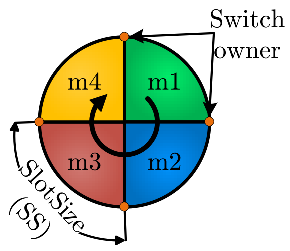
Under the Static Priority (SP) or fixed priority arbitration, each master is assigned a fixed priority. As soon as requests to access the shared memory arrive, they are scheduled according to the priorities of originating masters. In this thesis, without loss of generality, we assume that the SP is configured in a non-preemptive fashion. Here, higher priority master cannot preempt the ongoing lower priority burst. However, after `BurstLength` number of transfers of a lower priority master are finished, a new arbitration decision based on priorities is made. Thus, the highest priority master has to wait, in the worst case, for `BurstLength` number of lower priority transfers to be finished. We denote this number clock cycles as `SlotSize` – SS.

Upper bound and lower bound on latency Since the highest priority master may issue uninterrupted burst accesses and monopolize the shared memory, the worst case latency bound for co-existing lower priority masters is infinite. As explained above, in the worst case, the highest priority master must wait for one ongoing lower priority burst transfer. After waiting, its own access will be scheduled and finish within SS clock cycles. Thus, in the worst case an access request from the highest priority master completes in $2 \times SS$ clock cycles. In the best case, the shared resource is granted to the highest priority master immediately and the access completes after SS clock cycles.

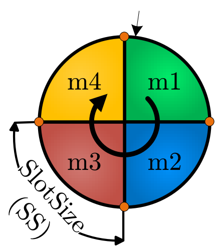
$$W_L^{sp} = 2 \times SS \tag{3.2}$$

$$B_L^{sp} = SS \tag{3.3}$$

Estimating WCET using computation trace needs only appending $W_L = W_L^{sp}$ for every cache miss in a computation trace. The SP arbitration is an ideal candidate for single HRT system. A carefully implemented HRT application will not starve co-existing lower priority applications.



(a)
Graphical view
of TDMA



(b)
Arrival of a late
request from m1

Figure 3.4: The TDMA Arbiter.

In the event of a fault in HRT, starvation to co-existing *any-time* applications may not be too much important since catastrophic consequence is expected.

Shared memory utilization: The SP arbiter schedules an incoming request based on their priorities without wasting a single clock cycle. Thus, $\Psi^{sp} = 100\%$.

3.1.6 Analysis of Time Division Multiple Access (TDMA) arbiter

The operation of the TDMA arbitration is graphically presented in Fig. 3.4(a) [1]. Each core gets a fixed timing window to access the shared memory, called a slot. The schedule of slots (`BusCycle`) repeats itself, like a virtual ring, after reaching the last master. Each slot is sufficiently big to allow a single cache-line refill. Each slot begins with a switch point. If an access request is pending from the owner of the slot then the requesting owner (master or core) will be scheduled in the slot. Otherwise, the slot will be wasted. The simple time based switching of owners removes interference from the system and co-existing applications cannot impact execution time of each other anymore. Thus, irrespective of co-existing applications, the execution time deviates by only ±`BusCycle`.

Upper bound and lower bound on latency Under TDMA arbitration, latency to any access depends on the arrival of the request compared to the beginning of the slot of the originating master. As depicted in Fig. 3.4(a), let us assume N masters in the system. Then, the experienced latency L_i for the i^{th} access is calculated by equations (3.4) and (3.5).

$$\overline{t_i} = N \times SS - \{c_{(i-1)} \bmod (N \times SS)\} \tag{3.4}$$

$$L_i^{tdma} = \begin{cases} \overline{t_i}, & \overline{t_i} \geq SS \\ \overline{t_i} + (N \times SS), & \overline{t_i} < SS \end{cases} \tag{3.5}$$

At first, the remaining time in the `BusCycle` is computed using the equation (3.4). The equation uses the time gap between two cache misses, c_i. If $\overline{t_i} < SS$, too less time is left in the slot and the access will occupy initial part of the next slot, hence, it will not be scheduled. Now, it has to wait for the new `BusCycle`. We assume that the test-master has the last slot in the `BusCycle`. So, the access has to wait for the entire new `BusCycle` until it is scheduled, as derived in equation (3.5).

Although the L_i^{tdma} value is bounded by $SS \leq L_i^{tdma} \leq (N+1) \times SS$, the bounds are not required. According to equations (3.4) and (3.5), the value of L_i^{tdma} does not depend on activity of co-existing applications and depends only on the constant parameters. Hence, execution time of paths can be measured directly and the highest one is considered as the WCET of the application.

Shared memory utilization: The predictable execution and the comfort of being able to directly measure the WCET without interference analysis takes it toll on the memory utilization. Here, unused slots are wasted. Assume that an access request arrives one clock cycle after the slot dedicated to the originating master has started, as depicted in Fig. 3.4(b). Since the access cannot complete in the dedicated slot, the current slot will be wasted. Additionally, for estimating utilization, we assume that co-existing masters are idle (equation (3.1)). Hence, other slots will not be utilized as well, resulting in huge loss of memory bandwidth and unnecessarily high latencies.

Under the assumption of idle co-existing masters, if an application does majority of accesses just after its slot has started then the worst case memory utilization occurs which is given by the following equation.

$$\forall i, c_i = (N-1) \times SS + n \times SS + 1, n \in \mathbb{N}^0 \;\mid\; \Psi_w^{tdma} = \frac{100 \times SS}{(N+1) \times SS} = \frac{100}{N+1}\% \qquad (3.6)$$

Similarly, best case memory utilization occurs if an application does all accesses right before its slot starts.

$$\forall i, c_i = (N-1) \times SS + n \times SS, n \in \mathbb{N}^0 \;\mid\; \Psi_b^{tdma} = 100\% \qquad (3.7)$$

A Round Robin (RR) arbiter can be used to increase the memory utilization. The following subsection describes the RR arbiter.

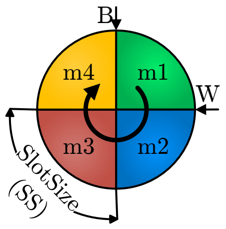


Figure 3.5: Graphical View of the Round Robin Arbiter.

3.1.7 Analysis of round robin arbiter

The operation of the Round Robin (Rr) arbiter is depicted in the Fig. 3.5 [1]. Similar to Tdma, masters are allocated number of timing windows (slots) to access the shared memory. Unlike Tdma, the starting time of slots are not fixed. Instead, the arbiter is always looking for a requesting master in a circular manner. As soon as a requesting master is found, its slot is started. The `SlotSize - SS` is fixed. Thus, after SS number of clock cycles, the arbiter resumes looking for a requesting master from the next slot in the circular schedule. Thus, if a single master is requesting, the memory stays occupied which increases the memory utilization.

Upper bound and lower bound on latency Due to the dynamic slot start time, the worst case latency is high under Rr. Assume that our test-master (`m1`) issues a request when the arbiter pointer is at **W** in Fig. 3.5. Additionally, assume that all co-existing masters also send requests at the same time. Hence, all slots will be occupied and the request from `m1` needs ($W_L = 4 \times SS$) to complete. This is the worst case. Similarly, the best case latency occurs if `m1` sends a request when the arbiter pointer is at **B**. In this case, the access request will be served immediately, hence, $B_L = SS$. Since state of arbiter pointer and activity of co-existing master are unknown, latency to each access must be considered as W_L for the Wcet analysis.

$$W_L^{rr} = N \times SS \tag{3.8}$$

$$B_L^{rr} = 1 \times SS \tag{3.9}$$

Shared memory utilization: As explained above, under the Rr arbitration, memory is occupied if at least one access request is backlogged. Hence, $\Psi^{rr} = 100\%$.

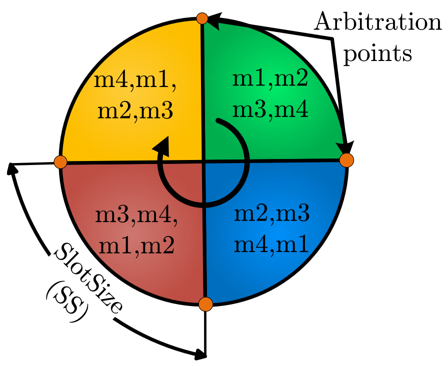


Figure 3.6: Graphical View of the Priority Division Arbiter.

3.2 Priority Division (PD) arbiter and analysis

This section describes the priority division (PD) arbiter. The arbiter was first published in our previous works [28, 1]. The arbiter takes advantage of high memory utilization of the SP arbiter and predictability of the TDMA arbiter.

3.2.1 Basic operation

The operating principle of the PD arbiter is graphically presented in the Fig. 3.6 [1]. Similar to TDMA, PD employs static time slots to grant masters an access to the shared memory. Unlike TDMA, slots are not owned by an exclusive master. Instead, masters are assigned a priority in every slot. At beginning of slots (arbitration points in the figure), the highest priority requesting master is granted to access the shared memory for `SlotSize` number of clock cycles. Thus, if the highest priority master does not want to access the shared memory then, instead of wasting a slot, a lower priority master is given an opportunity to access the shared memory. This increases memory utilization at a minor increase in complexity.

The priorities of masters are changed from slot to slot according to the required distribution of the memory bandwidth. For example, to prevent starvation, each master *must* have highest priority in at least one slot. Masters are depicted the descending order of priority.

Compared to the traditional arbiters, the PD arbiter offers more configurability. Here, number of slots and priorities within the slots can be tuned in order to achieve the design objectives. For example, a memory intensive master can be assigned highest priority in one slot and the second highest priority in all slots. So, it is the first to benefit from the unused bandwidth of co-existing masters.

Upper and lower bound on latency: Under the PD arbitration, the worst case occurs for a test-application if it does not benefit from unused slots of co-existing masters. Note that the test-application, irrespective of activity of co-existing masters, can access the shared memory in the slot where it has the highest priority. This is the same behavior as the native TDMA. Thus, the worst case latency under PD is equal to the experienced latency under TDMA.

$$W_{Li}^{pd} = L_i^{tdma} \tag{3.10}$$

Measuring WCET under PD is simple. The co-existing applications must execute synthetic stress pattern to logically convert the PD into the TDMA. Now, Observed Execution Time (OET) is the WCET, like under the TDMA.

The best case occurs when co-existing masters are idle. So, the test-master is able to use the slots when needed. However, unlike RR, the test-master must wait for an arbitration point.

$$B_{Li}^{pd} = (c_{(i-1)} \bmod SS) + SS \tag{3.11}$$

Shared memory utilization: As explained above, the unused slot of the highest priority master is re-arbitrated and can be used by a lower priority master. This increases memory utilization compared to TDMA. However, unlike RR where slot of a requesting master is immediately started, under PD, a new master is scheduled only at arbitration points. In the worst case under PD, the test-master sends request just after an arbitration point. Thus, the current slot is wasted and the request is served in the next slot. This results in 50% memory utilization in the worst case.

$$\forall i, c_i = n \times SS + 1, n \in \mathbb{N}^0 \;\mid\; \Psi_w^{pd} = \frac{100 \times SS}{2 \times SS} = 50\% \tag{3.12}$$

From equation (3.6) and (3.12), $\Psi_w^{pd} >> \Psi_w^{tdma}$.

Similar to TDMA, if an access request from the test-master arrives exactly at an arbitration point, the best case memory utilization occurs.

$$\forall i, c_i = n \times SS, n \in \mathbb{N}^0 \;\mid\; \Psi_b^{pd} = 100\% \tag{3.13}$$

3.2.2 h1 Configuration

In the native mode, PD is a *starvation free* arbitration policy and can support multiple HRT. However, in a single HRT system, PD can be configured in `h1` mode to minimize latencies to the critical master. Under the `h1` configuration, the critical master has the highest priority in all slots. Thus, an access request from a critical master is guaranteed to be scheduled at arbitration points, irrespective of activity of co-existing masters. This results in worst case latency = best case latency = observed latency.

$$L_i^{h1} = (c_{(i-1)} \bmod SS) + SS \tag{3.14}$$

Due to the static slot allotment, the memory utilization under the `h1` mode is less than the SP. However, latencies to critical master is less the SP, as derived in equations (3.2) and (3.14).

3.3 Comparison of the arbiters

This section compares the PD arbiter with the traditional arbiters, SP, TDMA and RR. The first test compares PD against *starvation free arbiters* – TDMA and RR and the second test is dedicated to single HRT capable arbiters – SP and PD in `h1` mode. The tests study the impact of arbitration policy on: i) WCET of applications, ii) memory utilization and iii) chip area.

3.3.1 Test setup

Our test architecture is similar to the one illustrated in the Fig. 3.1(b). We built a quad-core processor on Altera Cyclone III FPGA using Altera NIOS II cores. Each core contains instruction and data caches of size 512 Bytes, each. The cores share an on-chip SRAM under the above mentioned arbitration schemes. In this chapter, we selected single path[1] applications from the Mälardalen WCET benchmark suit [87]. To prevent direct and indirect memory monopolizing and to prevent invalidation of our latency parameters (B_L and W_L), we took care of caveats presented in Chapter 6.

The test-applications are executed on the core1 (`m1`) while the co-existing cores (`m2`, `m3` and `m4`) execute dummy shared memory stressing applications. The measurements of memory utilization, Ψ, was directly conducted in the target platform by observing internal request and grant signals.

[1]Multi-path applications need path analysis in addition to interference analysis. This is presented in Chapter 5.

Benchmark	TDMA		RR		PD	
	Ψ in %	OET	Ψ in %	WCET	Ψ in %	WCET
compress	30.37	26591	100	30506	71.27	26591
cover	31.56	15805	100	18024	71.97	15805
crc	30.03	106013	100	109163	67.49	106045
duff	34.49	4920	100	5281	76.28	4920
edn	33.62	494584	100	553972	72.04	494584
expint	32.65	16472	100	16708	74.42	16472
fac	30.19	1176	100	1240	70.07	1176
fdct	45.65	21918	100	31837	70.28	21918
fibcall	35.16	1150	100	1228	74.42	1150
fir	32.26	2005692	100	2225970	66.65	2005660
jane	30.35	1016	100	1108	76.19	1016
jfdcint	36.02	31486	100	37035	71.86	31390
matmul	31.01	1633112	100	1764383	67.03	1633048
minver	32.53	161624	100	191270	70.92	161624
ludcmp	34.29	371320	100	456766	69.56	371352
prime	32.58	180831	100	200651	78.17	180990
quart	33.97	223896	100	270686	71.46	223800
recursion	33.33	6813	100	6898	72.72	6813
ud	33.17	40247	100	48212	72.03	40247

Table 3.1: Execution time in clock cycles and memory utilization in %: TDMA vs RR vs PD. Source: [1]

3.3.2 Starvation free arbiters

Under TDMA arbiter, as explained in Sec. 3.1.6, the execution time remains constant (±`BusCycle`) irrespective of activity of co-existing masters. Hence, OET = WCET = BCET. Under the RR arbiter, the technique presented in Sec. 3.1.4 was used for WCET analysis. As explained in Sec. 3.2, the WCET under the PD arbiter was measured in the presence of the shared memory stressing traffic generated by the co-existing masters.

In the Table 3.1, Ψ values are presented in %. The WCET values are presented in number of clock cycles. From the table, it is clear that the RR produces the highest WCET. Unsurprisingly, this is due to the fact that in the worst case, the test-master could be the last to be served for each of its shared memory access. The TDMA and PD produce equal WCET. However, due to the re-arbitration of unused slots under PD, the memory utilization is more than twice that under TDMA.

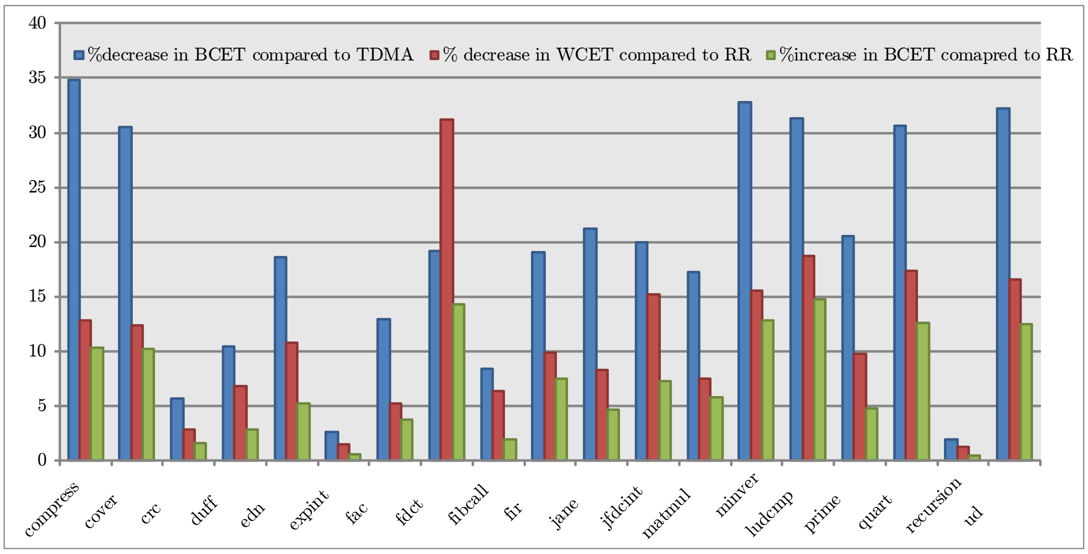


Figure 3.7: Improvement over TDMA and Round robin arbiters.

The advantages and drawbacks of using PD over TDMA and RR are clearly visible in the Fig. 3.7 [1]. Due to higher memory utilization under PD, applications execute faster in the average case. This is shown by the first bar in the figure which depicts reduction in BCET compared to TDMA. Similarly, due to the static *arbitration points*, WCET is reduced compared to the RR. This is presented by the second bar. The third bar depicts the drawback of PD compared to RR. Since RR produces the highest memory utilization, the applications execute fastest in the average case under RR. The third bar shows increase in BCET under PD compared to RR. However, the drawback of PD– the third bar, is the smallest for all applications.

3.3.3 Single HRT capable arbiters

This test compares single-HRT capable arbiters: PD in `h1` mode and SP. The test is valid for only the highest priority master (HRT executing master). For all lower priority masters, the worst case latency and hence the WCET is infinite.

WCET values in clock cycles under SP and PD in `h1` mode are presented in the Table 3.2. Under SP, the highest priority master must consider interference from at least one lower priority access. Instead, under `h1`, the highest priority master is scheduled at arbitration points irrespective of activity of co-existing masters. Hence, under `h1`, OET = BCET = WCET. From equations (3.2) and (3.14), $W_L^{sp} > L_i^{h1}$. This fact is reflected in the results presented in the

Benchmark	Static Priority WCET	PD^{h1} WCET
compress	20970	17327
cover	12537	10981
crc	103231	100021
duff	4608	4408
edn	439203	402368
expint	16210	16048
fac	1072	1024
fdct	20649	17710
fibcall	1098	1054
fir	1748622	1624372
jane	872	800
jfdcint	28008	25190
matmul	1444996	1352416
minver	126731	108704
ludcmp	296187	255024
prime	157944	143790
quart	180263	155416
recursion	6730	6685
ud	31993	27271

Table 3.2: Execution time in Clock Cycles: SP vs PD in `h1` mode. Source: [1]

table where WCET produced under `h1` is less than the WCET produced under SP for the most critical master.

The advantage of employing `h1` instead of SP is presented by the first bar in the Fig. 3.8 [1] (reduction in WCET). However, in the average case, the SP results in faster execution of applications due to its high memory utilization. This fact is reflected by the second bar in the figure where BCET increases under `h1` compared to SP. However, the drawback of `h1` – the second bar, is the smallest for most of our test applications.

Fig. 3.7 and Fig. 3.8 proves that the PD and its derivative `h1` configuration provides the best trade-off between predictability and performance (in terms of average case execution).

3.3.4 Area overheads

We synthesized the arbiters for Cyclone III FPGA @ 125 MHz clock frequency. The consumed chip area, in terms of Logic Elements (LE), is listed in the Table 3.3. The arbiters have similar

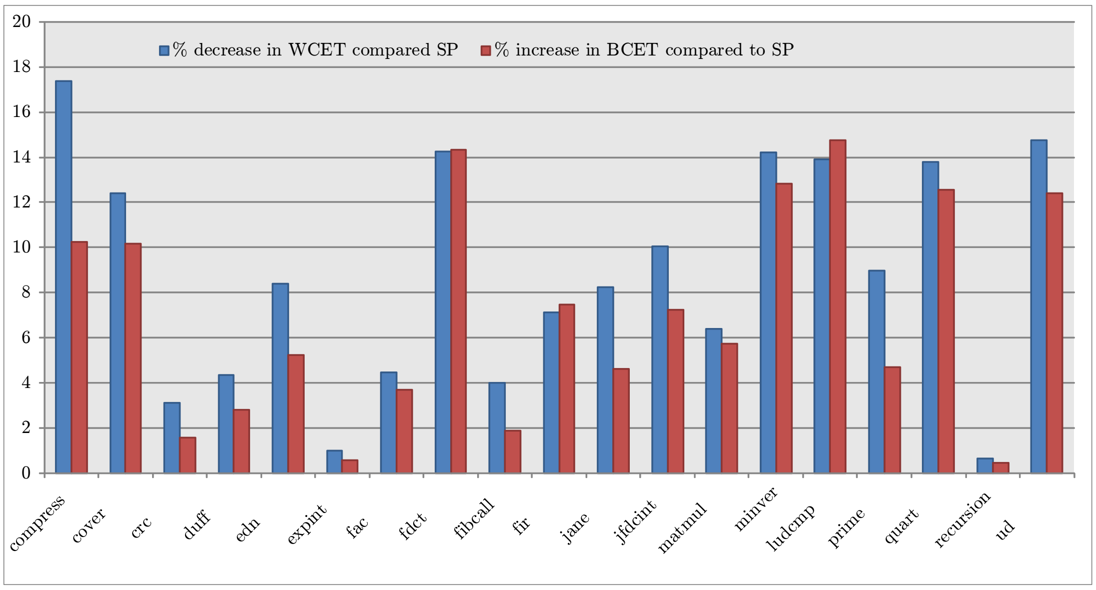


Figure 3.8: Improvement over SP arbiter.

Arbiter	SP	TDMA	RR	PD
Number of Logic Elements	281	277	288	285

Table 3.3: On-chip area overheads of the arbiters. Source: [1]

footprint on chip area due to their functional similarities. The RR is slightly bigger due to the complexity associated with dynamic slot management. The TDMA is smallest due to statically defined slots. The PD is slightly larger than TDMA due to priority management on top of static slots. The SP is also slightly larger than the TDMA due to the dynamic scheduling. However, all arbiters consume negligible chip area compared to the entire multi-core system.

3.4 Comments on certification

According to the MCP_Determinism_7 of the CAST-32 paper (see Sec. A.1), interference channels between software executing on different cores must be known and mitigated. We have shown theoretically and through experiments that the PD behaves, in the worst case, exactly as a strictly time-partitioned arbiter (TDMA) where one core cannot interfere with the other core while accessing the shared resource. One core can only help co-existing cores run faster by giving away its slot under the PD arbiter. Thus, it complies with the MCP_Determinism_7.

The MCP_Determinism_9 demands avoidance of shared memory monopolizing. In the basic mode of the PD arbiter, each master has one slot with the highest priority which is guaranteed to the master if it requests. Hence, the arbiter is *starvation free.* For asymmetric multi-processing (only physical shared memory device, no logically shared region), it is sufficient to use a starvation free arbiter in order to comply with MCP_Determinism_9.

For symmetric multi-processing, co-running tasks on different cores must be synchronized, for example by means of a mutex or message passing[1]. Avoiding a deadlock while synchronization is well researched problem from multi-threaded applications and tools are available, for example [90], to detect such conditions during application analysis. An automated technique to avoid dynamic deadlocks in legacy code is presented in [91]. Research on the deadlock avoidance while synchronization is out of the scope of this thesis.

According to MCP_Determinism_12, the memory usage of applications must be known and must stay below or equal to the available memory bandwidth. Here, computation trace (Sec. 3.1.4) can be used to determine memory requirements of applications. Computation of available bandwidth under shared SRAM (as in this chapter) is straightforward. Computation of worst case memory bandwidth under complex memory, such as SDRAM, is presented in Chapter 4.

The theoretical proofs and experiments to analyze the worst case interference presented in this chapter go inline with the suggested activities: MCP Interference Channels (4.(a), 4.(b) and 5.(a) see Sec. A.1) to demonstrate compliance with the safety standards.

3.5 Summary and future work

This chapter has introduced a novel arbitration scheme called priority division. The arbitration scheme is superior in the shared resource utilization compared to the TDMA and produces equal (much less than the Round robin) WCET bound as the TDMA. This makes it an arbiter of choice where multiple HRT applications exist and performance (in terms of memory utilization) is also critical. The arbiter can also be configured to support a mixed criticality system with a single HRT application. We call this configuration as `h1` configuration which stands for only one HRT application. Under the `h1` configuration, the PD produces even lower WCET bound than that produced under the static priority arbitration. This makes the PD an ideal arbiter for mixed critical system with single HRT application.

[1]For multi-core processors, a hardware mutex core [88] or hardware mailbox core [89] must be used.

The certification comments provided in the chapter show how the timing aspect of the PD arbiter is, in principle, certifiable for avionics.

The design space exploration based on the secondary priorities is an interesting research direction for future work. Here, using programmable secondary priorities and hardware-in-the-loop test platform can be developed. After each test, according to optimization criteria, secondary priorities can be re-programmed for the subsequent test. The process goes on until the optimization goals are reached. This is very helpful to a system integrator who has complete information about all the applications mapped to different cores.

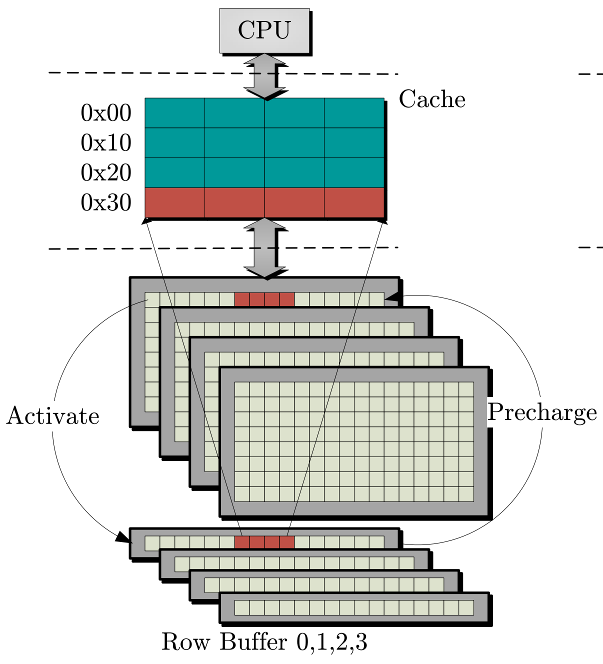
Chapter 4

Sharing an SDRAM with budget based arbiters

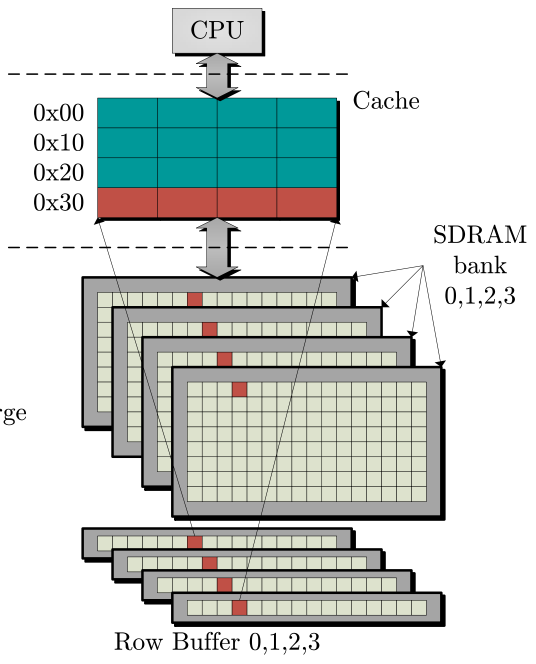
The previous chapter has presented the worst case access latency analysis under Static Priority (Sp), Time Division Multiple Access (Tdma), Round Robin (Rr) and Priority Division (Pd) arbiters. A shared Sram was used a shared memory. Sram is widely used in safety critical systems. However, usage of an Sdram is widespread in general purpose systems due to its low cost and small size. Safety critical hard real-time systems can greatly benefit by using Sdram as a main memory since it provides a large data storage and high data rate at low cost. The large data storage aids in implementing enhanced safety functionalities.

Performing the worst case latency analysis on a shared Sdram is complicated. Additionally, budget based real-time arbiters have been presented for shared resources in research e.g. Credit Controlled Static Priority (Ccsp) and Priority-based Budget Scheduler (Pbs) which further complicate the analysis. Unlike Tdma, Rr and Pd, these arbiters do not employ a circular wheel based scheduling policy. Instead, each master is assigned a number of accesses (budget) in a particular time frame, popularly called *replenishment period.* Access conflicts within a replenishment period is resolved by priorities.

An Sdram shared under these arbiters is challenging for the worst case latency analysis. This chapter focuses on this issue and provides algorithms to predict the worst case access latency and thereby the Wcet of applications executing on such multi-core architectures. The chapter includes contents from our previous works [36, 41] and improves analysis by considering corner cases at the replenishment period boundary. Additionally, shared Sdram latency analysis under Sp, Tdma, Rr and Pd and comparison of Wcet produced by all arbiters is presented.



(a) Standard cache-line mapping



(b) Bank interleaved cache-line mapping

Figure 4.1: SDRAM cache-line mapping modes

4.1 SDRAM operation

An SDRAM stores data in a 2D array of rows and columns, called banks, as depicted in Fig. 4.1. Cache-lines can be mapped to the banks in the following ways.

4.1.1 Standard cache-line mapping/Open page policy

In standard cache-line mapping, each cache-line is entirely mapped to a single row of single bank (Fig. 4.1(a)). Data from a bank can only be read through a row buffer, hence, before reading from or writing to any row, the targeted row is copied to the row buffer of the bank. This operation is called *Row Activation* (`ACT`) [36]. Data can be accessed rapidly from the row buffer. The row buffer typically increases the throughput of the memory since data is accessed with reasonable spatial locality. Hence, most of the time access to the row buffer produces a *row buffer hit*. This is the same principle on which caches are designed. However, if data from another row of the same bank is requested, the current content of the buffer must be copied back to the respected row. This operation is called *Precharge* (`PCH`). Subsequently, the new row is activated.

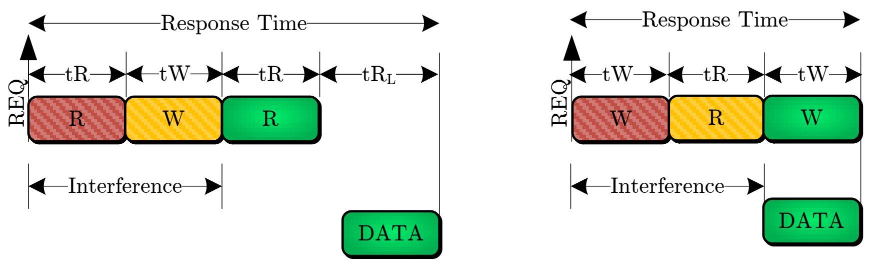


Figure 4.2: Response times: (a) read request (b) write request.

In mutli-core systems with shared SDRAM, one core may precharge the row buffer which is activated by the test-application (bank interference). Considering the possibility of the asynchronous eviction of contents of row buffer, for the worst case latency analysis, row buffer miss is assumed for each cache miss. This significantly increases the WCET of the test application.

Additionally, an SDRAM stores data in the form of capacitive charge which must be refreshed periodically (@ `tREFI`) to compensate for the charge leakage. The refresh operation is guarded by hardware independent of application execution and accessing an SDRAM during the refresh operation is not allowed. Again, for the worst case latency analysis, *refresh interference* must be assumed for every cache miss which further increases WCET.

4.1.2 Bank interleaved cache-line mapping

Bank interleaved mapping is used to avoid bank interference. Here, instead of mapping a cache-line on a single row of a bank, it is split and mapped to all banks, as depicted in Fig. 4.1(b). Cache-line can be accessed in a pipeline fashion, while one row of a bank is activated, data from another bank can be provided in parallel. Since the bank interleaved mapping (close page policy) is proven to reduces the impact of co-existing applications on the worst case latency [92], in this chapter, we use bank interleaved mapping to access a shared SDRAM. However, in Sec. 6.2.2, we raise questions on this popular assumption and leave further analysis for future work.

An SDRAM uses bidirectional pins to receive/transmit data. If consecutive accesses are of different types, read/write, the direction of the pin drivers must be switched which comes at cost of switching penalty (delay). The switching penalty is irrespective of open-page or closed page policy. Here again, for the worst case analysis, we assume that the access from the test application is interfered by a sequence of alternating accesses, as depicted in in Fig. 4.2.

Note the subtle difference between a read access and a write access. A write access, once scheduled, finishes in maximum tW clock cycles. A read access, once scheduled, needs maximum

tR clock cycles to issue a read command to the memory. The memory responds with the requested data after tR_L clock cycles, at most. Hence, the worst case read latency is $tR + tR_L$. The parameters tR_L, tR, and tW depend on the type of SDRAM, its configuration and its operating frequency.

4.2 Priority based budget scheduler – PBS

In circular wheel based scheduling schemes (TDMA, PD and RR), guaranteed low latency can only be achieved by allocating higher bandwidth. For example, by adding more slots for a particular master in the wheel. Unlike the circular wheel based arbitration, the PBS decouples the worst case latency from the allocated bandwidth. The PBS is graphically explained in Fig. 4.3, Fig. 4.4 and detailed in the following sections.

4.2.1 Basic operation

The PBS arbiter, instead of defining timing windows (slots) to access the shared memory, defines fixed budget of number of accesses for each master in a unit time (Replenishment period). The budgeting distributes the memory bandwidth to the masters according to their memory requirements and also prevents monopolizing of the memory. Due to the absence of fixed slots, the memory can be utilized more efficiently. After each access, the originating master's budget is reduced by one. After a master consumes all its budget, it becomes ineligible in the current replenishment period and has to wait for a grant until the next replenishment period starts. All masters receive their fixed budget back at the beginning of a replenishment period. A single replenishment period must be able to serve budget of all masters, also in the worst case. Equation (4.1) derives the worst case command time (tC) in clock cycles. The equation considers the alternating accesses. Equation (4.2) derives the length of a replenishment period considering the worst case scenario. Here, N is the total number of masters in the system.

$$tC = \left\lceil \frac{tR + tW}{2} \right\rceil \tag{4.1}$$

$$R_p = tC \times \sum_{i=0}^{(N-1)} Budget[i] \tag{4.2}$$

Conflicts of simultaneously arriving accesses are resolved by masters' priorities. Typically, a critical master is assigned a high priority for achieving low access latency.

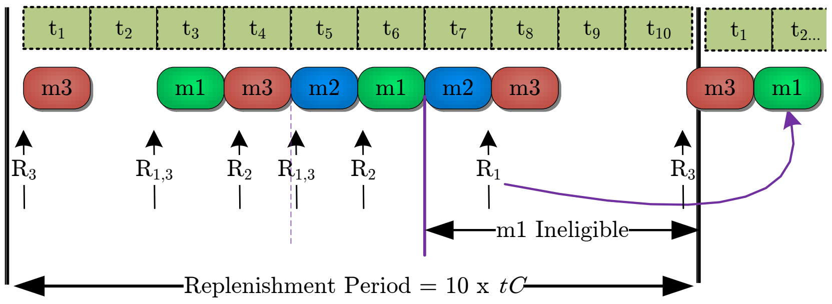
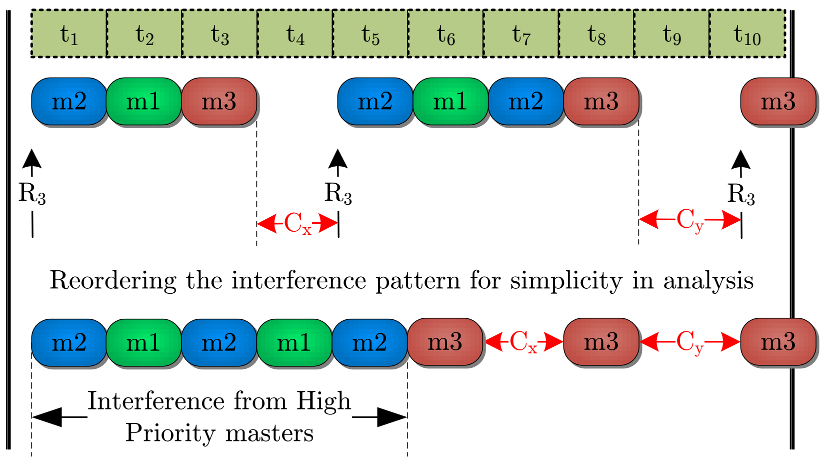


Figure 4.3: PBS operation. Priority [m1, m2, m3] = [p1, p2, p3], Budget[m1, m2, m3] = [2, 3, 5]. Σ Budget = 10.

Fig. 4.3 depicts three masters (`m1`, `m2` and `m3`) with their budgets (2, 3 and 5). The `m1` has the highest priority while `m3` has the lowest. The time instances $t_1..t_{10}$ are for explanation only. Such fixed slots do not exist in PBS. Consider at t_1, a request from `m3` (R_3) arrives. Since other masters are inactive at that time, the `m3` is scheduled immediately. During t_2, the bandwidth is wasted since there is no active master. At the beginning of t_3, both `m1` and `m3` request simultaneously. Since `m1` has higher priority than `m3`, `m1` is scheduled first and `m3` is scheduled during t_4. During t_4, a requests from `m2` (R_2) arrives. At the beginning of t_5, a request from `m2` is already backlogged and `m1` is inactive. Hence, `m2` is scheduled in t_5. Right after `m2` is scheduled, requests from `m1` and `m3` arrive. Although the `m1` has higher priority than the `m2`, `m1` has to wait until `m2` is finished since the PBS does not preempt a scheduled memory transfer. Thus, `m1` is scheduled in t_6.

After t_6, `m1` has consumed its allocated budget in the current replenishment period. Hence, it is considered ineligible during the remaining replenishment period and can only be scheduled in the next replenishment period. The PBS does not employ any frame based scheduling. The replenishment period, as the name suggests, are the fixed points in time when budgets are resettled. As long as a master is eligible, it is scheduled according to static priorities. This can be seen at the end of the replenishment period shown in Fig. 4.3. Here, just at the end of the current replenishment period, `m3` requests an access. Since `m3` is still eligible and `m2` is inactive, `m3` is scheduled immediately. However, it completes in the next replenishment period. This interferes with the already backlogged request from `m1` at the beginning of the next replenishment period.



t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 t_9 t_{10}
m2 m1 m3 m2 m1 m2 m3 m3
R_3
R_3
R_3
C_x
C_y
Reordering the interference pattern for simplicity in analysis
m2 m1 m2 m1 m2 m3 C_x m3 C_y m3
Interference from High Priority masters

Figure 4.4: Worst case latency analysis

4.2.2 Worst case latency analysis

The worst case latency analysis under the PBS arbiter is based on the following intuitions.

1. Interfering accesses and the access from the test-master form an alternating read/write access pattern which maximizes SDRAM latency (Sec. 4.1.2).

2. Each master in the system, except the test-master, preserves its budget and uses it only to interfere with the accesses from the test-master.

3. One lower priority master is scheduled just one clock cycle before an access request from the test-master arrives. Thus, this low priority master interferes with every access from the test-master.

4. One refresh latency is considered at every `tREFI` clock cycles during analysis.

Fig. 4.4 depicts an assumption that we made for simplifying the analysis. Here, consider that `m3`, the lowest priority master, is under investigation. As shown in the upper part of the figure, according to the intuition 2, high priority masters use their entire budget only to interfere with the test-master (`m3`). This interference may occur to a single access from `m3` or may be divided among multiple accesses. The upper part of the figure shows that the interference from high priority masters occurs to the first two accesses of `m3`. After the second access of `m3` is scheduled, all high priority masters are ineligible in the current replenishment period. Hence, subsequent accesses of `m3` in that replenishment period is scheduled without any interference.

During analysis, it is unknown which access from the test-master will suffer how much interference. Hence, we assume that the first access of test-master (here, `m3`) in a replenishment period suffers interference from all high priority masters exploiting their entire budget. The subsequent accesses from the test-master are scheduled immediately as long as the master has the budget. This assumption is explained graphically in the lower part of the figure. Here, c_x and c_y are the computation time (Sec. 3.1.4) between two accesses. The above mentioned assumption leads to the following simplification in counting number of interfering accesses.

The first access of master `m` in any replenishment period may be interfered by X_m^1 number of accesses in the worst case. The subsequent accesses in the *same* replenishment period is interfered by X_m number of accesses.

$$X_m^1 = \begin{cases} 1, & m = 1 \\ 1 + \sum_{i=1}^{m-1} Budget[i], & m \in (1, N) \\ \sum_{i=1}^{N-1} Budget[i], & m = N \end{cases} \tag{4.3}$$

Note that in equation (4.3), we assume that all the high priority masters consume their entire budget to interfere with the test-master (`m`). This is only possible if for the remaining time these masters are idle, according to the 2^{nd} intuition.

$$X_m = \begin{cases} 1, & m \in [1, N) \\ 0, & m = N \end{cases} \tag{4.4}$$

In equation (4.3), if the test-master is the highest priority master ($m = 1$), its first access in any replenishment period can be interfered by only one lower priority access. This is in accordance with the 3^{rd} intuition. If the master is neither the highest priority nor the lowest priority ($m \in (1, N)$), its first access can be interfered by all high priority masters exploiting their entire budgets and one lower priory master. If the master is the lowest priority master ($m = N$) then its first access can be interfered by all high priority masters exploiting their entire budgets.

Similarly, in (4.4), if the test-master is the lowest priority master, its subsequent accesses in a replenishment period are scheduled immediately since all the other masters are ineligible in that particular replenishment period. However, if the test-master is not the lowest priority master ($m \in [1, N)$), its subsequent accesses may be interfered by one on-going low priority access.

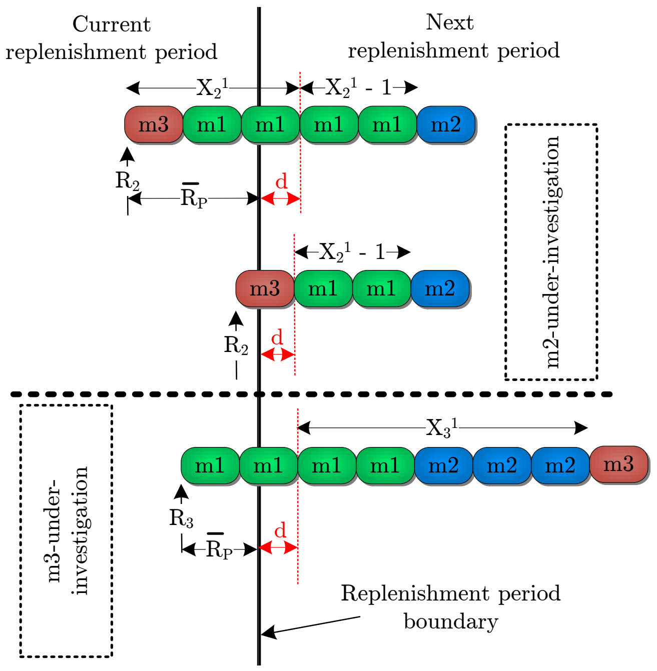


Figure 4.5: Worst case latency analysis while crossing the replenishment period boundary. $\overline{R_p}$ is the remaining time in the replenishment period when an access from the test-master arrives. d is the time occupied by an interfering access in the next replenishment period.

In Fig. 4.5, interference scenarios are presented where an access from the test-master can only be scheduled in the next replenishment period although the master still has budget left in the current replenishment period. In the upper part of the figure, `m2` (medium priority) is under investigation while in the lower part, `m3` (lowest priority) is under investigation. Here, an interfering access is scheduled in the current replenishment period, but, cannot be completed in the current replenishment period. While the interfering access finishes, in the background, a new replenishment period starts and all the masters get back their initial budget. The interfering access needs d number of clock cycles in the next replenishment period. Factor d can be calculated using equation (4.5).

$$d = tC - (\overline{R_p} \bmod tC) \tag{4.5}$$

Consider `m2`-under-investigation. The figure shows that just before the request R_2 arrives, the arbiter has scheduled a lower priority master `m3`. Since `m2` is eligible, no further lower priority accesses can be scheduled. However, the higher priority master (`m1`) regains its budget and may interfere once again exploiting its entire budget. Thus, in the next replenishment

Algorithm 1 Frontend method for WCET calculation

```
1: $FrontEnd()$
2: $sim = 0$
3: while $i \leq TotalAccesses$ do
4:     $sim \mathrel{+}= PbsLatency(sim, e_i)$
5:     $sim \mathrel{+}= c_i$
6: end while
7: $temp = (sim/tREFI) + 1$
8: $N_{ref} = (temp > TotalAccesses)?(TotalAccesses) : (temp)$
9: $sim \mathrel{+}= N_{ref} \times tRFC$
10: $wcet = sim$
```

period, interference from only `m1` (higher priority master) is considered. This is denoted by considering $X_2^1 - 1$ (remember that $X_m^1, m \neq N$ includes interference from one lower priority master – equation (4.3)). For the lowest priority master `m3`, the factor d must be added in the X_m^1 to consider the first access interference in the next replenishment period.

Considering the above discussion, we define a factor X_m^d which is given by the equation (4.6).

$$X_m^d = \begin{cases} d + X_m^1 - 1, & m \in [1, N) \\ d + X_m^1, & m = N \end{cases} \tag{4.6}$$

Assuming the computation trace is available as, $T = \{(t_0, e_0), (t_1, e_1), ..\}$ where $e_0, e_1...$ are read or write events (accesses) and $t_1, t_2, ...$ are occurrence times of these events, equations (4.1) to equations (4.6) and algorithms 1 and 2 gives the WCET of that particular execution path under the PBS arbiter. Here, c_i is the computation time.

In algorithm 1[1], the *FrontEnd* method is responsible for reading the computation trace. The *FrontEnd* method remains the same while the method *PbsLatency* is replaced by a method appropriate for the arbiter. The *FrontEnd* method adds the worst case latency of each access (event) e_i, calculated by *PbsLatency* method, into the computation time c_i. Similar to Sec. 3.1.4, this implies appending each event in a computation trace with its worst case latency. The penalty of refresh is considered at the end by considering the maximum number of possible refreshes at every `tREFI` clock cycles. Being conservative, it is assumed that one refresh may interfere with the first access of the trace, hence, $temp = (wcet/tREFI) + 1$. Also note that the number of interfering refreshes (N_{ref}) cannot be more than the number of total accesses in a computation trace. This complies with the 4th intuition of Sec. 4.2.2.

[1]The code statements of the algorithms presented in this thesis are in C-language.

Algorithm 2 Algorithm to Calculate the worst case latency for a single access under the Pbs

```
1: $PbsLatency(sim, e_i)$
2: $lat = New = 0$
3: while $sim > NextRp$ do
4:     $NextR_p \mathrel{+}= R_p$
5:     $New = 1$
6: end while
7: $\overline{R_p} = NextR_p - sim$
8: if $m = ineligible$ then
9:     $lat \mathrel{+}= \overline{R_p}$
10:    $NextR_p \mathrel{+}= R_p$
11:    $New = 1$
12:    $\overline{R_p} = R_p$
13: end if
14: if $New = 1$ then
15:    $Interference = X_m^1 \times tC$
16: else
17:    $Interference = X_m \times tC$
18: end if
19: if $Interference > \overline{R_p}$ then
20:    $lat \mathrel{+}= \overline{R_p} + X_m^d \times tC$
21:    $NextR_p \mathrel{+}= R_p$
22: else
23:    $lat \mathrel{+}= Interference$
24: end if
25: $lat \mathrel{+}= (e_i = R)?(tR + tRL) : (tW)$
26: $Budget_m --$
27: $lat \hookleftarrow$
```

The algorithm 2 analyzes the worst case latency of each access considering its available budget, time of occurrence and the intuitions presented in Sec. 4.2.2. In *PbsLatency* method, between lines 3 and 6, the arrival time of the incoming access is categorized in the current replenishment period or in one of next replenishment periods. If the arrival time is in one of next replenishment periods then the access is considered as a first access in that replenishment period and the time in analysis is incremented ($NextRp \mathrel{+}= R_p$). The beginning of a new replenishment period is flagged by $New = 1$ in line 5. The remaining time in the current replenishment period (either new or current), $\overline{R_p}$, is calculated in line 7.

Between lines 8 and 13, the eligibility (budget left or not) of the test-master in the replenishment period is investigated[1]. If the master is ineligible, it cannot be scheduled in the remaining replenishment period ($\overline{R_p}$). Hence, the $\overline{R_p}$ must be added to the latency. Moreover, the access is considered as a first access in the next replenishment period. Now, the entire new replenishment period is available for scheduling the access ($\overline{R_p} = R_p$).

Between lines 14 and 18, $interference$ is calculated based on whether it is the first access in the replenishment period or the subsequent access. Between lines 19 and 24, if the $interference$ crosses the boundary into the next replenishment, the remaining replenishment period ($\overline{R_p}$) and X_m^d are added to the latency. If the access can be scheduled in the current replenishment period then only $interference$ is added to the latency.

Finally, in line 25, the access from the test-master (`m`) is scheduled. Depending on its type, read or write, $(tR + tRL)$ or tW is added to the latency and its budget is reduced by one. The final latency value is returned to the $FrontEnd$ method.

Note that tC is the number of clock cycles required to execute a read access or a write access on the shared SDRAM considering alternating read/write accesses, on average (equation (4.1)). The parameters X_m, X_m^1 and X_m^d define the number of interfering accesses in the worst case. These parameters are multiplied by tC to stay in accordance with the 1^{st} intuition that the interfering access and the access-under-investigation form a sequence of alternating read/write accesses.

4.3 Credit controlled static priority arbiter – CCSP

The CCSP arbiter, like the PBS arbiter, employs static priorities to decouple latency and allocated rate. However, unlike the PBS arbiter, the replenishment period is not defined in terms of fixed periodic frames. Instead, it is defined in a more fine-grain manner. Each master in the system is replenished independently at different time according to its allocated rate. Before we explain the basic operation of the CCSP, let us describe the concept of the latency rate servers.

4.3.1 Latency rate servers

The latency-rate ($\mathcal{LR}$) [38] servers can be applied as an abstract analysis method for shared SDRAM. The $\mathcal{LR}$ server simplifies modeling of shared resource by representing it using only

[1]Certainly, if a new replenishment period is started, the master is eligible since it received its initial budget at the start of the replenishment period.

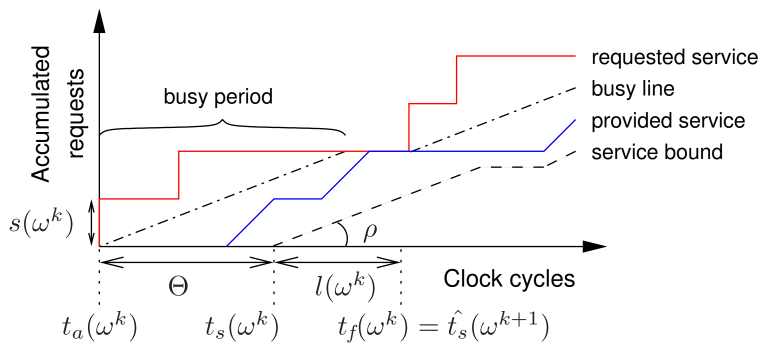


Figure 4.6: A $\mathcal{LR}$ server and its associated concepts.

two parameters, minimum allocated rate (bandwidth) – ρ and maximum service latency (interference) – Θ. These parameters hold their value irrespective of activity of co-existing masters. Fig. 4.6 explains the parameters graphically.

According to the $\mathcal{LR}$ model, the test-master must wait for, at most, Θ clock cycles to receive continuous service ρ. However, if the master does not send requests at rate ρ, then the worst case latency of Θ must be considered again. This fact is captured by *busy periods* (Fig. 4.6 [41]). In the figure the request arrival rate is more than ρ (above the busy line in the figure), then the provided service rate is guaranteed to be maintained at ρ. However, a break in the request rate results in reconsideration of Θ and a break in service curve.

Wiggers *et al* [93] have derived bounds on scheduling times and finishing times using $\mathcal{LR}$ model. They derived that the worst-case *scheduling time*, t_s, of the k^{th} request from a master, m, is the maximum of arrival time of the request plus Θ and finishing time of the $(k-1)^{th}$ request ($t_f(\omega^{k-1})$). This is presented by the first term in Equation (4.7). The size of the k^{th} request is denoted by $s(\omega^k)$. After the request is scheduled, it is served at allocated rate ρ and finishes after *completion latency*, $l(\omega^k) = s(\omega^k)/\rho$. The sum of *scheduling time* and *completion latency* gives the upper bound on finishing time of the k^{th} request, $t_f(\omega^k)$. The upper bound is derived in Equation (4.7) and graphically depicted in Fig. 4.6.

$$t_f(\omega^k) = \max(t_a(\omega^k) + \Theta, t_f(\omega^{k-1})) + s(\omega^k)/\rho \tag{4.7}$$

The equation (4.7) and Fig. 4.6 are for explaining the concept where the arrival of accesses are assumed independent from each other. However, in this thesis, we are analyzing cache misses[1]. Hence, $t_a(\omega^k) + \Theta > t_f(\omega^{k-1})$. Thus, the equation (4.7) can be re-written as,

[1]Latency to a cache miss delays occurrence of the subsequent cache misses Sec. 3.1.4.

$$t_f(\omega^k) = t_a(\omega^k) + \Theta + s(\omega^k)/\rho \tag{4.8}$$

Both Θ and $l(\omega^k)$ are represented in terms of service cycle. Here, a service cycle is equal to the time it takes to schedule one access request. Hence, considering the worst case alternating traffic, the service cycle is tC (equation (4.1)).

4.3.2 CCSP – Basic operation

The CCSP arbiter has two components, a rate regulator and a static priority arbiter [32]. The regulator controls the flow of incoming access requests and enforces the allocated rate, ρ, on the flow. The accesses which pass through the regulator are arbitrated according to their priorities. The masters are allocated burstiness (σ_m – initial credits) and allocated rate (ρ_m). Clearly, $\sum_{\forall m} \rho_m \leq 1$ must hold to make sure that the shared memory has sufficient bandwidth to serve total allocated rate of the masters in the system.

The allocated burstiness, σ_m and rate ρ_m are used by the regulator to compute the number of credits, $\lambda_m(t)$, at a given time – t, Equation (4.9) ([32]).

The operation can be explained as follows: A master is assigned initial credits of σ_m. The credit is incremented by ρ_m after every tC clock cycles (one service cycle), if the master is not scheduled. If the master is scheduled, $\gamma(t) = m$, its credit is decremented by one. If a master is not scheduled and does not have any backlogged request ($\beta_m = 0$), its credits keep on increasing by ρ_m. This may lead to accumulation of too many credits and may result in starvation of the lower priority masters. Hence, the credits are saturated at its initial value, σ_m. Only, if the master has a backlogged request then only the credits are allowed to increase beyond σ_m. The credit accounting is presented in Equation (4.9).

$$\begin{aligned} \lambda_m(0) &= \sigma_m \\ \lambda_m(t+1) &= \begin{cases} \lambda_m(t) + \rho_m - 1 & (\gamma(t) = m) \\ \lambda_m(t) + \rho_m & (\gamma(t) \neq m) \wedge (\beta_m > 0) \\ \min(\lambda_m(t) + \rho_m, \sigma_m) & (\gamma(t) \neq m) \wedge (\beta_m = 0) \end{cases} \end{aligned} \tag{4.9}$$

Equation (4.10) [32] gives service latency in terms of service cycles for master `m`. Here, set of masters with higher priority than `m` is denoted by M_m^+.

$$\Theta_m = \frac{\sum_{\forall m_j \in M_m^+} \sigma_{m_j}}{1 - \sum_{\forall m_j \in M_m^+} \rho_{m_j}} \tag{4.10}$$

The numerator of the equation (4.10) assumes that the all the higher priority masters consume all their credits to interfere only (similar to our intuition in Sec. 4.2.2 for the PBS arbiter). The denominator calculates the number of service cycles required to consume these credits. Notice that during consumption of these credits, according to equation (4.10), interfering masters are continuously replenished. Hence, it may be possible that some of the high priority masters regain credits and interfere again. It is worth mentioning the similarities between the PBS and the CCSP arbiters. In Fig. 4.5, notice that the `m1` interferes with the lower priority masters (`m2` and `m3`) twice since it regains its initial budget at the beginning of a replenishment period. This scenario is captured by the denominator of equation (4.10). Note that as $\sum_{\forall m_j \in M_m^+} \rho_{m_j} \to 1$, $\Theta_m \to \infty$. This means that higher priority masters are replenished as soon as they consume their credits and potentially interfere again with their newly acquired credits. Thus, the lower priority master will never be scheduled.

4.3.3 Detailed worst case latency analysis

In this subsection, instead of using the abstract $\mathcal{LR}$ models, we analyze the worst case latency under the CCSP arbitration in detail. The abstract $\mathcal{LR}$ analysis considers request and service as infinitesimally divisible. However, when a memory is a shared resource, neither the request nor the service is infinitesimally divisible. This leads to over estimation of the worst case latency in the following ways.

- In the equation (4.8), completion time is modeled as $s(\omega^k)/\rho$. This means the request is served at the rate ρ and it completes after $tC \times s(\omega^k)/\rho$ clock cycles. However, on the shared memory, once a burst access is scheduled (at $t_a(\omega^k) + \Theta$ time), it is served with the full bandwidth of the memory. Hence, in reality, it completes after $tC \times s(\omega^k)$ clock cycles. This has a large impact on the accuracy of the completion time. Consider the following values for the parameters. These values are also used in Sec. 4.5 while comparing all the arbiters.

  $s(\omega^k) = 1$ – one burst access to fill the cache line, $tC = 13, \rho = 0.25$. The $\mathcal{LR}$ approach yields 52 clock cycles as *completion latency*, $l(\omega^k)$, while in reality, the completion latency is only 13 clock cycles.

- The scheduling latency Θ_m is computed according to equation (4.10) in the $\mathcal{LR}$ approach. The numerator of the equation considers partial credits of masters and sums them up. Additionally, the denominator considers that these credits are consumed at the total allocated rate of higher priority masters. However, in reality, a higher priority master must have at least one full credit to be able to interfere. Moreover, as a higher priority master is scheduled, it is served with the full bandwidth of the shared memory. To consider the impact, consider the following parameters.

$\sigma_0 = 0.5$, $\rho_0 = 0.25$, $\sigma_1 = 0.5$, $\rho_1 = 0.25$, $\sigma_2 = 0.5$, $\rho_2 = 0.25$ and $\sigma_3 = 1$, $\rho_3 = 0.25$. $tC = 13$. For test-master, `m` = 3 – the lowest priority master, according to equation (4.10), $\Theta_3 = 13 \times 1.5/(1-0.75) = 78$. However, in reality, $\Theta_3 = 0$ since none of the higher priority masters have one full credit to pay for an access.

Based on the above observations and the following intuitions [41], we build our detailed worst case latency analysis. Although these intuitions are very similar to that of PBS, the analysis is reasonably complex due to the fine-grain replenishment of the CCSP arbiter.

1. Interfering accesses and the access from the test-master (`m`) form an alternating read/write access pattern which results in the worst case SDRAM latency (Sec. 4.1.2).

2. All masters use their credits *only* to interfere with `m`. When `m` is idle ($\beta_m = 0$) or ineligible ($\lambda_m < 1$), co-existing masters also stay idle and accumulate maximum possible credits. As soon as `m` becomes eligible and sends an access request, all higher priority masters send requests simultaneously to maximally interfere with the request from `m`.

3. Access from a lower priority master is scheduled one clock cycle before `m` becomes eligible and backlogged. Hence, this lower priority master interferes with all higher priority masters and `m`.

4. Penalty of a refresh is considered at every `tREFI` clock cycles. Clearly, the total number of refresh peanlty is less than the total number of accesses in the computation trace.

$$P_m = tC/\rho_m \qquad \forall m \in [1, N] \tag{4.11}$$

Equation (4.11) derives the replenishment period in clock cycles for each master in the system. After P_m clock cycles, master `m` receives one full credit. In the equation, to consider the worst case read/write pattern, according to the 1^{st} intuition, tC is considered as a time required to do one access to the shared memory. Remember that from equation (4.1), $tC = (tR+tW)/2$.

The variable η_m indicates the value of clock cycle when `m` will receive (in future) a next full credit. Thus, at time $sim = 0$, $\lambda_m = \sigma_m$ and $\eta_m = P_m$. However, as described in the equation

Algorithm 3 Algorithm for credit accounting under the CCSP

```
1: $UpdateCredit(sim, s, low, high)$
2: for $x \in [low, high]$ do
3:   if $\lambda_x \geq \sigma_x \land s$ then
4:     $\eta_x = sim + P_x$
5:   else
6:     if $sim \geq \eta_x$ then
7:       $\lambda_x$ += $1 + (sim - \eta_x)/P_x$                # $\lambda_x \in \mathbb{N}^0$
8:       $rem = (sim - \eta_x)\%P_x$
9:       $\eta_x = sim + P_x - rem$
10:    end if
11:    if $\lambda_x \geq \sigma_x \land s$ then
12:      $\lambda_x = \sigma_x$
13:    end if
14:  end if
15: end for
```

(4.9), if m does not consume its credits, the credits are saturated at σ_m and η_m is only updated if $(\lambda_m < \sigma_m) \lor (\beta_m > 0)$.

Algorithms 3 and 4 compute the WCET of the given computation trace. The *FrontEnd* method remains the same as in algorithm 1. However, instead of *PbsLatency*, here it calls *CcspLatency* method for the worst case latency computation of a single access. The refresh interference (4^{th} intuition) is considered in the *FrontEnd* method.

The algorithm 3 manages replenishment of credits. The *UpdateCredit* method is called if credits of a range of masters from $[low, high]$ must be calculated at time *sim*. The saturate flag, *s*, informs the method if the credits must be saturated at σ values. The saturation is applied when, as presented in equation (4.9), a master is not scheduled and does not have any backlogged request($(\gamma(t) \neq x) \land (\beta_x = 0)$). If the current time, *sim*, exceeds the time when a master is supposed to receive a full credit (η_x) then its credits (λ_x) are replenished accordingly and the new η_x values are computed. Note that $\lambda_x \in \mathbb{N}^0$, hence, unlike $\mathcal{LR}$ analysis, accumulation of partial credits are avoided.

The algorithm 4 computes the worst case latency any access to the shared memory may suffer. At first, in line 3, credits of all masters ([1,N]) are updated according to the current time ($sim' = sim$). Here, the *UpdateCredit* method is called with saturate flag *s*. According to our 2^{nd} intuition, other masters in the system remain idle when m is idle and accumulate as many credits as possible.

Algorithm 4 Algorithm to Calculate the worst case latency for a single access under the Ccsp

```
1: $CcspLatency(sim, e_i)$
2: $sim' = sim$
3: $UpdateCredit(sim', s, 1, N)$
4: while $\lambda_m = 0$ do
5:   $sim' = \eta_m$
6:   $UpdateCredit(sim', s, 1, N)$
7: end while
8: if $m \in [1, (N-1)] \wedge \sum \lambda_{m_i} > 0$, $(\forall m_i \in M_m^-)$ then
9:   $sim' \mathrel{+}= tC$
10:  $ConsumeLowPriorityCredit()$
11:  $UpdateCredit(sim', \overline{s}, 1, m)$
12: end if
13: while $\sum \lambda_{m_j} > 0$, $(\forall m_j \in M_m^+)$ do
14:   for $x \in m_j$ do
15:     $sim' \mathrel{+}= \lambda_x \times tC$
16:     $\lambda_x = 0$
17:     $UpdateCredit(sim', \overline{s}, x+1, m)$
18:   end for
19: end while
20: $sim' \mathrel{+}= (e_i = R)?(tR + tRL) : (tW)$
21: $\lambda_m --$
22: $lat = sim' - sim$
23: $lat \hookleftarrow$
```

If the test-master, m, does not have any credit ($\lambda_m = 0$) at the current simulation time, the simulation time is forwarded ($sim' = \eta_m$) until m receives a credit. Again, during this forwarded time, other masters in the system remain idle according to the 2nd intuition. Hence, in line 6, the *UpdateCredit* method is called with the saturate flag s. The *ConsumeLowPriorityCredit*() method, as its name suggests, consumes a credit from one of the lower priority masters and sets their η values accordingly.

Now, at line 8, m is ready to do an access. At this time, according to the 3rd intuition, interference from one lower priority master is considered. If the m is not the lowest priority master and a least one lowest priority master has a credit then the lower priority master is scheduled ahead of m in the worst case. This low priority transfer blocks other high priority masters including m ([1,m]). Hence, according to equation (4.9), they are allowed to accumulate more that σ credits. This is informed to the *UpdateCredit* method by $\overline{s}$ flag.

Algorithm 5 Algorithm to Calculate the worst case latency for a single access under the Rr and the Sp

```
1:  $RrLatency(sim, e_i)$
2:  # ifdef RR
3:          $lat = (N - 1) \times tC$
4:  # ENDIF

5:  # ifdef SP
6:          $lat = 1 \times tC$
7:  # ENDIF

8:  # ifdef $PD^{h1}$
9:          $lat = c_i \% tC$
10: # ENDIF

11: $lat \mathrel{+}= (e_i = R)?(tR + tRL) : (tW)$
12: $lat \hookleftarrow$
```

According to the 2nd intuition, interference from all higher priority masters is considered between lines 13 and 19. Here, each high priority access is blocking the lower priority masters. Hence, after each high priority access, the credits of lower priority masters are updated with $\overline{s}$ flag.

After consuming all high priority credits, in line 20, `m` is scheduled and its credits are reduced by one. The worst case latency for this access, $lat = sim' - sim$, is returned to the $FrontEnd$ method.

4.4 Worst case latency analysis of the Round robin, the TDMA, the PD and the Static Priority (SP) arbiters

The basic operation and the worst case latency analysis under Rr, Tdma and Pd arbiters are already presented in the chapter 3. In this section, we present how their latency analysis can be interfaced with the $FrontEnd()$ method presented in the Algorithm 1.

The worst case latency calculation for the Rr arbiter is presented in Algorithm 5. The $RrLatency$ method simply assumes that all other cores in the system requests an access at the same time and the test-master, `m`, has the last slot in the circular scheduling wheel.

Under the Sp scheduling, all masters are assigned static priorities and conflicts are resolved

Algorithm 6 Algorithm to Calculate the worst case latency for a single access under the TDMA and PD

```
1: $TdmaLatency(sim, e_i)$
2: $WheelLength = N \times tC$
3: $\bar{t} = WheelLength - (c_i \% WheelLength)$
4: if $\bar{t} \geq tC$ then
5:     $lat = \bar{t} - tC$
6: else
7:     $lat = \bar{t} + (N - 1) \times tC$
8: end if
9: $lat \mathrel{+}= (e_i = R)?(tR + tRL) : (tW)$
10: $lat \hookleftarrow$
```

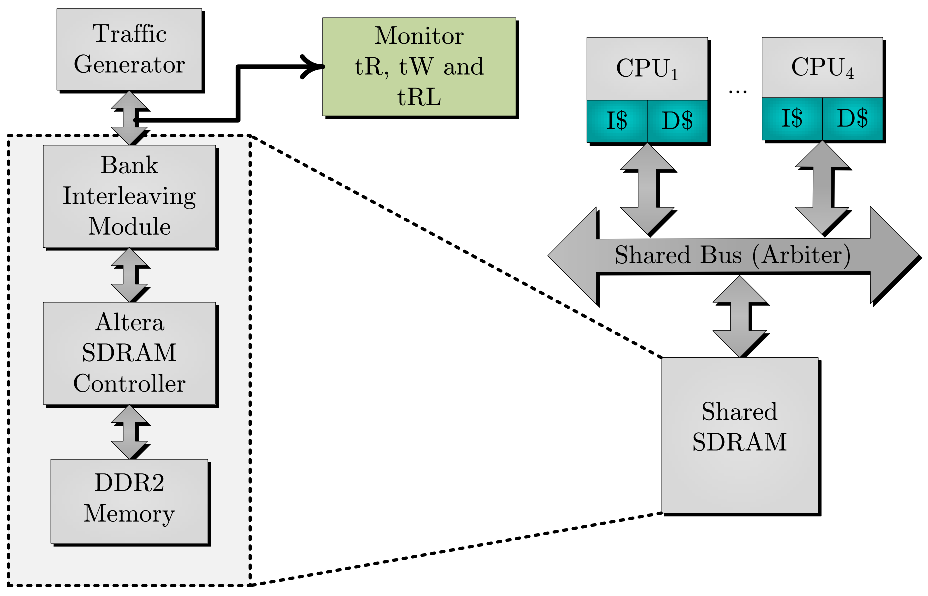
only considering the static priority. Clearly, under the SP, only the highest priority master ($m1$) is WCET analyzable. In the worst case, the highest priority master is interfered by only one ongoing lower priority access. Hence, as depicted in the Algorithm 5, for static priority, interference from only one lower priority master is considered in the worst case latency.

The Algorithm 6 computes the worst case latency of a single access under TDMA and PD arbiters. Note that, unlike the chapter 3, this chapter uses an SDRAM as a shared memory. Hence, simple measurement does not guarantee the worst case read latency, tRL. This worst case latency occurs only if the traffic towards the shared SDRAM is of read/write switching type. The algorithm assumes that each master has exactly one static slot in the TDMA schedule, hence, the $WheelLength = N \times tC$. The remaining time in the wheel is calculated based on the access gape (computation time $- c_i$) between the previous access and the current access.

If the time left in the wheel, $\bar{t}$, is greater or equal to tC then the access is scheduled in the current wheel cycle. Otherwise, the access is scheduled in the next wheel cycle. Note that if an application has c_i values such that when it accesses the shared resource, the remaining time in the wheel is approximately equal to tC ($\bar{t} \approx tC$), then according to line 5 of the algorithm, $lat = 0$. Thus, the access is scheduled immediately. Such applications are favored by the static rotating scheduler. More information about it can be found in Chapter 6.

4.5 Tests

The Fig. 4.7(a) depicts our method of extracting the worst case parameters and the Fig. 4.7(b) depicts our test architecture with four Altera NIOS II Fast cores. Each core has 512 Bytes of instruction and data caches. The cache-line size is 32 Bytes. A DDR2 memory is used as



(a) Extraction of parameters (b) Test architecture

Figure 4.7: Extraction of the worst case parameters from our test architecture

a shared main memory. All the masters have one allocated slot under TDMA, RR and PD in a wheel cycle. Similarly, they have budget of one transfer under the PBS in a replenishment period and under the CCSP all masters have, equal, allocated rate of 0.25.

As depicted in Fig. 4.7(a), at first a traffic generator is connected to the shared SDRAM (we consider the combination of the bank interleaving module, the Altera SDRAM controller and the SDRAM itself as a shared SDRAM). The traffic generator produces back-to-back alternating read/write accesses. The entire set-up is executed on a Cyclone III FPGA development board and the worst case parameters, the worst case write time tW, the worst case read time tR and the worst case read latency tRL are extracted by a monitoring device (e.g. Signal tap II logic analyzer). These parameters are used in the algorithms presented in this chapter. Under the above mentioned configuration, $tR = 12$, $tRL = 33$ and $tW = 14$ clock cycles.

The table 4.1 depicts the area overhead of the arbiters discussed in this thesis. Note that all conventional arbiters (SP, TDMA and RR) and the PD arbiter are almost equal in size. The RR arbiter is negligibly bigger than the conventional arbiters and the PD arbiter since, as explained in Sec. 3.3.4, it has to manage circular schedule dynamically. Compared to these arbiters, the budget based arbiters (PBS and CCSP) are significantly large due to their replenishment period management. The CCSP arbiter is further penalized due to the individual accounting and replenishment mechanism for each master.

Arbiter	Sp	Tdma	Rr	Pd	Pbs	Ccsp
Number of LEs	334	332	350	336	545	716

Table 4.1: Area overheads of arbiters

4.5.1 Comparison of the WCET produced under different arbiters

To compare the Wcet produced under the arbiters presented in this chapter, computation traces (Sec. 3.1.4) of each *single path*[1] applications from the Mälardalen Wcet benchmark suit is collected. As explained in Chapter 6, the wrap around burst access to the shared Sdram must be prevented for integrity of the worst case parameters. In Altera NIOS II processor, data-cache master can be instructed to issue a burst only on the *BurstBoundary*. However, instruction cache always issues a wrap around burst. Hence, the following user flags were used to build the program in order to align all branch targets to the 32 Byte cache-line boundary.
$-falign - functions = 32$
$- falign - jumps = 32$
$- falign - loops = 32$
$- falign - labels = 32$

After insertion of the above mentioned flags, the cache miss behavior does not remain consistent with the one shown in Table 3.1.

The Table 4.2 presents comparison of the Wcet produced under different arbitration to the Wcet produced under static priorities (Sp). Clearly, under the static priority, only the highest priority master, $m1$, is analyzable. All other masters have *infinite* Wcet. The highest priority masters under the Pbs and Ccsp behave similar to Sp since the allocated budget/rate are according to the memory requirements of these applications. Unlike Sp, Pbs and Ccsp are starvation free arbiters. Hence, the Wcet of lower priority masters can also be estimated as presented in this chapter. The Wcet significantly increases as the priority is lowered.

The conventional arbiters (Tdma and Rr) and the Pd are starvation free and do not employ priorities. Hence, the Wcet estimated for all masters is the same. The Tdma and Pd produce significantly tighter bounds[2] compared to the Rr. Moreover, the Wcet produced by Tdma/Pd

[1] Analysis of multi-path applications is presented in Chapter 5
[2] Pd and Tdma produce equal Wcet.

Bench-	SP	PD^{h1}	PD	RR	PBS				CCSP			
mark	$m1$	$m1$	All	All	$m1$	$m2$	$m3$	$m4$	$m1$	$m2$	$m3$	$m4$
compress	1	0.87	1.17	1.44	1.04	1.28	1.63	1.64	1.02	1.26	1.63	1.63
cover	1	0.90	1.13	1.35	1.01	1.22	1.48	1.62	1.02	1.22	1.47	1.47
crc	1	0.95	1.08	1.12	1.03	1.09	1.16	1.18	1.02	1.07	1.16	1.16
duff	1	0.93	1.14	1.26	1.03	1.17	1.35	1.36	1.01	1.16	1.35	1.35
edn	1	0.91	1.13	1.29	1.02	1.20	1.40	1.43	1.02	1.18	1.40	1.40
expint	1	0.93	1.15	1.35	1	1.23	1.46	1.58	1	1.21	1.46	1.46
fac	1	0.97	1.08	1.16	1	1.11	1.21	1.25	1	1.10	1.24	1.24
fdct	1	0.88	1.27	1.41	1.03	1.28	1.57	1.57	1.01	1.26	1.57	1.57
fibcall	1	0.96	1.08	1.15	1.01	1.13	1.22	1.22	1	1.07	1.22	1.22
fir	1	0.94	1.11	1.21	1.02	1.16	1.31	1.33	1.02	1.13	1.28	1.28
jane	1	0.94	1.04	1.20	1.01	1.12	1.27	1.34	1	1.12	1.27	1.27
jfdcint	1	0.92	1.28	1.26	1	1.18	1.33	1.36	1	1.17	1.33	1.33
matmul	1	0.91	1.16	1.29	1.05	1.20	1.39	1.45	1.02	1.17	1.38	1.38
minver	1	0.88	1.21	1.35	1.01	1.22	1.50	1.52	1	1.20	1.50	1.50
ludcmp	1	0.88	1.22	1.37	1.01	1.22	1.53	1.54	1	1.21	1.53	1.53
prime	1	0.97	1.08	1.14	1	1.09	1.19	1.25	1	1.09	1.18	1.18
quart	1	0.88	1.22	1.35	1	1.21	1.50	1.52	1	1.20	1.51	1.51
recursion	1	0.98	1.04	1.06	1.01	1.05	1.09	1.10	1	1.04	1.10	1.10
ud	1	0.88	1.22	1.37	1.02	1.24	1.51	1.54	1.01	1.22	1.52	1.52

Table 4.2: WCET produced under different arbiters compared to the SP

are slightly less than the one produced by the second highest priority masters under CCSP and PBS.

4.5.2 Effects of reduction in allocated bandwidth

In this test, we studied the effect of reduction in allocated bandwidth on the WCET of two application, `jfdcint` and `quart`, under various arbiters. Except for the TDMA/PD and RR, only the highest priority masters are analyzed. The WCET of these application on the highest priority master under the PBS and the CCSP is equal to the WCET produced under SP on the quad-core configuration in the above mentioned test. Moreover, the WCET produced under the TDMA/PD of these application is significantly higher than that produced under SP. This suggests that the allocated slots/budget/rate are sufficient for the memory requirements of these applications under TDMA/PD, PBS and CCSP, respectively. Here, the TDMA/PD produces worse WCET since the shared resource accesses are issued when the next scheduling opportunity

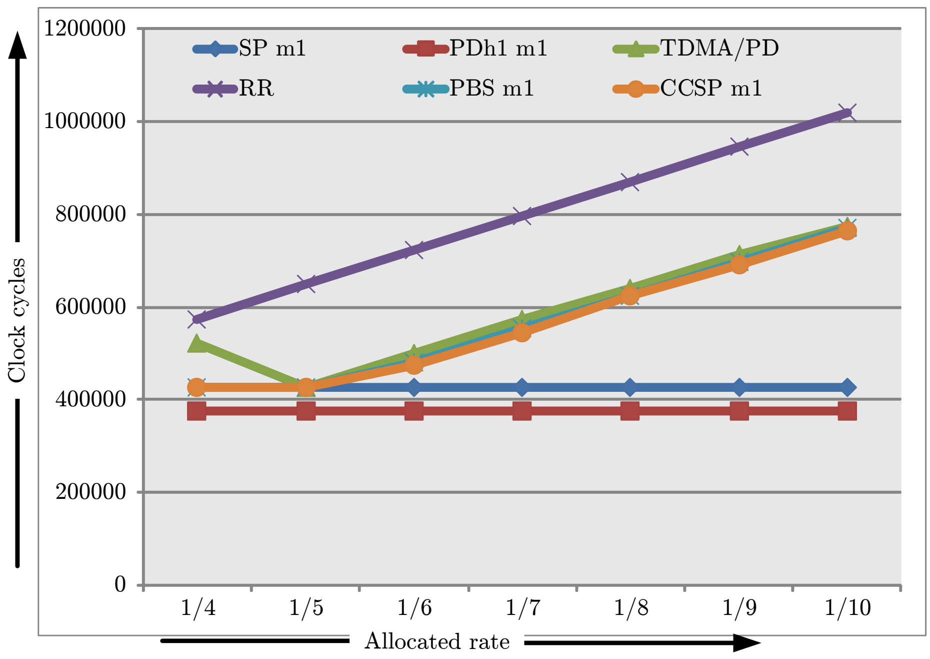


Figure 4.8: Effect of reduction in the allocated bandwidth on the WCET of `jfdcint` application

is far away in the wheel.

Now, let us consider that the allocated bandwidth is less than the required. We emulated this behavior by increasing the number of masters in the system. Since we equally divide the worst case available bandwidth of SDRAM, increasing the number of masters results in increased number of slots in TDMA, longer replenishment period in PBS and reduced allocated rate in the CCSP. As depicted in Fig. 4.8 and Fig. 4.9, the WCET produced under TDMA/PD, PBS and CCSP all converge as the allocated bandwidth is reduced. Here, for the most of the time, the master is waiting for its slot under TDMA/PD or it is waiting to be replenished under PBS and CCSP. Hence, the WCET produced under these arbiters is similar. This supports our analysis that the highest priority master under PBS and CCSP benefits only if the allocated bandwidth satisfies the master's memory needs.

The RR produces, as expected, a linearly increasing WCET with reduction in the allocated bandwidth. Here, more number of masters results in more interference. Interestingly, under the TDMA/PD, reduction in the allocated bandwidth from 1/4 to 1/5 results in reduction in the WCET. This counter intuitive behavior is explained in the Chapter 6. Due to the statically assigned priorities and independence from the allocated bandwidth, the PD^{h1} and SP produce constant WCET.

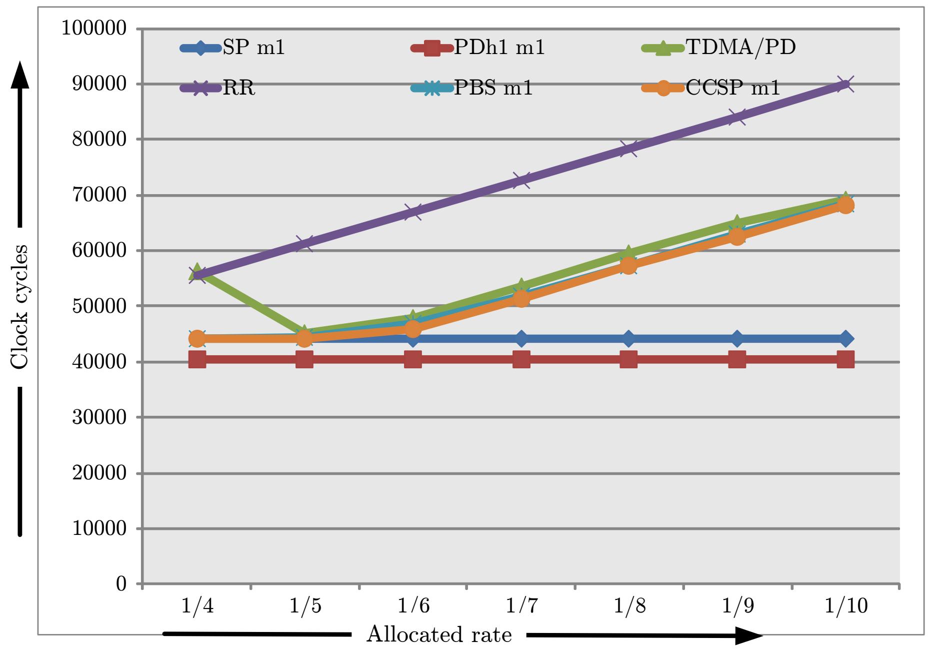


Figure 4.9: Effect of reduction in allocated bandwidth on the WCET of `quart` application

4.5.3 Comparison of the WCET produced by the $\mathcal{LR}$ and the detailed analysis

In this subsection, we compare the WCET produced by the $\mathcal{LR}$ analysis (equations (4.8) to (4.10)) and the detailed analysis (Algorithms 3 and 4) under the CCSP arbiter for the applications from the Mälardalen WCET benchmark suit. These are the same applications used in the Sec. 3.3.1 and Sec. 4.5.1.

The Table 4.3 depicts that the WCET produced by the $\mathcal{LR}$ analysis is up to 2.71 times larger (compress $m4$) than the WCET produced by the detailed analysis. As shown in our previous work [41], the overestimation for memory intensive applications can be as large as 7 times. The reasons for this over estimation are explained in Sec. 4.3.3. Notice that the over estimation increases as the priority is lowered. This is due to the Equation (4.10). Here, the lower priority master considers interference also from the partial credits of higher priority masters.

4.6 Comments on certification

According to MCP_Determinism_12, the memory usage of applications must be known and must stay below or equal to the available memory bandwidth. As stated in Sec. 3.4, compu-

Benchmark	$WCET_{lr}/WCET_{det}$			
	$m1$	$m2$	$m3$	$m4$
compress	1.76	1.65	1.64	2.71
cover	1.79	1.68	1.71	2.67
crc	1.30	1.31	1.35	1.78
duff	1.47	1.43	1.49	2.25
edn	1.67	1.61	1.62	2.45
expint	1.67	1.56	1.61	2.56
fac	1.22	1.21	1.26	1.80
fdct	1.66	1.55	1.60	2.66
fibcall	1.26	1.26	1.27	1.77
fir	1.46	1.44	1.49	2.14
jane	1.33	1.30	1.35	2.00
jfdcint	1.42	1.36	1.46	2.23
matmul	1.69	1.62	1.66	2.49
minver	1.56	1.49	1.51	2.44
ludcmp	1.58	1.51	1.51	2.47
prime	1.41	1.37	1.42	1.89
quart	1.55	1.48	1.49	2.41
recursion	1.15	1.16	1.17	1.41
ud	1.64	1.55	1.57	2.54

Table 4.3: Overestimation of the $\mathfrak{LR}$ analysis compared to the detailed analysis

tation trace can be used to determine the memory requirements of applications. According to MCP_Determinism_14, the maximum capacity of the resource (here, shared SDRAM) must be known. For the available SDRAM bandwidth in the worst case, the read/write switch and worst case refresh penalty must be considered. This has been taken care of in Sec. 4.1.2 and in equation (4.1). Note, that the worst case parameters are directly extracted from the targeted board (Fig. 4.7).

Similarly, detailed worst case interference analysis provided through various equations and algorithms in this chapter are compliant to MCP_Determinism_7 and suggested activities: MCP Interference Channels (4.(a), 4.(b) and 5.(a)).

4.7 Summary

This chapter and the previous chapter have presented basic operation of various arbiters and their worst case latency analysis. The worst case latencies were incorporated in the WCET calculation of application-paths. From the analysis and results, the arbiters can be characterized on the following parameters.

Arbiter	COTS	WCET Bound	Shared Resource Utilization	Hardware Complexity	Analysis Complexity
SP	++	+	++	++	++
TDMA	++	++	--	++	++
RR	++	--	++	++	++
PD	+	++	+	++	++
PD^{h1}	+	++	+	++	++
PBS	--	+	+	--	--
CCSP	--	+	+	--	--

Table 4.4: Characteristic comparison of Arbiters

The SP arbiter is simple and provides high resource utilization due to its *work conservativeness.* It supports only one HRT application (executing on the highest priority master). Here, the HRT may starve other SRT applications by monopolizing the shared resource, hence, for lower priority masters the WCET is infinite. However, HRT applications are designed very carefully, hence, probability of such behavior is very low except in faulty situations. If an HRT becomes faulty, it may not matter much if co-existing SRT applications achieve their dead-line or not since heavy damage or catastrophe may occur due to the faulty HRT.

The TDMA arbiter is predictable and simple. However, due to strict static slot allotment, it results in poor shared resource utilization.

The RR arbiter is simple, starvation free and work conserving. However, due to the employment of dynamic slot assignments, it produces large WCET bounds. If the system contains multiple SRT applications only, then the RR is the preferable arbiter.

The PD arbiter is not a COTS component, however, it is simple to design and analyze. It produces as low WCET bounds as that produced by the TDMA and provides better shared resource utilization than TDMA. If the system consists multiple HRT applications, then the PD is the preferable arbiter. PD ^{h1} supports only one HRT application and produces the lowest WCET bound for the HRT among all arbiters presented in this thesis. Thus, if the system consists only one HRT application, then the PD^{h1} is the preferable arbiter.

Both PBS and CCSP produce tight WCET bound for the highest priority master provided that the allocated bandwidth to them satisfies the memory requirements of the highest priority application. As priority is lowered, the WCET grows significantly. Moreover, both the arbiters are significantly complex to design and analyze.

Chapter 5

Measuring WCET of applications executing on Cots multi-core architectures

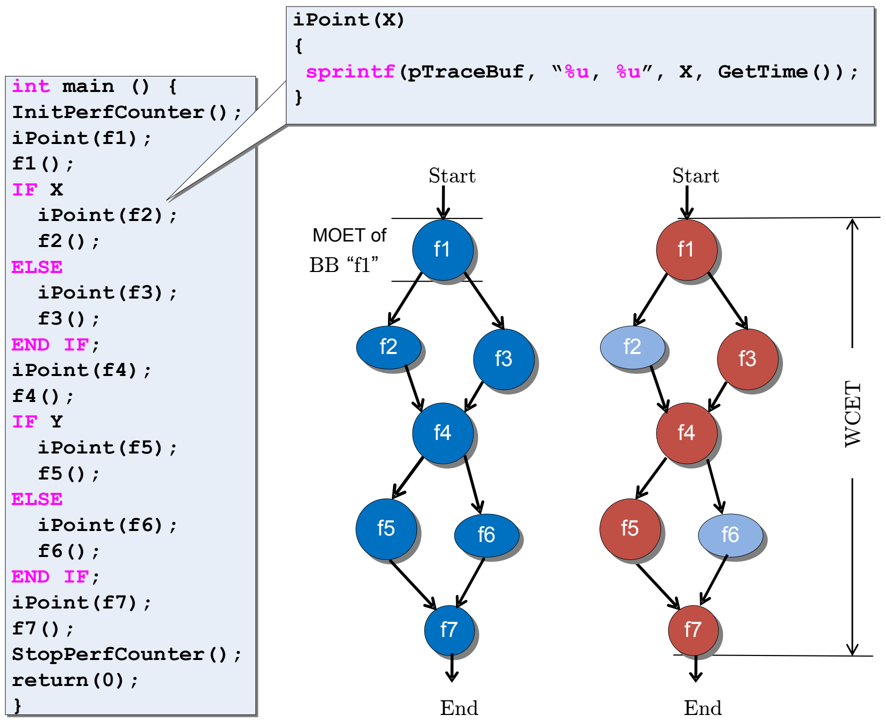
The previous chapters have analyzed advantages and drawbacks of conventional arbiters, Static Priority (Sp), Time Division Multiple Access (Tdma) and Round robin (Rr), and unconventional arbiters, Priority Division (Pd), Priority-based Budget Scheduler (Pbs) and Credit Controlled Static Priority (Ccsp) arbiters. The previous chapters have concluded that the Pd arbiter produces the lowest Wcet bound at slight penalty in the *worst case resource utilization*.

In spite of these advantages and its simple architecture, it may be prohibitively expensive for semiconductor industry to replace existing conventional arbiters and mass produce new chips with the Pd arbiter. Additionally, every hardware change enforces re-certification of the architecture in the safety-critical hard real-time domain. Hence, this chapter[1] presents a technique to measure the Wcet of applications executing on Cots multi-core architecture. The technique enforces modifications in neither the mass produced Cots devices nor the commercialized measurement based timing analysis tools. Only a minor modification in *test chips* of Cots components is required. Thus, the tool qualification costs are also avoided.

5.1 Background

This section explains the measurement based Wcet analysis for applications executing on single-core architectures. The section also explains the unsuitability of this technique for ap-

[1]The chapter includes figures and results from our previous work [2]



```
iPoint(X)
{
 sprintf(pTraceBuf, "%u, %u", X, GetTime());
}
```

```
int main () {
InitPerfCounter();
iPoint(f1);
f1();
IF X
  iPoint(f2);
  f2();
ELSE
  iPoint(f3);
  f3();
END IF;
iPoint(f4);
f4();
IF Y
  iPoint(f5);
  f5();
ELSE
  iPoint(f6);
  f6();
END IF;
iPoint(f7);
f7();
StopPerfCounter();
return(0);
}
```

(a) Pseudo code (b) Control flow (c) Worst case path f1-f3-f4-f5-f7

Figure 5.1: Measurement based WCET analysis – RapiTime approach

plications executing on multi-core architectures with a dynamically scheduled shared memory arbiter.

5.1.1 Hybrid Measurement Based WCET analysis

The hybrid measurement based WCET analysis technique is employed by RapiTime tool from Rapita systems Ltd[1]. The technique augments execution time measurements with static code analysis.

At first, the application source code is statically analyzed to identify function boundaries, conditional structures and control flow in order to create control flow graph. For example, the pseudo code of Fig. 5.1(a) is analyzed to build the control flow graph of Fig. 5.1(b) [2]. In the analysis step, Basic Block – BB[2] are identified and given a unique id. Later, instrumentation points (`iPoint()`) are inserted at the beginning and end (in the figure, shown only at the beginning) of each BB. An `iPoint()` simply saves its time of execution in a trace. Thus,

[1]http://www.rapitasystems.com/
[2]A linear code segment with a single entry and a single exit points.

by deducting start-time from end-time, execution time of any Bb can be determined. The instrumented code is executed multiple times on the targeted hardware in order to achieve higher coverage.

RapiTime analyzes the recorded trace (aka `iPoint()` trace) offline. During multiple executions, a single Bb may experience different execution times. Hence, an execution time profile for every Bb is created. From the profile, the highest of the recorded execution times is termed as Moet– Maximum Observed Execution Time. The control flow graph is then populated by the Moet of each basic block. In Fig. 5.1(b), bigger circles indicate higher Moet.

To determine the worst case path, all paths from `Start` to `End` are analyzed and Moet of all Bb falling on each path is summed up. The path with the highest sum is considered the worst case path and the sum is considered the Wcet of the analyzed application.

Following are the advantages of the technique: i) Cost of building hardware model of the target platform is avoided since the platform itself is used for measurements. ii) Test coverage information is also available together with timing information. iii) Optimization hotspots for reduction in Wcet are immediately identified, i.e. Bb on the worst case path.

Despite the above mentioned advantages, the techniques is unsuitable to multi-core architectures with dynamically arbitrated shared memory.

5.1.2 Unsuitability of the technique to multi-core architectures with dynamically arbitrated shared memory

Consider that the shared memory is arbitrated by Round Robin – Rr[1] (Sec. 3.1.7) policy. Under Rr, the experienced latency of a cache miss depends on the interference on the shared memory. Thus, co-existing applications can heavily influence the Moet of basic blocks. Here, execution time of any Bb cannot be guaranteed to be its Moet (since it also depends on activity of co-existing applications). This leads to invalidation of the constructed worst case path and Wcet.

5.2 Worst Case Interference Compensated WCET Measurements

This section explains our technique of extending the measurement based Wcet analysis (Sec. 5.1.1) to multi-core architectures. Our technique adds a tiny observation module to correctly com-

[1]The argument is true for any dynamic arbitration policy.

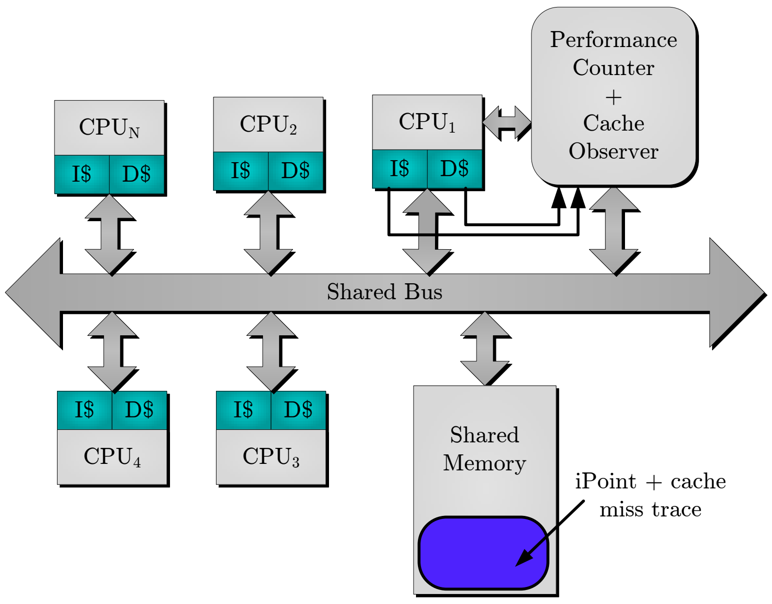


Figure 5.2: Cache Observation and cache miss trace generator

pensate for the worst case interference according to the observed cache misses. The module is active only during analysis and turned off during a deployed application execution. Hence, test chips (Sec. 5.2.5) are the perfect target for our technique.

The first subsection explains the basic technique and the latter presents an optimized version.

5.2.1 Cache observation and cache miss trace generation

The Fig. 5.2 [2] depicts a *cache observer* module on a multi-core architecture with N number of cores. The module contains a counter, a cache observation unit and a bus master interface.

The instrumented code is executed on this architecture (CPU_1 in this case) and `iPoint()` trace is captured in the memory as explained in Sec. 5.1.1. During the instrumented execution, the module observes caches and records occurrence time and experienced latency of each shared memory access (due to a cache miss). The experienced latency, $L_i \in [B_L, W_L]$[1]. Now two traces are available, an `iPoint()` trace and a cache miss trace. The two traces are merged to form a combined trace. The Fig. 5.3(a) depicts a combined trace of single Bb.

The host machines processes the combined trace offline before invoking a RapiTime instance. In the first step, experienced latencies are removed from the combined trace to obtain a

[1]Note that in this setup, the L_i could be more that W_L since additional interference from the cache observer must be considered. However, during application deployment, the cache observer is turned off. Hence, W_L value should only consider co-existing cores.

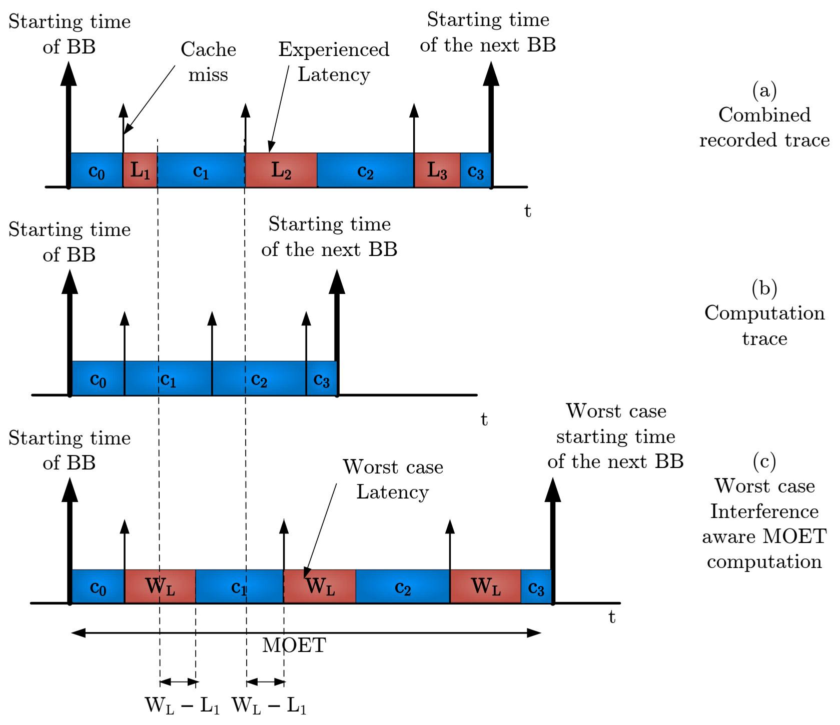


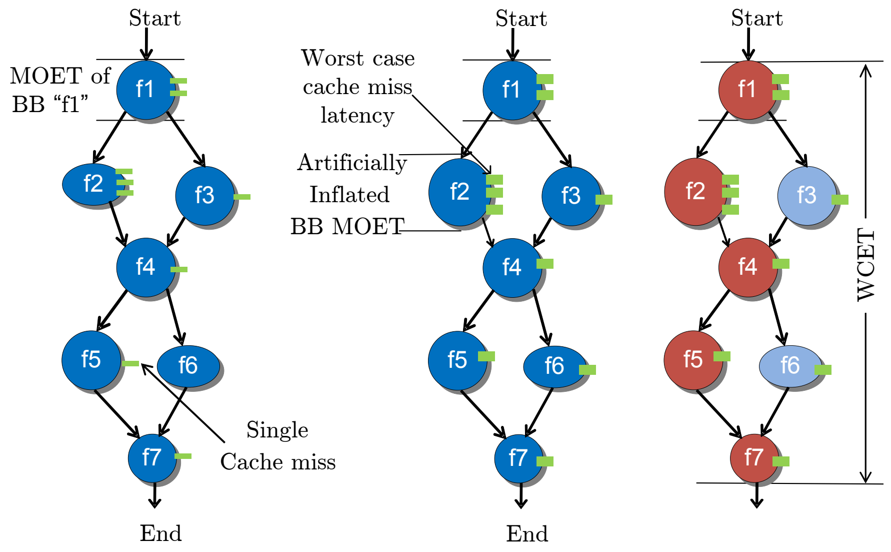
Figure 5.3: Pre-processing of trace

computation trace (Fig. 5.3(b), Sec. 3.1.4). Later, as explained in Sec. 3.1.4, theoretical worst case latency, W_L is inserted for each cache miss shifting subsequent cache misses to the right. This results in the worst case interference compensated inflation of MOET of basic blocks. The resulting trace is depicted in the Fig. 5.3(c).

The Fig. 5.4 [2] depicts the offline trace manipulation on the host machine. The Fig. 5.4(a) is the captured combined trace. The Fig. 5.4(b) is the control flow graph with worst case compensated basic block MOET. RapiTime takes the worst case compensated trace of Fig. 5.4(b) as input and construct the worst case path and corresponding WCET. Thus, the estimated WCET is worst case interference compensated. Note that for the hypothetical application of Fig. 5.1, the single-core worst case path is f1-f3-f4-f5-f7. For the same application, the multi-core worst case path is f1-f2-f4-f5-f7. This is due to higher number of cache misses, as observed, on that path.

Advantages: Following are the advantages of our technique:

1. Performance of a COTS chip remains intact since our unit is only an observation unit.



(a)
Control Flow Graph with combined trace

(b)
Control Flow Graph with artificially inflated BB MOET

(c)
Worst case path, f1-f2-f4-f5-f7, considering W_L for each cache miss.

Figure 5.4: Worst case interference aware WCET estimation

2. WCET analysis under advanced arbiters (PBS, CCSP and DPQ) is enabled through the algorithms presented in Sec. 4.2, Sec. 4.3 and [94].
3. Minor modification in an existing architecture is required.
4. The above mentioned modification is limited to "test chips" only. Hence, production chips remain unchanged.
5. The existing tools for timing analysis on single-core architectures, e.g. RapiTime, do not need porting to multi-core.

Disadvantages: There are three drawbacks of the above mentioned technique:

1. Memory intensive applications have too many cache misses which may overflow a trace memory or restrict analysis to small parts.
2. The operating frequency of interconnect between a shared memory and processors is inversely proportional to the number of connections on the interface. Here, an additional connection of the cache observer may reduce the frequency due to the increased capacitive loading.

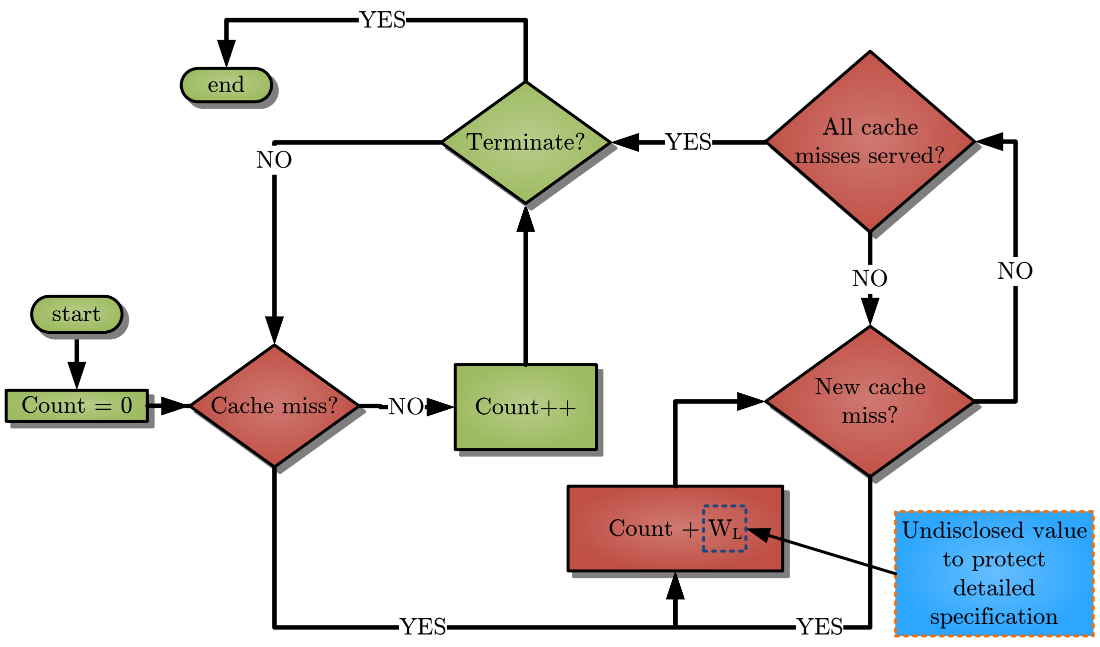


Figure 5.5: Optimized cache observer

3. Precise specification of the underlying arbiter and the shared memory must be known in order to do the worst case latency analysis. COTS vendors are reluctant to disclose these specifications in order to preserve their competitive advantage.

These drawbacks are mitigated in our optimized solution which is presented in the next subsection.

5.2.2 The Optimized cache observation module

Our optimized technique consists of only a modified counter and a cache monitoring logic. As depicted in Fig. 5.5 [2], our module continuously observes the caches (I$ and D$). If a cache miss occurs, theoretical worst case latency, W_L, is immediately added to the `count` register. The counting is suspended until the cache miss is served. Meanwhile, if another cache miss occurs (due to out of order execution), `count` is again incremented by W_L. The `count` resumes standard performance counter operation, increment by one on every clock cycle, as long as there are no remaining cache misses to be served.

Our technique can be explained by an example. Assume that a cache miss occurs at `count` value t. The experienced latency of the cache miss is L_i. Hence, after the cache miss is served the `count` value, in absolute time, is $t + L_i$. Instead of considering value $t + L_i$, we consider the artificially forwarded value $t + W_L$. The difference between the forwarded time and the absolute time is $W_L - L_i$. Here, instead of forwarding the time on the host machine (as in the basic technique of Sec. 5.2.1), the time is already forwarded as soon as a cache miss occurs. Thus, the `GetTime()` method of Fig. 5.1 returns already worst case interference compensated time. So, the generated `iPoint` trace is also worst case interference compensated. Thus, the WCET estimated by RapiTime is also worst case interference compensated.

The optimized cache observer can be configured via an API to supply additional information, for example, number of enabled[1] co-existing masters.

Advantages: Apart from the advantages of the basic technique, the optimized technique has the following additional advantages.

1. In Fig. 5.5, the green blocks belong to the native performance counter operation. Only the red blocks are added to it in order to estimate the worst case latencies due to the worst case interference. This results in ultra small area overhead.

2. Due to the absence of an additional bus master, additional capacitive loading on the shared bus is avoided.

3. The semiconductor vendor can hide the specification by adding the W_L to the `count` internally without disclosing its value. For budget based arbiters, hardware version of algorithms presented in Sec. 4.2 and Sec. 4.3 can be implemented at minor area overhead. The hardware implementation then supplies the appropriate W_L value (depending on the employed arbitration scheme, shared memory specification, number of enabled co-existing masters and access history) in real-time for the current cache miss.

5.2.3 Simulation as an alternative?

The technique described in Sec. 5.2.1 and Sec. 5.2.2 can also be implemented in simulation model of a processor architecture instead of test chips. However, there are following drawbacks of applying the technique to simulation models instead of test chips.

1. The technique needs Cycle Accurate Byte Accurate (CABA) model of the processor architecture. Generally, these models are unavailable to application developers. Additionally, as explained earlier, the detailed specification of the underlying arbiter and the shared memory are undisclosed by the chip manufacturers to protect their competitive advantage.

[1]To avoid interference to highly critical tasks, designers may pause co-existing masters until the critical task is finished.

2. Depending on the application-under-test, CABA models of peripherals such as sensors and actuators may be required to estimate the Worst Case Response Time (WCRT) of the system.

3. It must be proven that the CABA models accurately model the physical processor core and peripherals. In the worst case, the CABA model may have to follow the tool qualification process which is tremendously expensive.

4. Typically, executing applications on a CABA simulation model is very slow which results in a long measurement time for each measurement iteration.

5.2.4 Overestimation of our technique

Our technique inherits the drawback of intrusiveness from the hybrid WCET measurement based technique. The instrumentation points impact the measured WCET of test-application. Since the number of clock cycles consumed by an `iPoint` (N_{ip}) is constant[1], it may be intuitive to deduct $2 \times N_{ip}$ from measured BB times (remember that an `iPoint` is inserted at the beginning and end of a BB). This will compensate for the number of clock cycles consumed by `iPoints`, however, it does not compensate for the changed cache state due to instrumentation code. The instrumentation code occupies space in instruction cache which leads to reduced available space in the cache for the test application. This results in more number of cache misses during instrumented code execution than during non-instrumented execution. This directly contributes in increasing the measured WCET of test applications. Additionally, there is an indirect impact. The branch to `iPoint` is always taken. This may impact the operation of history based branch predictors which may behave differently during non-instrumented execution.

Although assuming W_L for every cache miss sounds pessimistic, it is the only safe assumption if the analysis presented in our previous work [95] is unavailable.

It is interesting to investigate the impact of instrumentation on the measured WCET. Note that the overhead on WCET originates from the trace based technique (hybrid WCET measurement) and not from our technique. To determine the overhead on WCET, we conducted tests using instrumented code and non-instrumented code both. At first, using the instrumented code execution and RapiTime analysis, we determined the worst case path and corresponding WCET. We then execute the same path and measure start to end time using the `GetTime()`

[1]If cached and separately implemented in a function.

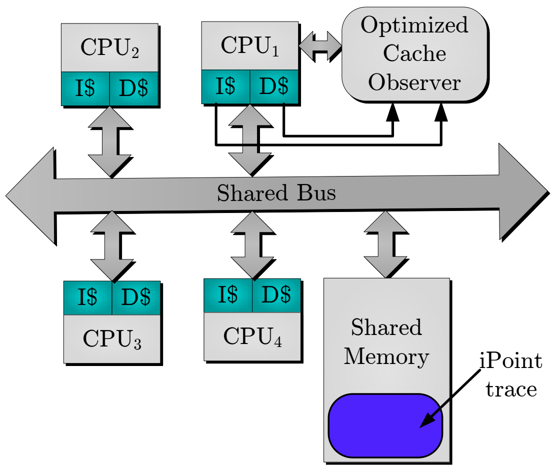


Figure 5.6: Test Architecture.

method and denote it as $WCET_{ni}$ (WCET of non-instrumented application code). In both the cases, our optimized cache observer compensates for the worst case interference.

We are aware that during non-instrumented code execution, some other path could be the worst case path then during the instrumented code execution. To mitigate this problem, intensive measurements of execution times from start to end must be performed to achieve as much test coverage as possible.

5.2.5 Test chips

This subsection briefly explains the design flow with test chips. Test chips are manufactured in low volumes by semiconductor vendors with additional debug facilities which are unavailable on production chips. The product/application developers buy these chips and test their applications using advanced on-chip debugging infrastructure. After successful test runs, production version of the test chips are ordered in large volume and deployed in commercial products.

The test chips are also known as emulation devices. An example of the emulation devices from Infineon TriCore is presented here[1]

5.3 Test Cases

In this section we explain our test architecture and discuss the results. The goal is to test our technique with real benchmark applications executing on a multi-core architecture.

[1]http://www.isystem.com/downloads/winIDEA/help/index.html?OCDTriCore.html

Architecture	Logic Elements
Without the Cache observer	13555
With the Cache observer	14272

Table 5.1: Synthesis results. Source: [2].

5.3.1 Test Architecture

Our test architecture (Fig. 5.6 [2]) consists of a quad-core processor composed of Nios II F cores. The cores have I$ and D$ each of 512 Bytes and 32 Bytes cache-line size. The shared main memory and trace buffer are mapped to on-chip memory. The current set-up considers only L1 caches. An extension to our technique under shared L2 caches is explained in the Sec. 5.4. The test architecture is built on Altera cyclone III development board and operates at 125 MHz frequency.

In our set-up, test applications are executed on core1 and memory stressing applications are executed on co-existing cores. We selected multi-path applications from the Mälardalen WCET benchmark suit [87] (results of single path applications can be found in Sec. 3.3).

Numbers in table 5.1 highlight area overhead of our technique. The impact of our technique on the consumed on-chip resources is ≈5%. Note that a COTS multi-core architecture is a complex device composed of processor cores, memories, hardware accelerators, DMAs, I/O controllers etc. Compared to that device the area overhead of our technique will be much less than 5%. Additionally, the area overhead is limited to test chips only.

5.3.2 Results

In Table 5.2 and Table 5.3 our test results are presented. There two main parts in the tables. The first part presents results from instrumented execution and the second part presents results from non-instrumented execution. WCET during non-instrumented execution is obtained as explained in Sec. 5.2.4. In both the parts, the WCET is worst case interference compensated.

In the above tables, the maximum of observed execution times, under varying stress pattern, is denoted by MOET of applications. WCET of an instrumented application without any shared memory interference (single-core equivalent) is denoted by $WCET_s$ while WCET of an instrumented application on our quad-core test architecture is denoted by $WCET$. The factor $WCET/WCET_s$ denotes the increase in WCET if an instrumented application is ported from single-core to multi-core. Similarly, $WCET_{ni}/WCET_{nis}$ denotes increase in

Bench-mark	Instrumented Execution				Non-instrumented Execution			$\frac{WCET}{WCET_{ni}}$
	WCET	MOET	$WCET_s$	$\frac{WCET}{WCET_s}$	$WCET_{ni}$	$WCET_{nis}$	$\frac{WCET_{ni}}{WCET_{nis}}$	
Binary search	2857	2662	1729	1.65	775	415	1.86	3.68
BBSort	133570	111935	72896	1.83	20941	20362	1.02	6.37
cnt	54930	35182	28435	1.93	8900	7772	1.14	6.17
insertsort	20741	17310	10081	2.05	4096	3590	1.14	5.06
ludcmp	515703	423071	241266	2.13	456778	220309	2.07	1.12
ndes	1468191	1161950	697753	2.10	623355	295531	2.10	2.35
ns	194401	163983	119977	1.62	83165	36555	2.27	2.33
nsichneu	183104	152792	86492	2.11	64738	25996	2.49	2.82
qsort-exam	101364	82574	46289	2.18	74008	31887	2.32	1.36
select	153437	131381	60855	2.52	47025	21867	2.15	3.26
ST	59384913	45377728	26377862	2.25	51651564	23957615	2.15	1.14

Table 5.2: Test Results: Execution Times in Clock Cycles, 512 Bytes I$ and D$. Source: [2].

WCET if a non-instrumented application is ported from single-core to the quad-core. The factor $WCET/WCET_{ni}$ represent overhead of instrumentation.

Bench-mark	Instrumented Execution				Non-instrumented Execution			$\frac{WCET}{WCET_{ni}}$
	WCET	MOET	$WCET_s$	$\frac{WCET}{WCET_s}$	$WCET_{ni}$	$WCET_{nis}$	$\frac{WCET_{ni}}{WCET_{nis}}$	
Binary search	2382	2093	1553	1.53	705	395	1.78	3.37
BBSort	71618	68254	60293	1.18	20704	20294	1.02	3.45
cnt	27405	25935	22701	1.20	7975	7612	1.04	3.43
insertsort	7789	7686	6923	1.12	4026	3570	1.12	1.93
ludcmp	205085	184567	159866	1.28	182309	146528	1.24	1.12
ndes	748654	655750	523350	1.43	357602	209319	1.70	2.09
ns	130930	122743	106831	1.22	37794	35266	1.07	3.46
nsichneu	126971	105534	73221	1.73	54383	23563	2.30	2.33
qsort-exam	33582	31957	29058	1.15	19189	17162	1.11	1.75
select	30938	29447	26769	1.15	15042	13383	1.12	2.05
ST	25334087	19541872	17578497	1.44	19209880	15164451	1.26	1.31

Table 5.3: Test Results: Execution Times in Clock Cycles, 4 KBytes I$ and D$. Source: [2].

It is clear from $WCET_{ni}/WCET_{nis}$ that the WCET is increased when an application is ported from a single-core to multi-core architectures. Nowotsch et al [96] and Radojkovic et

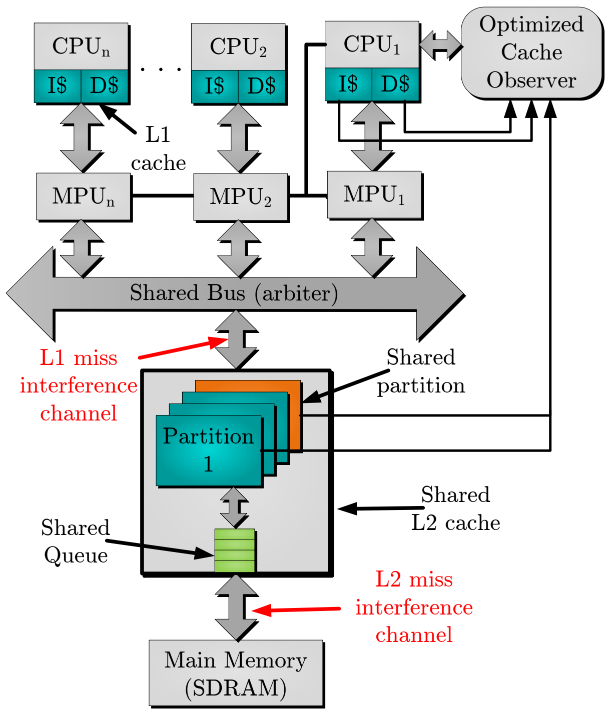


Figure 5.7: MPU protected and partitioned shared L2 cache

al [97] also observed the same effect. This increase is due to the interference on shared memory and proportional to number of accesses to the shared memory. Increasing cache sizes may decrease number of cache misses (capacity misses) and result in small $WCET_{ni}/WCET_{nis}$ factor. This fact is visible in Table 5.2 and Table 5.3. The average $WCET_{ni}/WCET_{nis}$ factors in the tables are 1.89 and 1.35, respectively. Remember that the Table 5.3 presents results from tests with larger cache size (4 KB).

Similar to $WCET_{ni}/WCET_{nis}$, overhead of the instrumentation also decreases when bigger caches are employed. As stated earlier, instrumentation code changes cache state and occupies space in caches which results in reduced available space for application code. However, if bigger caches are employed, this impact is diluted. This is seen in the tables where average value of $WCET/WCET_{ni}$ drops from 3.24 (table 5.2) to 2.39 (table 5.3).

5.4 Scaling to multi-level cache hierarchy

COTS processors have multi-level cache hierarchy. Typically, the first level (L1) caches are private and higher levels are shared. A cache miss occurring in one level is, if data available, served by the next higher level. If the requested data is not available, the next higher level is searched until the off-chip memory (usually an SDRAM) is reached. In multi-level cache hierarchy, since higher levels are shared among cores, it may be possible that one core evicts useful data of other cores from the shared cache. Hence, for the worst case analysis, for every L1 cache miss, higher level cache misses must be assumed. This increases WCET of applications tremendously.

In the Fig. 5.7, a multi-level cache hierarchy is presented. The shared L2 cache is protected by Memory Protection Unit (MPU). After reset, one of the trusted cores programs all MPUs. Since each core can only access a particular address space, the shared L2 cache is logically partitioned. Now one core can access data only in its allocated partition and the shared partition. Thus, one core cannot evict data from the partition allocated to another core.

The shared partition is used for symmetric multi-processing. Since every core can access the shared partition, the data within the partition is unpredictable. Hence, every access to the shared partition should be considered a miss.

In the Fig. 5.7, our cache observer monitors L1 cache, L2 cache partition dedicated to the core1 and also the shared partition. If a cache miss originates from the partition dedicated to core1, the `count` is incremented by the worst case latency to access the off-chip memory. Irrespective of shared partition hit or miss, the `count` is incremented by the worst case latency to access the off-chip memory if core1 accesses the shared partition since data in shared partition is unpredictable. Cores must synchronize before accessing the shared partition.

The shared cache partitioning through MPUs is *not* limited to test chips only. It must stay in commercial products as well. This is an expensive modification. Another approach to achieve the logical partition is by means of Memory Management Unit (MMU) of each core. It must be made sure that each core configures its MMU correctly after reset in order to achieve a correct partition.

5.4.1 Operation and interference channels of the shared L2 cache

Although we employ a strict partition based sharing, there are two interference channels as depicted in Fig. 5.7. A pure L1 miss (L2 hit) must consider interference from L1 misses originating from co-existing cores. Similarly, an L2 miss must consider interference from queued L2 misses (accesses to off-chip memory) originating from other partitions.

If the L2 cache processes only one request at a time, in the worst case, each pure L1 miss (L2 hit) must assume that the cache is busy in processing L2 misses from co-existing cores.

This tremendously increases the worst case latency to access a shared L2 cache (irrespective of L2 hit or miss). To mitigate this problem, our proposed L2 cache operates in the following way:

1. A separate mechanism to serve L2 hits and L2 misses. Hence, the hit mechanism stays operational while a miss is being served, and vice versa.
2. The shared bus arbiter (Fig. 5.7) and the L2 cache are coupled with each other. As soon as an L2 miss occurs, the miss manager informs the shared bus arbiter to block the respected core from sending further requests. In fact, its subsequent slot are evicted from the arbitration schedule effectively blocking the core until its L2 miss is served. This simple modification can be performed easily on either PD or RR arbiters and it is beneficial to L2 hit producing cores since they can utilize unused slots of the blocked core.
3. An L2 cache miss is detected and queued for off-chip access within the arbitration slot in which it occurred. This eliminates the need of an arbiter for the shared queue. The size of the queue is bounded (= `NumberOfCores`) since a core is blocked until its L2 miss is served.
4. The co-existing cores stay operational as long as their accesses result in L2 hits since they are served by the hit manager (separately).
5. Both, hit and miss managers respect LRU policy.

The above mentioned operation principal has the following advantages.

1. The worst case interference analysis is straight forward since an L2 hit can only be interfered by other L2 hits and an L2 miss can only be interfered by other L2 misses. This also significantly reduces worst case penalties and thereby, WCET.
2. Bounded shared queue size. Bigger queue and unblocking operation is beneficial for average case, however, they degrade the worst case timing behavior.
3. Simple architecture and clear identification of interference channels.

5.5 Comments on certification

It is clear that the techniques presented in this chapter considers worst case interference for estimating the WCET of applications. This follows the suggested activities: MCP Interference Channels (4.(a), 4.(b) and 5.(a)) related to MCP_Determinism_7.

Here, employment of a shared L2 cache is more interesting. According to MCP_Determinism_10, the usage and strategy to manage the shared cache must be stated. According to MCP_Determinism_11, the worst case effects of using shared cache must be described and analyzed. Our L2 cache design and the presented worst case interference analysis satisfies these requirements. Our technique of WCET estimation considering the worst case interference is related to the suggested activities: Shared Memory and cache (5.(c), 5.(d) and 5.(e)).

5.6 Summary

This chapter has presented a novel technique of measurement based WCET analysis for applications executing on multi-core architectures. The technique uses unchanged single-core timing analysis tools for measuring WCET on multi-cores. As a demonstration, multi-path applications from the Mälardalen benchmark suit has been selected and their WCET is measured using COTS single-core timing analysis tool – RapiTime. The applications are executed on a quad-core NIOS processor built on Cyclone III FPGA. The approach, being only an observation technique, does not reduce performance of a multi-core processor and the implementation is limited to test chips only. Thus, the commercial chips remain unchanged. The technique is unique since it does not demand modifications in either COTS chips or COTS timing analysis tools. Additionally, it enables analysis in isolation (assuming the worst possible interference from co-existing cores) making it a convenient and an economical method.

The scalability of the technique is presented for multi-level shared caches. The technique is also compatible with any starvation free arbiter and finite response time shared memory.

The certification comments provided in the chapter show how the measurement technique and shared L2 cache management are, in principle, certifiable for avionics.

Chapter 6

Caveats in building predictable multi-core architectures

The previous chapters are focused on either providing low WCET at high resource utilization (PD arbiter), analyzing the worst case latencies under complex budget based arbiters or measuring the WCET of applications executing on COTS multi-core architectures. This chapter[1] presents some caveats in the WCET analysis especially for applications executing on multi-core architectures. Disrespecting these caveats can lead to optimistic WCET.

For example, it is proposed to measure the execution time of an application in the presence of artificially generated uninterrupted interference and consider the measured execution time as the WCET. In this chapter, we refute this approach and show that, counter intuitively, for some applications, the measured execution time in the presence of uninterrupted interference is even less than the Average Case Execution Time (ACET).

Typically, the WCET analysis/measurement is done at high level in industry. For example, WCET measurements is done directly on the target platform (Chapter. 5) accepting the given platform as it is. Similarly, WCET analysis is done on abstract models of platform and code. At this high level, many low level architectural details are abstracted, e.g. maximum latency to access off-chip memory. In this chapter, we will show that processor architectures built on popular interconnect specifications, e.g. AMBA [99], Avalon [85] and PLB [100], are simply unfit for WCET analysis in their native modes. On these architectures, the worst case latency for memory access is infinite.

[1]The chapter includes contents from our previous works [98, 86]

6.1 Timing anomalies due to the interference on the shared memory

Timing anomaly is a *counter intuitive* timing behavior. The term was first introduced by Lundqvist & Stenström [101]. They observed that in a dynamically scheduled processor, a cache hit at certain execution point could lead to longer execution time than a cache miss at the same point. Thus, the timing anomaly for processor architectures is defined as, *A processor architecture is said to be timing anomalous when a locally favorable event (e.g. cache hit) could result in a globally unfavorable event (e.g. longer execution time) and vice versa.* Reineke et al [102] present a formal definition of timing anomaly.

Chapters 3, 4 and 5 have highlighted through analysis and number of experiments that the co-existing applications on multi-core architectures prolongs execution of test-application. Instead of using the techniques presented in these chapters, an intuitive approach could be to make the co-existing applications stress the shared memory and observe its impact on execution time of the test application. Another intuition could be that the more number of co-existing applications stress the shared memory, the more interference is experienced by the test application. In this section, we show through theoretical analysis and practical evidences that none of these intuitions is universally true.

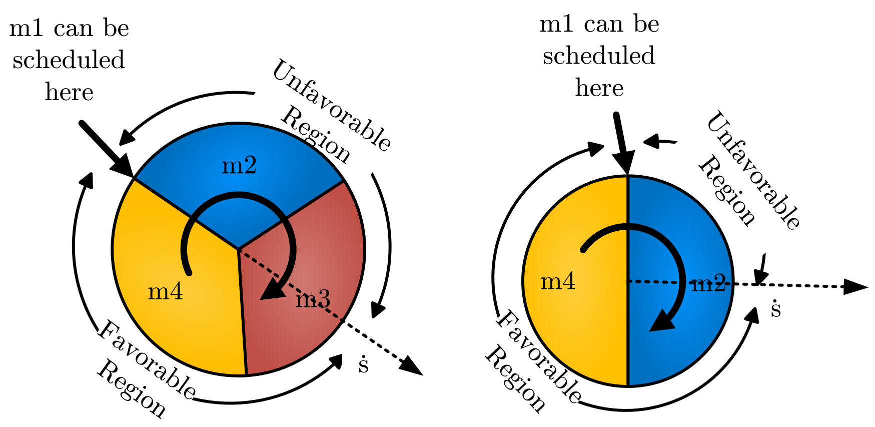
6.1.1 Latency analysis under shared memory stressing co-existing applications

This section analyzes effects on experienced latency of a test applications in the presence of shared memory stressing co-existing applications. Here, we considered Round Robin (Rr) as the shared memory arbiter, however, the observed phenomenon is valid for other arbiters as well (Sec. 6.1.6.3).

6.1.2 Alpha Interference

Definition: *The α interference is defined as the uninterrupted interference produced by α number of co-existing masters [86].*

Under α interference, co-existing masters either continuously send access requests to the shared memory or does not send any request at all. Thus, either they utilize their slots always,



(a) Three Interfering Masters, α = 3

(b) Two Interfering Masters, α = 2

Figure 6.1: Rotation of the Arbiter Pointer under Rr (not to the scale).

or they do not utilize them at all[1]. The test application executes on the master `m1`. This section analyzes experienced latency of application(s) executing on `m1` under the above mentioned interference scenario.

Although the α interference sounds a rare phenomenon, it is common to observe uninterrupted accesses from co-existing masters in the following scenarios: i) Cores fill-in there cache lines after a reset or after a new task is scheduled. For filling-up caches, dense traffic towards memory is generated. ii) In the presence of memory intensive I/Os, for example, cameras, radar etc. Our goal is to show that counter intuitively, test-application may benefit from the α interference and experience less than the average case latencies. Hence, simple measurements of execution time in the presence of uninterrupted interference from co-existing masters and considering the measured execution time as the worst case is unsafe.

We build our theory on a generic multi-core architecture and provide real-life evidences on the test architecture of the Fig. 5.6. Similar to previous chapters, here we assume that the test-application is executing on `m1` and co-existing masters are executing shared memory stressing applications.

[1]The assumption of an idle co-existing master helps us prove theoretically that the less number of aggressive co-existing masters may result in longer latencies to the test application than more number of aggressive co-existing masters.

6.1.3 Analysis

Two α scenarios are presented in Fig. 6.1 [86] under Rr arbiter. In the first scenario, $\alpha = 3$. Here, all co-existing masters, `m2`, `m3` and `m4` uninterruptedly access the shared memory. In the second scenario, $\alpha = 2$. Here, two co-existing masters – `m2` and `m4` access the shared memory uninterruptedly, `m3` remains idle. Hence, considering the *work conserving* operation of the Rr (Sec. 3.1.7), there are only two slots in Fig. 6.1(b). Recall that the theoretical values of B_L, W_L and A_L are derived considering all masters in the system, *irrespective* of number of active masters.

Note that the arbiter pointer rotation of the Rr arbiter now becomes deterministic in Fig. 6.1 – slots of `m2` and `m4` will always be utilized and the slot of `m3` will not be utilized ($\alpha = 2$). Similarly, under $\alpha = 3$, slots of all co-existing masters will always be used making the rotation of the arbiter pointer deterministic. Thus, latency to an access request from test-application can be determined by the access gap (computation times – c_i) as derived in the following equations.

This deterministic latency of i^{th} access under the α interference is denoted by DL^i_α and derived by the following equation.

$$DL^i_\alpha = (\alpha + 1) \times SS - \{c_{(i-1)} \bmod (\alpha \times SS)\} \tag{6.1}$$

Remember that $c_{(i-1)}$ is the time gap between previous access and the current access (computation time between two cache misses – Sec. 3.1.4). Consider the second part of the above equation as $\Theta^{(i-1)}_\alpha$.

$$\Theta^{(i-1)}_\alpha = \{c_{(i-1)} \bmod (\alpha \times SS)\} \tag{6.2}$$

$$DL^i_\alpha = (\alpha + 1) \times SS - \Theta^{(i-1)}_\alpha \tag{6.3}$$

The average of all DL^i_α values through the application execution path is denoted by $\overline{DL_\alpha}$. Similarly, average of Θ^i_α values is denoted by $\overline{\Theta_\alpha}$.

From equation (6.3),

$$\overline{DL_\alpha} = (\alpha + 1) \times SS - \overline{\Theta_\alpha} \tag{6.4}$$

For the Rr arbiter, from equations (3.8) and (3.9), $B_L = 1 \times SS$ and $W_L = N \times SS$, N is total number of masters in the system. From this information, the average-case latency of the i^{th} access, A^i_L, is given by the following equation,

$$A_L^i = \frac{N+1}{2} \times SS \tag{6.5}$$

$\overline{A_L}$ is the average value of all A_L^i values through the application execution. Since values on the right side in the equation (6.5) are constant, $\overline{A_L}$ can be given by the following equation.

$$\overline{A_L} = \frac{N+1}{2} \times SS \tag{6.6}$$

Equations (6.1) to (6.6) are used to infer the following counter intuitive timing behavior.

6.1.4 Anomaly − 1

From equation (6.2), $\Theta_\alpha^{(i-1)} \in [0, (\alpha \times SS - 1)]$. Assuming $c_{(i-1)} = n(\alpha \times SS) - 1$, $n \in \mathbb{N}^+$, $\Theta_\alpha^{(i-1)} = \alpha \times SS - 1$.

Putting the above mentioned Θ value in equation (6.3),

$$DL_\alpha^i = SS + 1 << A_L^i \tag{6.7}$$

Equation (6.7) proves that, counter intuitively, experienced latency could be much less than the average latency if $c_{(i-1)} = n(\alpha \times SS) - 1$, $n \in \mathbb{N}^+$. The Fig. 6.1 explains this scenario, graphically. The $\Theta_\alpha^{(i-1)}$ value, indicates the arrival of access respective to the circular schedule of arbitration. This divides the circular schedule in favorable and unfavorable regions. If an access request arrives close to (far from) its scheduling opportunity, it fall in favorable (unfavorable) region. Hence, the experienced latency is less (more) than the average case latency. If an application does majority of its accesses in the favorable region, then $\overline{DL_\alpha} < A_L$.

The Fig. 6.2 presents a hypothetical computation trace of a test-application that exhibits this phenomenon. In scenario A, the co-existing applications are aggressively accessing the shared memory while in scenario B, the co-existing applications are sparsely accessing the shared memory. Counter intuitively, the total execution time of the test application is longer in scenario B than in scenario A. This phenomenon is not only hypothetical, a real-life evidence is provided in Sec. 6.1.6.1.

Remember from Sec. 3.1.7 that prediction of arbiter pointer rotation under Rr is extremely difficult. Here, however, due to the uninterrupted accesses from the co-existing masters, the prediction becomes simple and the division of the favorable region and unfavorable region is possible. The boundary point between the regions can be given by the following equation.

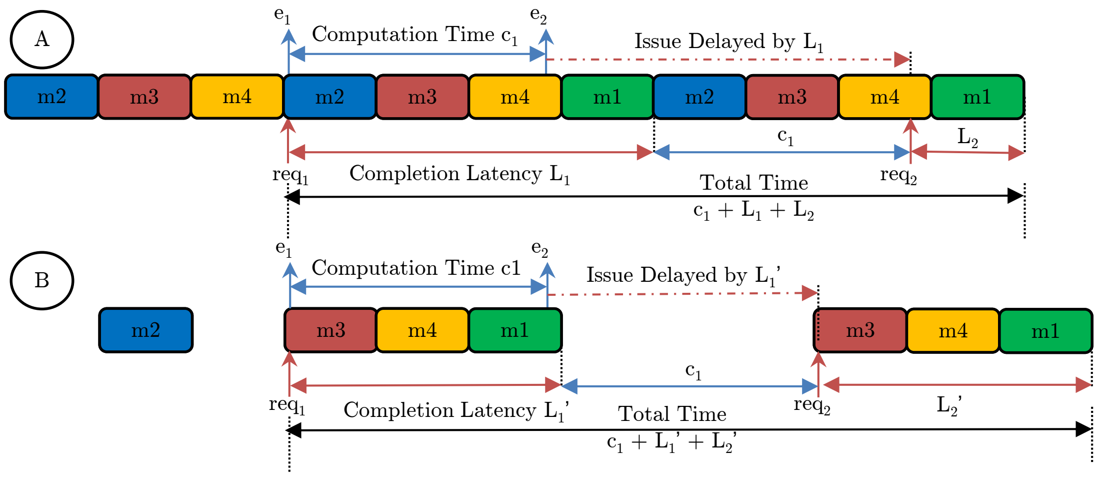


Figure 6.2: Counter intuitively, in scenario A, the execution time of the hypothetical test-application is shorter than in scenario B, although, the co-existing masters are aggressive in scenario A.

$$\dot{s} = W_L - A_L \tag{6.8}$$

6.1.5 Anomaly – 2

Using equations (6.1) to (6.6), we can prove that certain applications experience more interference in the presence of less number of aggressive co-existing masters than in the presence of more number of aggressive co-existing masters. Mathematically, $DL^i_{\breve{\alpha}} > DL^i_{\hat{\alpha}}$, $\breve{\alpha} < \hat{\alpha}$.

Assume, $c_{(i-1)} = (\breve{\alpha} \times SS)$. From equation (6.1), under $\breve{\alpha}$ interference,

$$DL^i_{\breve{\alpha}} = (\breve{\alpha} + 1) \times SS = \breve{\alpha} \times SS + SS \tag{6.9}$$

For the above mentioned $C_{(i-1)}$, the experienced latency under $\hat{\alpha}$ can be given by the following equation. Since since $\breve{\alpha} < \hat{\alpha}$, $c_{(i-1)}$ mod $(\hat{\alpha} \times SS) = \breve{\alpha} \times SS$. Hence,

$$DL^i_{\hat{\alpha}} = (\hat{\alpha} + 1) \times SS - (\breve{\alpha} \times SS) \tag{6.10}$$

$$DL^i_{\alpha} = (\hat{\alpha} - \breve{\alpha}) \times SS + SS \tag{6.11}$$

From equations (6.9) and (6.11), $DL^i_{\breve{\alpha}} > DL^i_{\hat{\alpha}}, \forall \hat{\alpha} : \breve{\alpha} < \hat{\alpha} \leq 2\breve{\alpha}$.

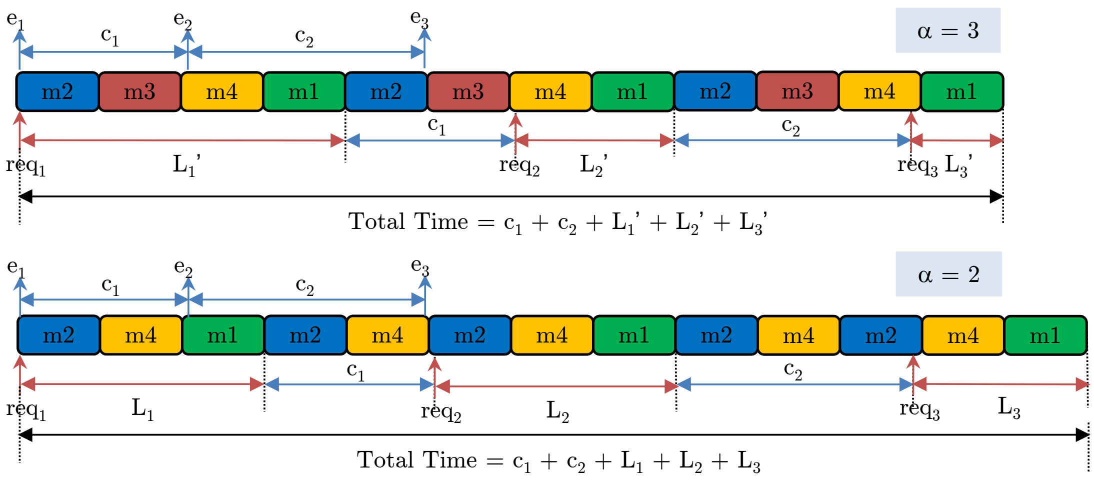

α = 3
e1
e2
e3
c1
c2
m2
m3
m4
m1
req1
req2
req3
L1'
L2'
L3'
Total Time = c1 + c2 + L1' + L2' + L3'
α = 2
L1
L2
L3
Total Time = c1 + c2 + L1 + L2 + L3

Figure 6.3: Execution time is longer under $\alpha = 3$ than under $\alpha = 2$ interference

The Fig. 6.3 explains this scenario using a hypothetical computation trace. Note that execution time under $\alpha = 2$ interference is longer than under $\alpha = 3$. Similar to the first anomaly, this anomaly is not only hypothetical and a real-life evidence is provided in Sec. 6.1.6.2.

6.1.6 Test cases

This section explores the real-life evidences of the timing anomalies derived hypothetically in the previous section. We chose single path[1] applications from the Mälardalen WCET benchmark suit [87] and conducted intensive experiments on multi-core architectures implemented on an Altera FPGA. We started with test architecture from chapter 3 and changed hardware parameters, such as number of cores and cache configurations, to find evidences of the above mentioned timing anomalies. Note that, unlike changing number of cores, modifying cache configuration modifies computation trace (change in c_i values). Hence, computation traces for each application was regenerated after modifying cache configuration.

As in previous chapters, test applications are executed on core1 and co-existing masters execute shared memory stressing application (similar to [103, 97, 96]). We step-by-step increased total number of cores from four to eight.

6.1.6.1 Test 1

The goal of this experiment is to explore an evidence of existence of the timing anomaly – 1 (Sec. 6.1.4) in real-life applications. In our intensive testing process, we found evidences of the

[1] Analysis of multi-path applications is presented in chapter 5.

Benchmark	OET	ACET	WCET	$\overline{DL_3}$
compress	26879	22381	30506	23.89
cover	15589	14052	18024	24.34
crc	104229	101168	109163	26.9
duff	4904	4789	5281	23.29
edn	472896	466624	553972	22.56
expint	16408	16289	16708	25.5
fac	1208	1128	1240	25.0
fdct	21919	23606	31837	19.54
fibcall	1167	1133	1228	27.87
fir	1924805	1845887	2225970	24.35
jane	1016	941	1108	26.35
jfdcint	32039	30788	37035	21.45
matmul	1571600	1507138	1764383	24.65
minver	157968	142106	191270	24.53
ludcmp	371279	337656	456766	24.95
prime	175238	169281	200651	24.07
quart	224120	204210	270686	24.45
recursion	6846	6782	6898	24.4
ud	41007	36905	48212	23.87

Table 6.1: Execution times in clock cycles under $\alpha = 3$ (N = 4) Interference

anomaly when using $N = 4$, $\alpha = N - 1 = 3$ and cache sizes of 512 Bytes each. The WCET and ACET were calculated by putting parameters, $W_L = 4 \times SS, B_L = SS$ and $A_L = (B_L + W_L)/2$, the technique presented in Sec. 3.1.4.

Results of this test are presented in Table 6.1. For most of the applications in the table, the OET (observed execution time) is more than the ACET. This is an intuitive behavior. However, for the `fdct` application, OET < ACET. This is a counter intuitive behavior.

Note the low $\overline{DL_3}$ parameter of the `fdct` application in the table. The low value of the parameter indicates that the application did a majority of its accesses in the favorable region of Fig. 6.1 under $\alpha = 3$ interference and for the 512 Byte cache configuration.

6.1.6.2 Test2

Similar to the previous test, the goal of this experiment is to explore an evidence of existence of the timing anomaly – 2 (Sec. 6.1.5) in real-life applications. During our intensive test process, we step by step increases the total number of cores to eight ($\alpha = 7$). After adding each core, $\overline{DL_\alpha}$

Benchmark	$\alpha = 4$		$\alpha = 5$		$\alpha = 6$		$\alpha = 7$	
	$\overline{DL_4}$	OET	$\overline{DL_5}$	OET	$\overline{DL_6}$	OET	$\overline{DL_7}$	OET
compress	27.63	30807	34.3	35647	40.44	40999	45.06	44615
cover	28.97	17365	32.32	18749	40.04	21573	48.41	24661
crc	33.78	107677	40.11	110933	46.92	113965	53.78	117333
duff	31.09	5216	37.14	5520	44.36	5840	51.19	6088
edn	31.82	541152	36.77	580008	40.8	607072	49.17	669552
expint	33.23	16696	39.1	16928	47.1	17144	54.56	17384
fac	29.57	1208	35.28	1288	45.57	1384	45.57	1432
fdct	27.54	27398	35.54	32878	43.54	38358	51.54	43838
fibcall	33.87	1214	44.87	1263	51.87	1359	51.87	1407
fir	34.07	2222340	41.33	2463316	41.02	2392804	48.64	2625492
jane	31.5	1096	37.21	1184	45.21	1304	50.92	1392
jfdcint	27.86	34990	35.22	39710	42.05	43902	49.53	48550
matmul	28.39	1654800	33.31	1785104	40.71	1923072	44.66	2094224
minver	31.75	190296	38.39	218552	45.9	250248	53.19	281304
ludcmp	32.3	446648	39.94	523008	47.68	600952	54.87	675096
prime	26.38	181286	32.89	200823	38.69	213191	48.05	238126
quart	33.08	266536	40.2	305960	48.26	351256	55.97	393608
recursion	32.4	6893	37.2	6917	43.6	7037	48.4	7077
ud	30.47	47007	37.46	54487	44.82	62207	51.43	69423

Table 6.2: Execution times in Clock Cycles under varying α interference. $\alpha \in [4, 7], N \in [5, 8]$

was analyzed and OET was measured. Fortunately, we did not have to regenerate computation trace set since the computation trace is independent of number of co-existing masters. The OET increased for the most of the applications when an aggressive co-existing core is added to the system, except, the `fir` application. This application executed faster when α (number of aggressive co-existing masters) was increased from five to six. This is a counter intuitive behavior.

This test asserts that measuring execution time of a test-application in the presence of aggressive co-existing masters and considering the measured execution time as the WCET is unsafe since, less number of aggressive co-existing masters could incur more interference.

Under TDMA arbitration, the rotation of the arbiter pointer is deterministic by specification (Sec. 3.1.6 – fixed slot schedule). Hence, such anomalies can be observed there. See the results presented Fig. 4.8 and Fig. 4.9. In these figures, an additional master in the system reduces the WCET of these particular applications.

This section has presented two counter intuitive timing behavior of applications executing on multi-core architectures. It has proved theoretically that measuring OET of applications in the presence of aggressively interfering masters and considering the OET as WCET is unsafe. The section has also presented evidences of the phenomenon by executing applications from a popular benchmark suit on an FPGA platform.

6.1.6.3 Discussion

This section has presented timing anomalies originating from the interference on the shared memory and built formal and practical arguments around the RR arbiter. However, the phenomenon can also be observed in advanced budget based arbiters, for example - CCSP (Sec. 4.3) and PBS (Sec. 4.2). Under the budget based arbiters, due to the finite budget allocated in a unit time – Replenishment period, an aggressive co-existing masters quickly consume their budgets and become ineligible. Hence, they do not have a potential (budget) left to interfere with the test application which is beneficial to the test-application.

An aggressive co-existing master also goes against our 2^{nd} intuition for the worst case latency analysis (Sec. 4.2.2 and Sec. 4.3.3). There we had assumed, co-existing masters *preserve* (not quickly consume) their budgets to interfere with the test-application.

This section has presented timing anomalies originating from the shared memory interference. The anomalies depend on the shared resource access pattern of test applications. The shared memory access pattern depends on the application code itself and the cache configuration. A trivial bug fix in source code can either introduce or remove the anomalies. Hence, measuring OET of applications in the presence of uninterrupted (aggressive) interference and considering the measured OET as WCET is highly unreliable.

6.2 Unpredictable behavior due to the COTS interconnect specifications

COTS components, if usable, promise a low cost multi-core solution for safety critical HRT applications. Hence, there is a growing interest from industry as well as academia in using COTS multi-core products for safety critical applications. However, low level specifications of underlying hardware is often undisclosed to protect competitive edge and only vague high level details are released, for example, memory response time – 70 ns. It is then a common practice to take this factor in abstract model for WCET analysis or to start measuring execution

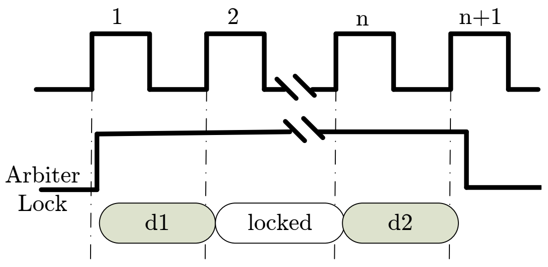


Figure 6.4: Explicit bus monopolizing.

time of applications on the given processor. However, some Cots components are simply not suitable to Wcet analysis. This fact originates from deep inside their hardware specification which is invisible to the analyzers. The goal of this section is to highlight the unsuitability of Cots interconnect specifications and demand Wcet analyzable interconnect specification subset before an industrial integration of unsuitable Cots components is attempted.

6.2.1 Explicit and Implicit bus monopolizing

Popular Cots interconnect specifications, for example AMBA [99], Avalon [85] and PLB [100], specify mechanisms which lets one bus master monopolize the bus forever irrespective of the employed arbitration scheme. The bus can be monopolized explicitly or implicitly.

6.2.1.1 Explicit bus monopolizing

To facilitate transfer of large data chunk and avoid arbitration latencies, a dedicated signal – `ArbiterLock` is provided by the above mentioned interconnect specifications. The signal, as its name suggests, locks the arbitration to the asserting master until the asserting master releases it (Fig. 6.4 [98]). Clearly, this signal must be avoided in multi-Hrt systems to avoid bus monopolizing by any master. In a single-Hrt system, only the critical master can be allowed to use the `ArbiterLock` signal. However, as we investigate, the arbiter can be locked without using the `ArbiterLock` signal.

6.2.1.2 Implicit bus monopolizing

Implicit bus monopolizing is depicted in Fig. 6.5 [98] and it could occur during a burst write transfer. As explained in Sec. 3.1.1, a burst access is issued to the main memory when a cache miss occurs. In case of a write back, the evicted cache line is written back to the main memory in a burst fashion. In a burst write transfer a write signal is raised together with the first word

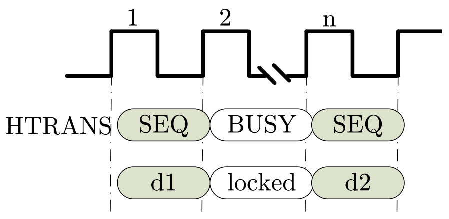


Figure 6.5: Implicit bus monopolizing.

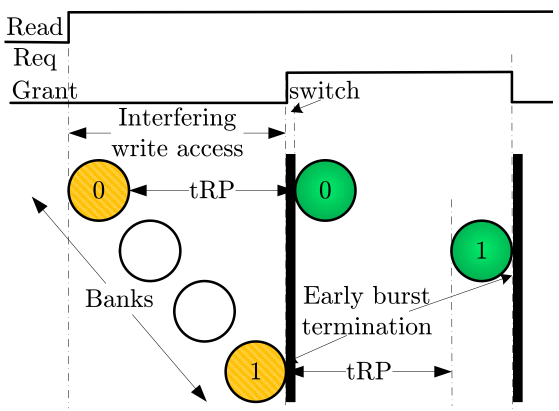


Figure 6.6: `EarlyBurstTermination` and its impact on bank interleaved mapping

to be written. At the same time number of words in the burst (`BurstLength` – cache-line size) is issued. An arbiter now guarantees shared memory to the requesting master until `BurstLength` words are transferred. Generally, when a processor core issues a burst write request, it has all the data available. However, burst transfers can also be used by peripherals. They may issue a burst request before buffering all data to be transferred. In this case, they can issue `BUSY` response, for example, in AMBA-AHB (in Avalon and PLB, `WRITE` can be released to achieve the same effect). The arbiter remains locked until `BurstLength` number of words are transferred in this case. Thus, a malfunctioning peripheral can indefinitely monopolize the shared memory.

Clearly, implicit bus monopolizing must be avoided in real-time capable interconnect specification. A mechanism, called `EarlyBurstTermination`, is provided only in AMBA to avoid implicit bus monopolizing. The monopoly problem is solved by it, however, this mechanism may lead to invalidation of worst case latency parameter as explained in the following subsection.

6.2.2 Impact of Bank interleaved mapping

Bank interleaved mapping is employed extensively in research projects aiming to reduce the effect of bank interference. In the chapter 4 of this thesis also, bank interleaved mapping is

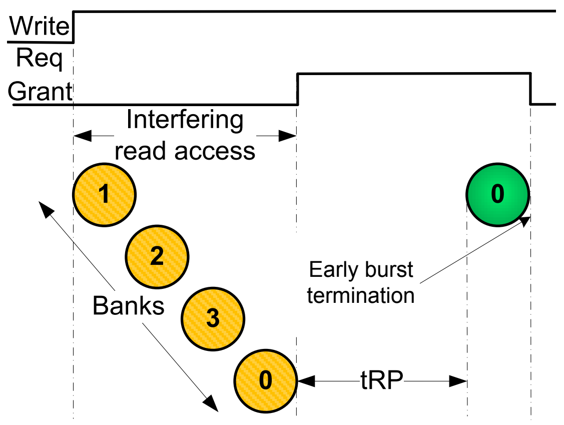


Figure 6.7: Wrap around burst and its impact on bank interleaved mapping

employed. We will see in this subsection that the bank interleaved mapping of cache-lines is not suitable if `EarlyBurstTermination` is employed.

The Fig. 6.6 depicts `tRP` (Activate to Precharge delay) timing requirement which must be satisfied according to SDRAM specification. The circles in the figure represents activate-access-precharge of a bank. The `tRP` requirement enforces delay between the consecutive accesses to the same SDRAM bank.

The figure shows that a read access is being interfered by a write access (the worst case for a shared SDRAM). Let us assume that a write access issues a `BUSY` response for two clock cycles. This results in the write access taking more than its allocated time. Hence, arbiter intervenes and issues `EarlyBurstTermination` signal. Following that, the read access is scheduled in the next slot. However, due to the `tRP` requirement, the read access cannot complete in its predefined number of clock cycles. This results in a partial completion of the read access and prolongation of execution time although it was adhering to the interconnect specification.

6.2.3 Impact of wrap around burst

Wrap around burst is typically employed in instruction cache to increase the average case performance. It is typically issued when a branch target is not aligned to the cache-line boundary. In this case, to get the blocking instruction at first, a burst is issued starting from the word in cache-line which includes the blocking instruction. However, if bank interleaved mapping is used to access the shared SDRAM, scenario presented in Fig. 6.7 may occur. Here, a wrap around burst issued by an interfering access may lead to partial completion of the succeeding sequential access.

Benchmark	Without cache-line alignment	With cache-line alignment	% deviation
compress	693	1162	67.6
cover	339	386	13.8
crc	632	623	-1.4
duff	41	86	109.7
edn	7437	8067	8.4
expint	32	921	2778.1
fac	12	10	-16.6
fdct	685	1115	62.7
fibcall	8	10	25.0
fir	31765	18213	-42.6
jane	14	12	-14.2
jfdcint	574	422	-26.4
matmul	21679	26408	21.8
minver	4000	3660	-8.5
ludcmp	9852	10396	5.5
prime	2613	896	-65.7
quart	5543	5475	-1.2
recursion	10	21	110.0
ud	1021	1049	2.7

Table 6.3: Impact of branch target alignment on the number of cache misses

As explained above, the wrap around burst and bank interleaved mapping for a shared SDRAM cannot be combined. There are two ways to resolve this issue.

- Add a reorder buffer between masters and the shared SDRAM which will buffer the incoming unaligned burst request and issue a sequential burst request. Similarly, the reorder buffer must capture the data reverted by the shared SDRAM and re-arrange it in the original requested order.
- A second approach is employed in Sec. 4.5.1. Here, compiler flags are used to align all branch targets to cache-line size (32 B) boundaries. Hence, only sequential burst requests are issued.

It is clear that the first approach will increase the memory access latencies due to two intermediate buffers. Obviously, the sequential burst accesses can bypass these buffers resulting in unchanged latency. However, unaligned burst accesses will result in higher latencies than if the buffers were not used. This results in poor average case as well as poor worst case performance. Hence, it must be researched if the reduced worst case latencies due to bank interleaving compensates for the increased latencies due to the buffers.

The second approach manipulates computation trace of application. To align branch targets, padding code is inserted. This results in increased executable binary size and modified cache miss pattern. For example, table 6.3 presents deviation in the number of cache misses after building applications using branch target alignment flags. Here, application code (Mälardalen WCET benchmark suit), processor (NIOS II F) and cache configuration is the same (32 B cache-line size, 512 B I$ and D$). The only difference is the usage of alignment flags to avoid wrap around burst. This difference results in a significant difference in number of cache misses. For example, for the `expint` application the number of cache miss is increased by 2778% and for the `prime` application the number of cache miss is decreased by $\tilde{6}6\%$. Clearly, the deviation in number of cache miss impacts WCET and it must be investigated if the bank interleaved mapping of caches results in reduction in the WCET or not.

6.3 Summary

This chapter has presented some precautions a designer and an analyzer of a safety critical multi-core should undertake. Although the safety critical aspects of a multi-core architecture are deeply studied by recent research projects, some untouched caveats are presented in this chapter.

The first part of the chapter presents two *counter intuitive* timing behavior due to the interference on a shared memory in multi-core architectures. It proves theoretically that the aggressive shared memory traffic generated by co-existing masters could be helpful to the test application to execute faster. Additionally, it also proves that some applications experience more interference in the presence of less number of aggressive co-existing masters. The practical evidences of the presence of these *counter intuitive* timing behavior (timing anomalies) are also presented using test applications from a popular benchmark suit executing on a real hardware multi-core processor built on an FPGA.

The second part of the chapter targets the popular interconnect specifications and demands a real-time capable subset of the specification. It shows that certain facilities provided in the current specifications increase average case performance, however, render the architecture unsuitable for safety critical hard real-time systems. It also recommends in depth investigation before adopting bank interleaved mapping, a popular technique to reduce bank interference, for a shared Sdram. The side effects of adopting a bank interleaved mapping may result in poor average case as well as poor worst case performance.

Disregarding the precautions presented in this chapter is also non-compliant to the guidelines presented in CAST-32 paper.

Chapter 7

Demonstrators

The previous chapters have presented various hardware and analysis techniques for predictable and higher performance arbitration schemes in multi-core processors. These chapters provided proof of the concept by executing test applications from a popular benchmark suit on a multi-core architecture built on an FPGA. The chapter 6 cautioned the safety critical multi-core designers and analyzers against subtle *easy-to-overlook* details which render the architecture unsuitable or the analysis invalidated for safety critical applications.

This chapter presents demonstrators which use techniques presented in previous chapters. The demonstrators are close to real-life applications and exhibit the potential industrial exploitation of the techniques. The first demonstrator presents a time and space separated multi-core system. This setup is attractive to mixed critical systems where one application is execution time sensitive while other applications are shared memory bandwidth sensitive. The second demonstrator exhibit the technique presented in the 5^{th} chapter. The last demonstrator uses predictable and high performance multi-core architecture to build centralized multi-core Electronic Control Unit (ECU) for eCars.

7.1 Time and Space Separated Multi-core System - TASERS

The Fig. 7.1 (MPU provided by [104]) depicts a multi-core architecture built on an Altera FPGA board. In total six bus masters are connected to a shared SDRAM (DDR2 in this case). The shared SDRAM is arbitrated under the PBS arbitration scheme. Four out of six masters are NIOS II cores. Two cores are dedicated to generate graphics on an LCD. One core is responsible for controlling the levitating magnet. And one core emulates a malfunctioning core. The remaining two bus masters are graphics and IO DMA engines. The video of the demo can be seen on the

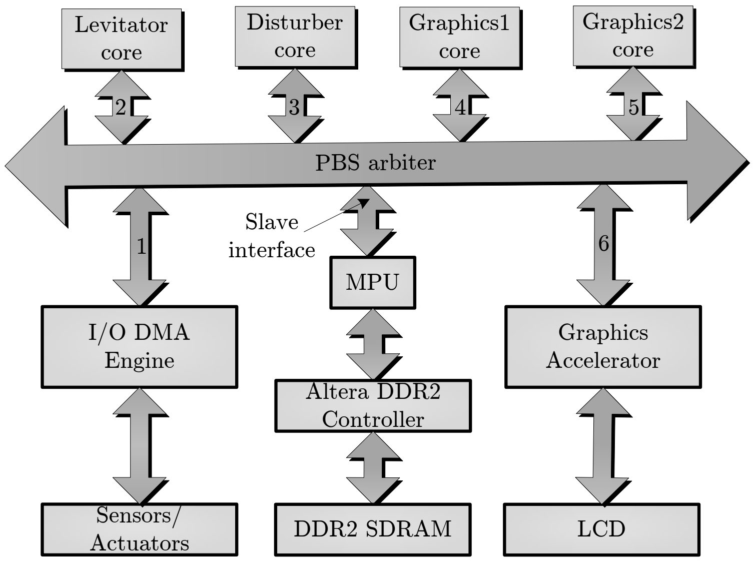


Figure 7.1: TASER Architecture

following link[1]. The demo clearly shows resilience of the mixed critical systems from faults and robust time and space partitioning.

The space partitioning of the demo is contributed by Hattendorf *et al* [105] while the time partitioning and functional code of the levitation and the graphics application was developed under this thesis.

7.1.1 Operation

The demo platform executes three applications: a deadline sensitive hard real-time application, a memory bandwidth sensitive graphics application and a faulty application. The description of the applications are as follows.

7.1.1.1 Deadline sensitive hard real-time application

The Fig. 7.2 [105] depicts the deadline sensitive magnetic levitation application. In this application, a controlling coil generates only sufficient magnetic force to let the permanent magnet floating. The controller is a simple memory-less *proportional* controller with a sampling period of 100 μs. The controller is intentionally kept simple to make it highly vulnerable to a deadline miss. The IO-DMA engine reads the actual position of the magnet and writes it into the shared

[1]`https://www.youtube.com/watch?feature=player_embedded&v=tIkgxyE-Z2s`

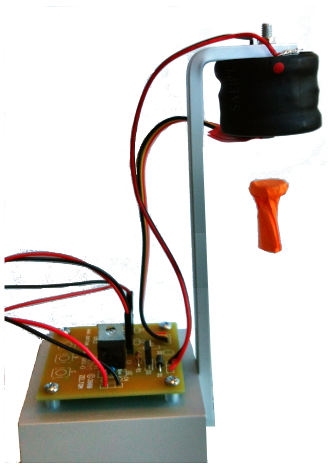


Figure 7.2: Magnetic levitation: A deadline sensitive hard real-time application

SDRAM. The position of the magnet is detected by a combination of hall-effect sensor and an analog to digital converter.

The DMA updates the shared SDRAM every 100 μs with the actual position of the magnet. The levitation controlling core receives an interrupt from a timer every 100 μs. The interrupt service routine then accesses the shared SDRAM, reads the updated magnet position and tunes the current flowing through the coil to control the electromagnetic force. Thus, in a 100 μs period, the DMA engine accesses the shared SDRAM once. In the same time period, the NIOS core which controls the levitation accesses a dummy location at first. Then the location where the actual sensor value is stored is accessed. This is required to inform the cache (self made single line cache[1]) to explicitly read an updated sensor value. Thus, the controller (cache) needs to access the shared SDRAM twice in 100 μs time period.

If either the DMA engine or the levitation controlling core does not get an access to the shared SDRAM in this time frame, the magnet becomes unstable.

7.1.1.2 Bandwidth sensitive graphics application

The graphics application displays two rectangular grids on the LCD. Two individual NIOS cores control each of the grids. The grids are initialized after reset with red color. The NIOS cores synchronize with each other only once after they have initialized their respective grid. The left grid controller rotates two green tokens, 180 degrees apart, on the left grid. Similarly,

[1] A customized cache having only a single line was created to access the shared SDRAM in a burst fashion and avoid undesirable cache effects at same time. Cache analysis is not the focus of this thesis.

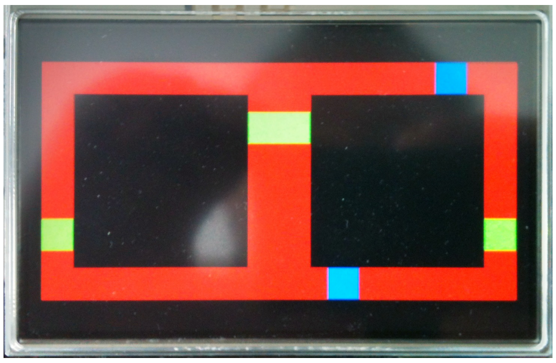


Figure 7.3: Bandwidth sensitive graphics application

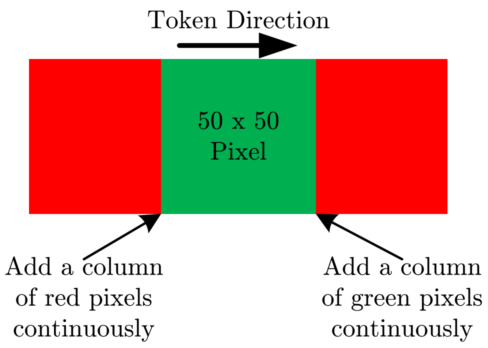


Figure 7.4: Rotation of the tokens

the right grid controller rotates two green and two blue tokens, interleaved 90 degrees apart, on the right grid. The tokens on the left grid rotates clockwise while the tokens on the right grid rotates anticlockwise. The requirement is, green tokens from both the grids must align with each other when they are traversing through the middle of the LCD. The scenario is depicted in Fig. 7.3 [105].

Each pixel on the LCD is represented by an `unsigned int`. The rotation of the tokens is achieved by shifting them by one pixel, either column wise (when tokens are traversing horizontally) or row wise (when tokens are traversing vertically), continuously. The movement of the tokens is depicted in Fig. 7.4. Each token is made of 50 x 50 pixels. The generated video image is stored in the shared SDRAM. The graphics DMA engine continuously reads the prepared video image from the shared SDRAM and throws it on the LCD. The graphics accelerator should be provided very high bandwidth to achieve higher frame rate on the LCD. Thus, the complete graphics application involves two NIOS cores and a graphic accelerator as a shared SDRAM masters.

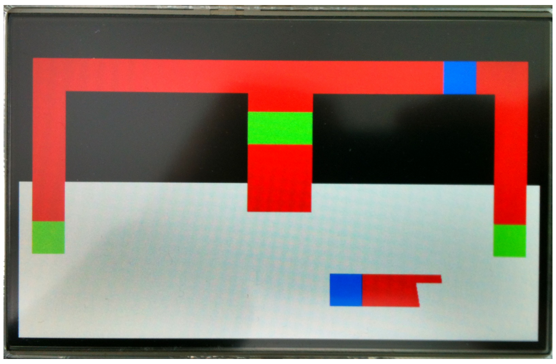


Figure 7.5: Faulty application disrupts the image

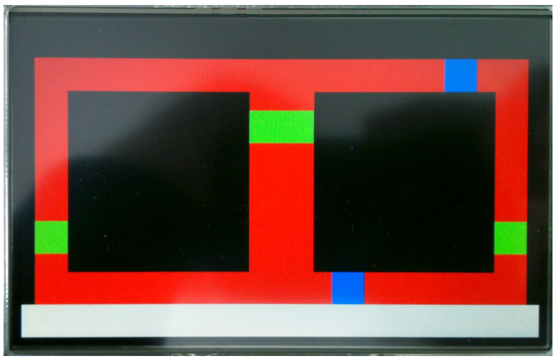


Figure 7.6: Enabling the MPU protects the core part of the grid from the faulty application

As explained earlier, the left grid controller moves only two token while the right grid controller moves four tokens in a unit time period. Hence, the allocated budget to the right grid controller is twice as much as the left grid controller under the PBS arbitration scheme. Since the cores are not explicitly synchronized, if one core gets slightly more or less than the allocated bandwidth then the green tokens will be out of synch in the middle of the LCD.

7.1.1.3 Faulty application

The faulty application emulates stuck-at fault on the `accessRequest` signal. Thus, it continuously demands a shared SDRAM access and overwrites half of the video image with white pixels (see Fig. 7.5 [105]). Here, the PBS arbiter must control the amount of bandwidth assigned to the faulty application and keep co-existing applications (levitation, graphics) unaffected. Otherwise, the magnet falls down and the tokens go out of synch. The faulty application can be enabled by a push of a button.

7.1.1.4 The shared Memory Protection Unity - MPU

We used the MPU from Hattendorf *et al* [105]. It is similar to one presented in Fig. 5.7 and protects memory regions of the shared SDRAM. When the MPU is enabled, the faulty application can only overwrite a part of the LCD. The grids and tokens are protected as depicted in Fig. 7.6 [105]. Thus, the MPU provides space partitioning.

7.1.2 Assignment of Priorities and Budgets

Allocation of the priorities and budgets are very crucial for such mixed critical applications. We began with two parameters. i) The levitation control application has a period of 100 μs and it is a HRT. ii) The graphics accelerator needs the highest shared SDRAM bandwidth to produce high frame rate on the LCD.

We fixed the replenishment period at 100 μs. At 125 MHz operating frequency, the replenishment period is 12500 clock cycles long. One refresh is issued at every `tREFI` = 7.8 μs and it takes `tRFC` = 41 clock cycles to serve the refresh operation. Hence, in a single replenishment period, there are $\approx$ 13 refreshes. They occupy 13 x 41 = 533 clock cycles. Considering the worst case of read/write switching accesses to the shared SDRAM, one access can be served every 13 clock cycles. Thus, the total available budget in 100 μs period is (12500 - 533)/13 = $\approx$ 920.

The DMA-IO engine accesses the shared SDRAM once while the levitation controller core accesses the shared SDRAM twice in 100 μs. Additionally, they belong to the most latency sensitive application. Hence, the DMA-IO receives highest priority (shown at the interface points in Fig. 7.1) and a budget of 1 while the levitation controller receives the second highest priority and budget of 2 in 100 μs replenishment period. The left grid controller is allocated budget of 100 and the right grid controller is allocated budget of 200 in 100 μs replenishment period with priorities 4^{th} and 5^{th}, respectively. The faulty application is allocated, equal to the right grid controller, budget of 200, however, priority 3^{rd}. The higher priority of the faulty application than the graphics application makes graphics application vulnerable to the faults. Finally, the remaining budget of 417 is allocated to the graphics accelerator and 6^{th} priority. Thus, the graphics application can read 417 x 8 (burst of 8 integers per access) = 3336 pixels in 100 μs. This results in 3360e4 pixel reads in a second. However, the interface between the FPGA board and the LCD is 8 bits wide. Hence, 24 bit pixel needs 3 clock cycles to be pushed on the LCD. This results in 3360e4/3 = 1120e4 pixels thrown at the LCD per second. At the resolution of 800 x 480, this results in $\approx$ 29 fps. The high frame rate creates crisp images on the LCD as can be seen in the video.

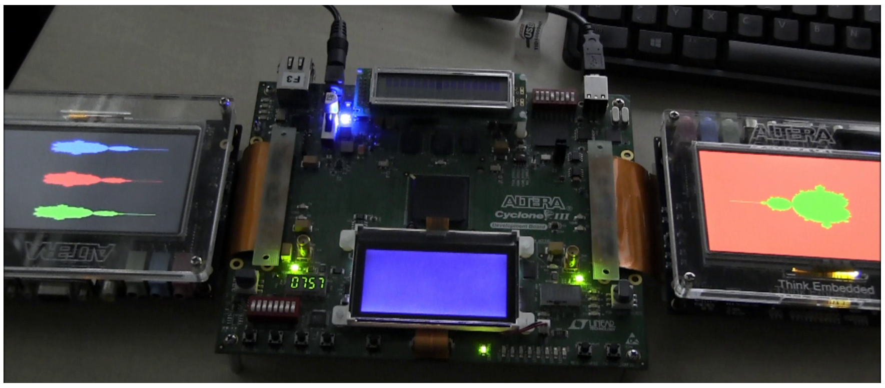


Figure 7.7: Demo setup

7.1.3 Comments on certification

The demo analyzes each applications according to their latency and memory requirements. The worst case SDRAM bandwidth (shared memory bandwidth) is also analyzed and allocated to the masters to fulfill their duties. This approach is conceptually certifiable based on the guidelines of the CAST32 [3] paper. For more details, please refer Sec. 4.6.

7.2 Measurement based WCET analysis for multi-core systems

This section demonstrates the technique presented in Chapter 5. It employs the optimized cache observer (Sec. 5.2.2) and uses computation and memory intensive Mandelbrot set generation. The C code for the Mandelbrot set generation was taken from Rosettacode [106] and ported to the Altera environment. Typically, the code was adopted to generate continuous video frames on the LCD. Video of the demonstrator can be viewed on the following link[1].

7.2.1 Demo setup

The setup of the demonstrator is pictured in Fig. 7.7. The Fig. 7.8 represents graphical view of the demonstrator. A quad-core processor is built using NIOS II F cores. Each core has 4KB of instruction and data caches. All cores produce a Mandelbrot set and display it on the connected LCDs as shown in Fig. 7.7. The interfering cores share an LCD while a dedicated LCD is connected to the test core. Additionally, the optimized cache observer is connected to

[1]`https://www.youtube.com/watch?feature=player_embedded&v=31BmaIhK1Ww`

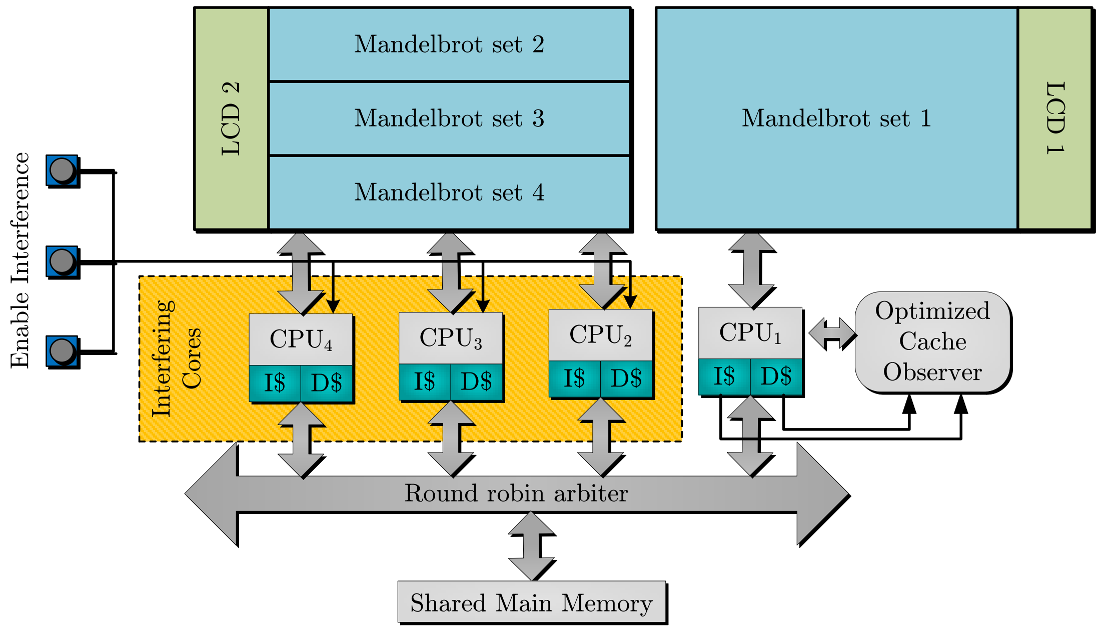


Figure 7.8: Graphical representation of the demo setup

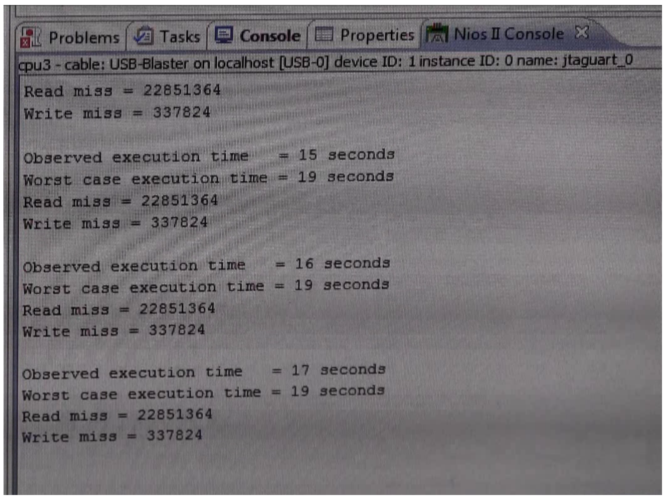
the test core. An on-chip memory serves as a shared main memory to all cores. The interfering cores can be enabled/disabled by push button switches.

7.2.2 Operation

The optimized cache observer, connected to the test core, adds worst case latency for each cache miss considering the worst possible interference. Moreover, it also measures the observed execution time of Mandelbrot set generation on the test-core. The Fig. 7.9 depicts different observed execution times while the WCET stays constant. The observed execution time is dependent on the interference and varies between 14 and 17 seconds while the WCET is calculated based on the worst possible interference, hence, it has a constant value of 19 seconds. The C code for the Mandelbrot set generation is a single path code, hence, measuring start to end time considering the worst interference is sufficient. Obviously, a soft-reset is issued and the LCD is cleared before a new set is created.

7.2.3 Comments on certification

The timing aspect of the demo is certifiable according to the CAST32 [3] guidelines. For more details, readers are referred to Sec. 5.5.



```
cpu3 - cable: USB-Blaster on localhost [USB-0] device ID: 1 instance ID: 0 name: jtaguart_0
Read miss = 22851364
Write miss = 337824

Observed execution time   = 15 seconds
Worst case execution time = 19 seconds
Read miss = 22851364
Write miss = 337824

Observed execution time   = 16 seconds
Worst case execution time = 19 seconds
Read miss = 22851364
Write miss = 337824

Observed execution time   = 17 seconds
Worst case execution time = 19 seconds
Read miss = 22851364
Write miss = 337824
```

Figure 7.9: Measurement based WCET analysis fo multi-core architectures

7.3 A predictable and high performance multi-core processor as a centralized eCar controller

The challenge of the ICT architecture and performance scaling in modern automobiles has been described in Sec. 1.1. This section presents a model car as a demonstrator which shows how our techniques provide solution to the challenge. As suggested by researchers [107], we employed Ethernet as the communication backbone of the car.

In total, two complete model cars were developed in this thesis work. The hardware architecture was designed under the thesis. The model cars were handed over to student participants of the LEGO lab course[1] at Technische Universität München. Under supervision, students developed the software applications executing on the ECUs. The following subsections describe the architecture of peripheral ECUs and the central *predictable and high performance* quad-core ECU.

7.3.1 Demo setup

Like a real car, our model car consists many components built/developed separately and integrated in one unit. However, unlike real cars, an addition of an ECU consists of the following simple steps: i) Connect the ECU to the Ethernet backbone with a dedicated IP address and ii) Add a software thread in the central ECU to manage its functionality. These simple integra-

[1]http://www6.in.tum.de/Main/TeachingWs2014HSCDLegoCar

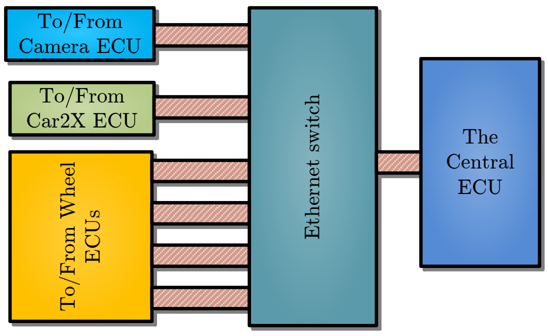


Figure 7.10: Ethernet communication architecture

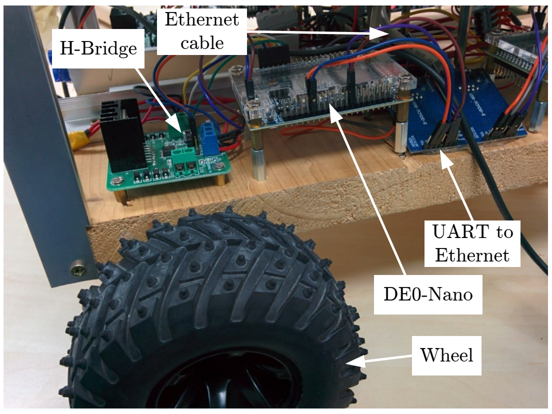


Figure 7.11: The wheel ECU

tion steps ease scaling, debugging and verification processes. The following subsections detail different ECU components and the communication architecture.

7.3.1.1 Communication architecture

The Fig. 7.10 depicts Ethernet communication architecture built in the model cars. We chose star topology in our cars due to its simplicity and availability of COTS Ethernet switch. Our communication architecture is not time predictable due to the usage of the COTS components, e.g. UART-2-Ethernet converter, Ethernet switch etc. We chose COTS components due to the cost restrictions and our main focus is to provide a proof of concept. In real cars, components operating on time-triggered Ethernet protocol must be employed. EtherCat [9], PowerLink [10] etc. are the examples of such predictable Ethernet communication protocols.

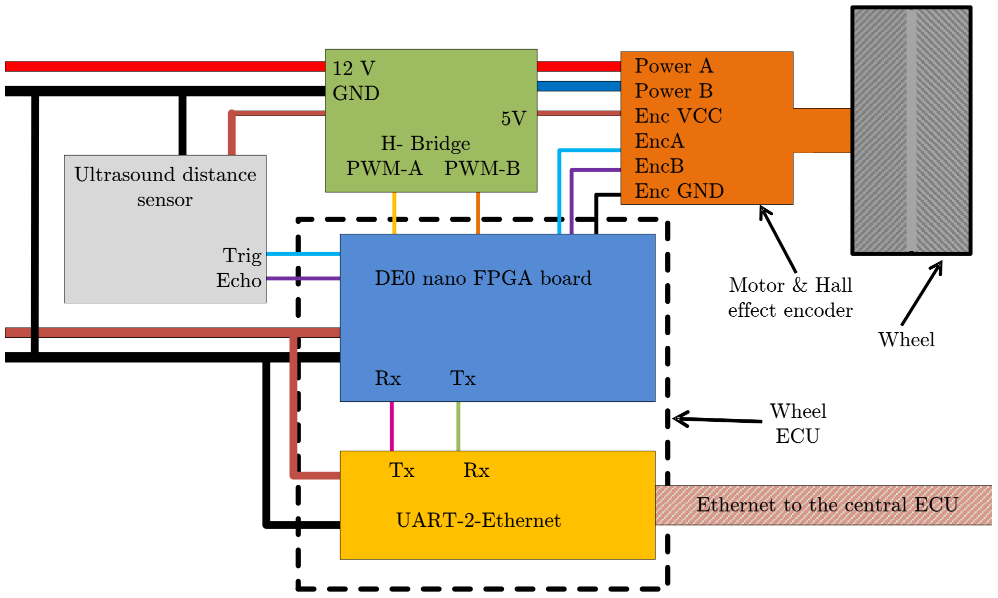


Figure 7.12: Graphical representation of the wheel ECU

7.3.1.2 The wheel ECU

The Fig. 7.11 depicts the wheel ECU as implemented on model cars. The functional/graphical description of the ECU is depicted in Fig. 7.12. Each wheel of the cars has a dedicated ECU. The ECU generates the PWM signals to set the speed and direction of the motor/wheel rotation. The PWM signal is generated by a dedicated hardware IP block for higher accuracy and efficient operation. Similarly, dedicated hardware IPs are used for reading the actual speed through Hall effect encoders and for measuring distance from an obstacle through ultrasound distance measurement device (Fig. 7.12).

The wheel ECUs are implemented on Altera DE0-nano boards due to their low cost and high usability. The boards are connected to the Ethernet via UART-2-Ethernet converters. The on-board NIOS core receives the desired rotation speed from the central ECU via Ethernet. It implements a PID controller to maintain the desired speed. The NIOS core also packs the sensor related data, for example actual speed, distance from an obstacle etc, in a frame and sends it to the central ECU via Ethernet.

The central ECU can set individual speed/direction for each wheel through Ethernet commands. The forward and backward motions of the cars are achieved by driving all wheels in the same direction. Rotation of the car on place can be achieved by rotating wheels on both sides

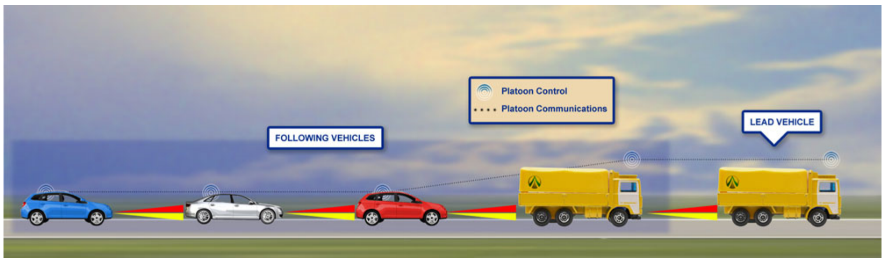


Figure 7.13: Car train – Platooning

in opposite directions. Similarly, turning is achieved by driving wheels on the sides at different speeds.

7.3.1.3 The Car2X ECU

A car train (Fig. 7.13[1]), also called platooning, can reduce fuel consumption, increase road usage and reduce fatalities in accident situations [108]. Here, cars driving to the same destination form a virtual train. Within the train, cars drive very close to each other to reduce air drag and thereby reduce fuel consumption. However, an emergency breaking by a lead car may cause a pile-up. Hence, continuous automated communication among cars must be present. Here, an emergency message is sent to following cars in case of a sudden breaking. The following cars then breaks automatically without any intervention from their drivers to react quickly. To enable this functionality a Car2X ECU is added as a peripheral ECU in our model cars.

In our model cars, the Car2X ECU is built using a WLAN-2-Ethernet converter. The car provides a wireless access point and other car(s) can connect to the central ECU via the access point.

7.3.1.4 The camera ECU

The Fig. 7.14 presents a graphical view of the camera ECU. As shown in the figure, the camera ECU is composed of a webcam and a BeagleBone Black. The main goal of the ECU is to detect a marker attached to the back of the leading car and keep following the marker. Here, again a PID controller is used to keep the marker in the middle of the webcam image. The angular correction in terms of wheels' speed is sent to the central ECU. The central ECU then sends the updated speed to the respective wheel ECUs.

[1]Source: www.driverlesstransportation.com/wp-content/uploads/2014/03/dt-platooning-1000x2881.jpg.

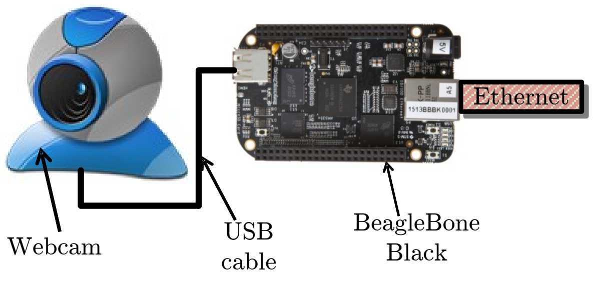


Figure 7.14: Graphical representation of the camera ECU

We chose a very simple marker due to the limited processing power of BeagleBone Black. OpenCV [109] library is used under Ubuntu Linux executing on BeagleBone Black for image processing algorithms. Communication between OpenCV nodes is implemented using Robot Operating System (ROS) [110].

7.3.1.5 The central ECU

The central ECU consists of, as depicted in Fig. 7.15, a quad-core NIOS II processor. The cores are connected to a shared SRAM using the PD arbiter. The shared SRAM is partitioned in such a way that each core has its own exclusive partition in the memory. Additionally, there is a shared partition between the *control core* and the *communication core*. Accesses to the shared memory partition is resolved via a hardware mutex core (not shown in the figure).

The interfering cores, randomly generate traffic towards their shared partition. Here, we would like to focus on the robustness of our time predictable technique. Hence, hardware enabled spatial partition is avoided. The hardware enabled spatial partitioning is already covered in the demonstrator presented in Sec. 7.1.1.

The control and communication cores implement the functionality of the central ECU. As their names suggest, the communication core is connected to the Ethernet for communicating with the *outer world* and the control core is responsible for controlling the speed and direction of the wheels. The *car state* data structure is maintained in the shared partition.

The car state data structure has an *incoming* and *outgoing* sections. The communication core receives incoming Ethernet packets and puts them in the incoming section of the data structure. The control core processes information available in the incoming section and updates the necessary information in the outgoing section. This information is later packed in Ethernet packets and sent to the corresponding ECUs by the communication core. The Nichestack

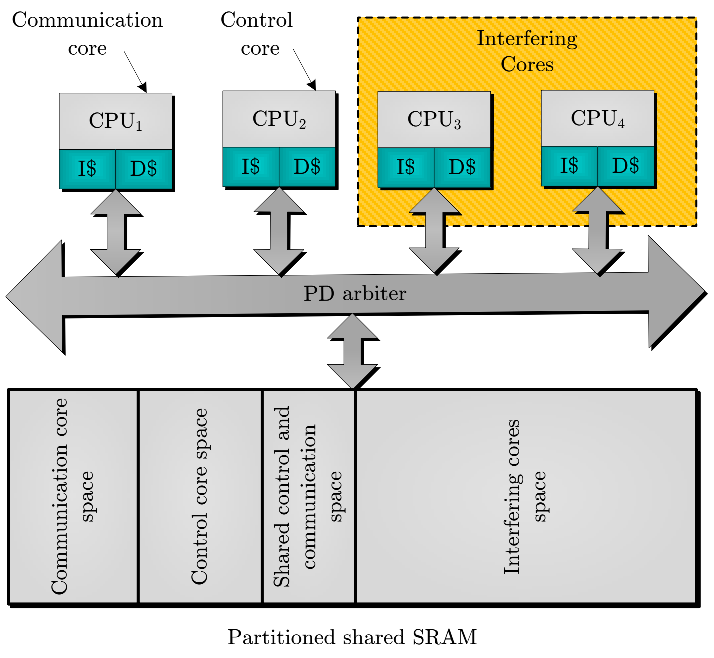


Figure 7.15: Graphical representation of the central ECU

TCP/IP stack was used for Ethernet communication. Threads are implemented using Micro C real-time operating system.

Typical information available in the incoming section is the following. i) From wheel ECUs: actual speed of wheels and distance from an obstacle. ii) From the camera ECU: desired speed of each wheel. iii) From the Car2X ECU: emergency break. In the real world, Car2X ECU may also send additional information such as actual speed of the leading car, destination, rout information etc.

7.3.2 Operation

The communication core executes the TCP/IP stack on the μC OS II. It is configured in server mode. After power cycle, the peripheral wheel ECUs keep on sending a *welcome messages* until they receive an acknowledgment by the central ECU. The wheel ECUs are configured in client mode. The binding process ensures functional Ethernet communication infrastructure between the wheel ECUs and the central ECU.

The camera ECU and the Car2X ECUs are by default (hard coded) bound with the central ECU. The camera ECU detects the marker position of the preceding car and sends wheel velocities such that the marker stays in the middle of the camera frame. This ensures alignment with the preceding car. The car does not have a steering motor, hence, turn is achieved by only difference in wheel velocities. The central ECU receives the wheel velocities and relays them to the wheel ECUs via Ethernet communication. The Car2X ECU listens to an emergency break message sent by the preceding car on Wifi (IEEE 802.11) and forwards it to the central ECU. Upon receiving the emergency break message, the central ECU forwards a stop message to all wheel ECUs.

Although the messages are received and transmitted by the communication core, the information within is processed by the control core. It is control core which sets correct wheel velocities for wheels based on the messages sent by the Car2X and camera ECUs. Both control core and communication core share memory which is aggressively accessed by the interfering cores (Fig. 7.15). However, the PD arbiter (Sec. 3.2) ensures the minimum bandwidth allocated (25 %) to both the cores with efficient shared memory utilization for increased performance.

7.4 Summary

This chapter has presented three demonstrators for the techniques presented in the previous chapters of the thesis. The demonstrators highlight how our techniques can be exploited in real-world applications. The first demonstrator highlight the application of the analysis techniques presented in chapter 4. The system in the demonstrator is the typical case of mixed critical system where applications with different latency and bandwidth requirements co-exist. The second demonstrator is an application of the technique presented in 5^{th} chapter. Here, the WCET of computationally intensive application executing on a multi-core architecture is measured. The technique is certifiable and does not need any change in COTS multi-core chips or timing analysis tools. The third demonstrator is the application of the predicable and high performance multi-core architecture as a centralized ECU of model cars.

Chapter 8

Conclusion and future work

This thesis presents a few novel techniques to achieve predictable execution time of applications executing on multi-core processors with minimal or no compromise on performance. This stands out from the state-of-art where predictability is achieved after significantly sacrificing performance. The thesis approaches the problem in multiple ways using as many COTS components as possible to reduce the adoption costs and time-to-market. Due to the small market share of HRT systems, such approach is highly cost effective.

The thesis focuses on shared memory interference since it is the key differentiating factor on performance and predictability while migrating from single-core to multi-core. Multi-cores employ shared memory to reduce over all cost of the product and to share data. Thus, the shared memory bandwidth is a critical resource impacting predictability and performance of a multi-core system.

The shared memory arbiter design and the worst case latency analysis, both, affect performance and predictability of a multi-core system. This thesis contributes by providing novel techniques in both the dimensions. A new arbitration scheme, Priority Division (PD) is presented. The PD is proven theoretically and through experiments to have equal worst case latency as TDMA in a multi-HRT multi-core systems. Although the TDMA is considered as the best candidate when time predictable execution is required, it inefficiently utilizes the shared memory resulting in degraded performance. The PD arbiter, having equal worst case latencies, increases the memory utilization by applying a secondary – priority based arbitration. For a single-HRT system (mixed critical), static priority (SP– aka fixed priority) based scheduling can be considered as the best candidate. Here, the highest priority is assigned to the critical master for faster execution of applications executing on it. The thesis proposes PD^{h1} mode which results in even faster execution than the SP for the highest priority master in the worst case.

Thus, the PD arbiter is equal to TDMA in the worst case, however, results in much faster execution than TDMA in the average case. This makes it highly attractive to performance demanding safety hard real-time systems.

To satisfy different latency and bandwidth requirements, budget based arbiters are proposed in literature. These arbiters are complex to analyze for the worst case latency since the current latency depends on activity of co-existing masters as well as the past activity of the test-master. To reduce analysis complexity, abstract models, e.g. $\mathcal{LR}$, are proposed. This thesis presents detailed worst case latency analysis in the presence of shared SDRAM. The detailed analysis produces much precise worst case latency analysis which heavily reduces the estimated WCET. Thus, precision in analysis results in better predictability (low WCET) under high performance arbiters.

In order to further ease the WCET analysis and bridge the gap between chip manufacturers and WCET analyzers, the thesis presents measurement based WCET analysis technique using internal monitoring. The technique empowers existing single-core WCET measurement tools to measure WCET on multi-core architectures without any modifications. Additionally, the technique does not interfere with normal operation of the chip, hence, the performance, energy consumption etc. benefits are preserved. Another major advantage of this technique is, it brings chip manufacturers and WCET analyzers together. Chip vendors are reluctant to reveal details of their architecture due to their competitive advantage. Without detailed specification of underlying architecture, the WCET analyzers cannot perform reliable (and usable) analysis. Using the technique presented in this thesis, WCET measurements on multi-core architectures is enabled without revealing the underlying arbiter specifications.

Building predictable multi-core architecture involves deep knowledge of multi-core architecture, system integration as well as analysis techniques. Typically, people with different expertise work in their individual domains. Hence, techniques employed in one domain may invalidate assumptions/techniques employed in other domain(s). This thesis brings knowledge of different domains together and explains how general assumptions made in one domain may interfere with assumptions made in another domain. In the process, the thesis highlights caveats in building and analyzing predictable multi-core systems.

All the techniques in this thesis are supported by related experiments conducted on multi-core architecture built on an FPGA and test applications chosen from a popular benchmark suit.

Although the experiments provide proof of concept of the techniques, to exhibit the usability of the techniques in the real world, *three* demonstrators are presented. Each demonstrator highlights industry deployability and ease of use of the techniques.

Apart from the cost effective integration, our techniques are conceptually certifiable for avionics applications according to the guidelines presented in CAST32 [3] paper. Currently, these guidelines are only published for dual-core processors, however, we expect minor change in the guideline for multi-core processors. Each technique presented in the thesis is accompanied by compliance arguments of the CAST32 paper.

Future work: A framework for Design Space Exploration (DSE) incorporating our techniques could be an interesting future work. The DSE can be applied to the priority arrangement of the PD arbiter. Here, Hardware-In-Loop (HIL) can be developed and priorities within the slots can be re-programmed until the optimization goals are reached. Here, optimization goal could be reduction in WCET on the critical master and faster average case execution on other masters.

The internal monitoring technique for WCET measurements provides OET and WCET simultaneously in real-time. This can be used for OET *health check*. Here, during the application test runs, if the OET is often close to WCET, the test application is heavily interfered by the co-existing applications. A minor structural code modification in either test application or the co-existing applications can significantly change the interference scenario. The modification can be applied to achieve reduction in OET (faster average case execution) of the test application.

An optimized L2 cache partitioning is an ongoing research work. The internal monitoring technique presented in this thesis can be used to build HIL for testing partitioning schemes on a real hardware.

Appendix A

Appendix

A.1 Certification of avionics software executing on a multi-core platform

This section contains multi-core timing related text of CAST-32 paper [3].

Multi-cores are not commonly used in airplanes due to serious challenges to the deterministic software behavior. Hence, industrial experience and certification guidelines are not available. Due to the growing interest of using Multi-core Processors (MCP) in avionics, Federal Aviation Agency (FAA) has published (position paper CAST-32 [3]) objectives and suggested activities for the demonstration of compliance to safety standards. Although the publication is not the official certification policy, it is currently the best available guidance on how to certify multi-cores for avionics.

The CAST-32 position paper lists various rationale (concerns) and objectives in categories of determinism, software and error handling. From the available rationale and objectives, we concentrate on topics related to shared resource interference since it is the main focus of this thesis. Here, for readability reasons, we list objectives related to this thesis mentioned in the CAST-32 [3] paper.

MCP_Determinism_7: The applicant has conducted a functional interference analysis to identify all the interference channels between the software hosted on the cores of the MCP and has designed, implemented and verified a means of mitigation for each of those interference channels.

MCP_Determinism_8: The applicant has stated in their software plans whether or not they intend to use shared memory (between the processing cores) and if they do, has described in those plans the means they intend to use to control access to shared memory locations and to prevent the disruptions to deterministic software execution caused by problems such as race conditions, data starvation, deadlocks or live-locks.

MCP_Determinism_9: If the applicant uses shared memory between the processing cores, the applicant has tested the means that they have designed to control the access to shared memory and has ensured that the implemented means provides uninterrupted access to the shared memory locations from either core of the MCP and prevents either core being locked out from accessing the shared memory.

MCP_Determinism_10: The applicant has stated in their software plans whether or not they intend to use shared cache between the processing cores, and if they do, has also described in their plans their strategy for managing and verifying cache usage.

MCP_Determinism_11: If the applicant uses shared cache between the processing cores, the applicant has conducted analyses and tests to determine the worst-case effects that the use of shared cache and memory can have on the execution of the specific software applications hosted on the two cores of the MCP, has described those effects to the certification authority, and has implemented and verified a means to mitigate the effects of using shared cache.

MCP_Determinism_12: The applicant has described in their software/AEH plans or other deliverable documents how they intend to allocate, manage and measure the use of resources and the use of the interconnect by the applications hosted on the MCP and by other MCP peripherals so as to avoid contention for MCP resources and to prevent the capacity of the interconnect and the resources of the MCP from being exceeded.

MCP_Determinism_13: The applicant has allocated the usage of the MCP resources to the software applications hosted on the MCP and has verified that the total of the resource demands when all applications are executing in the worst-case situation does not exceed the total of the resources available.

MCP_Determinism_14: The applicant has determined the maximum capacity of any interconnect mechanism of their MCP to sustain transactions in a deterministic manner and has verified that the demands made on that mechanism by the software hosted on the MCP or by any peripherals of the MCP do not exceed its maximum capacity during any phase of operation of the system.

The CAST-32 paper also lists suggested activities to demonstrate the compliance to the certification authority. The following are the activities suggested by the CAST-32 paper related to above mentioned objectives. However, an applicant may also propose other activities for the review for acceptability.

4. **MCP Interference Channels.** The applicant should -
   (a) conduct an interference analysis in order to identify all the channels by which the software programs executing on the two cores could interfere with each other via the internal mechanisms of the MCP or through any of the software hosted in it, including the kinds of features of MCPs described in this paper that are not present in single core processors.
   (b) for each of the interference channels identified, design a means to deactivate the interference channel or to mitigate the effects of the interference between the software hosted on the separate cores. This interference analysis should be available for review during audits.

5. **Shared Memory and Cache** The applicant should -
   (a) develop and implement means to control access to the shared memory areas by the software hosted on the cores of an MCP. The resulting implementation should prevent situations in which an access to the shared memory by the software hosted on one core does not cause disruption to the execution of the software hosted by another core due to such problems as race conditions, data starvation, deadlocks, or live-locks.
   (b) analyze the means that are used to communicate between the cores via shared memory. This supporting analysis should be documented. The resulting implementation should be documented in the applicable software and AEH requirements and design data.
   (c) describe in their software plans their strategy for managing cache memory and state whether or not they intend to use shared cache.
   (d) if shared cache is going to be used, state in the software plans how they intend to mitigate any interference this may cause between the applications executing on the two cores of the MCP.
   (e) if shared cache is used, conduct analyses and tests to determine for the software hosted on each core the worst extent of the effects due to the use of shared cache in

terms of aspects such as data corruption, scheduling and the WCET due to accesses to cache from the other core. The applicant should allow for these effects in the allocation of processing time to the processes hosted on the two cores.

(f) document these analyses and tests and their results for the certification authority review and provide preliminary results as part of the justification that a proposed MCP installation is feasible and can be fully verified.

Throughout the thesis, we provide comments on how our techniques relate to the above mentioned objectives and activities.

References

[1] Hardik Shah, Kai Huang, and Alois Knoll. **The Priority Division Arbiter for low WCET and high Resource Utilization in Multi-core Architectures**. In *RTNS 2014, Versailles, France*. xiii, 10, 21, 24, 27, 29, 30, 33, 34, 35, 36

[2] Hardik Shah, Andrew Coombes, Andreas Raabe, Kai Huang, and Alois Knoll. **Measurement based WCET Analysis for Multi-core Architectures**. In *RTNS 2014, Versailles, France*. xiii, 15, 67, 68, 70, 71, 73, 77, 78, 79

[3] **Certification Authorities Software Team. Position paper CAST-32, Multi-core Processors.** `www.faa.gov/aircraft/air_cert/design_approvals/air_software/cast/cast_papers/media/cast-32.pdf`. Accessed: 2015-05-17. 1, 19, 107, 108, 119, 121

[4] KW Tindell, Hans Hansson, and Andy J Wellings. **Analysing real-time communications: controller area network (CAN)**. In *Real-Time Systems Symposium, 1994., Proceedings.*, pages 259–263. IEEE, 1994. 2

[5] Karen Parnell. **LIN Bus–A Cost-Effective Alternative to CAN**. *Electronic Engineering & Product World*, **8**:97–99, 2005. 2

[6] Robert Tappe, Christian Thiel, and R Konig. **MOST Media Oriented Systems Transport**. *Elektronik. July*, 2000. 2

[7] Traian Pop, Paul Pop, Petru Eles, Zebo Peng, and Alexandru Andrei. **Timing analysis of the FlexRay communication protocol**. *Real-time systems*, **39**(1-3):205–235, 2008. 2

[8] Christian Buckl, Alexander Camek, Gerd Kainz, Carsten Simon, Ljubo Mercep, Haucke Stähle, and Alois Knoll. **The software car: Building ICT architectures for future electric vehicles**. In *Electric Vehicle Conference (IEVC), 2012 IEEE International*, pages 1–8, March 2012. 2, 3

[9] Dirk Jansen and Holger Buttner. **Real-time Ethernet: the EtherCAT solution**. *Computing and Control Engineering*, **15**(1):16–21, 2004. 3, 110

[10] Gianluca Cena, Lucia Seno, Adriano Valenzano, and Stefano Vitturi. **Performance analysis of Ethernet Powerlink networks for distributed control and automation systems**. *Computer Standards & Interfaces*, **31**(3):566–572, 2009. 3, 110

[11] Hermann Kopetz, Astrit Ademaj, Petr Grillinger, and Klaus Steinhammer. **The time-triggered Ethernet (TTE) design**. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, pages 22–33, May 2005. 3

[12] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2012. 3

[13] Chuck Moore and Pat Conway. **General-purpose multi-core processors**. In *Multicore Processors and Systems*, pages 173–203. Springer, 2009. 4

[14] Isaac Liu. *Precision Timed Machines*. PhD thesis, EECS Department, University of California, Berkeley, May 2012. 4, 16

[15] Theo Ungerer, Francisco Cazorla, Hugues Casse, Sascha Uhrig, Irakli Guliashvili, Michael Houston, and Floria Kluge *et al.* **Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability**. *Micro, IEEE*, **30**(5):66–75, Sept 2010. 4, 16

[16] Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, Ashkan Beyranvand Nejad, et al. **Virtual execution platforms for mixed-time-criticality systems: the CompSOC architecture and design flow**. *ACM SIGBED Review*, **10**(3):23–34, 2013. 4, 16

[17] Christian El Salloum, Martin Elshuber, Oliver Höftberger, Haris Isakovic, and Armin Wasicek. **The ACROSS MPSoC – A New Generation of Multi-core Processors Designed for Safety-Critical Embedded Systems**. In *Digital System Design (DSD), 2012 15th Euromicro Conference on*, pages 105–113, Sept 2012. 4, 16

[18] Paul Pop, Leonidas Tsiopoulos, Sebastian Voss, Christoph Ficek Oscar Slotosch, Ulrik Nyman, and Alejandra Ruiz Lopez. **Methods and tools for reducing certification costs of mixed-criticality applications on multi-core platforms: the RECOMP approach**. In *WICERT 2013 proceedings*, 2013. 4, 16

[19] Jens Sparso. **Design of Networks-on-Chip for Real-Time Multi-processor Systems-on-Chip**. *2010 10th International Conference on Application of Concurrency to System Design*, **0**:1–5, 2012. 4, 16

[20] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. **The Worst-case Execution-time Problem&Mdash;Overview of Methods and Survey of Tools**. *ACM Trans. Embed. Comput. Syst.*, **7**(3):36:1–36:53, May 2008. 5, 9

[21] Lothar Thiele and Reinhard Wilhelm. **Design for timing predictability**. *Real-Time Systems*, **28**(2-3):157–177, 2004. 5

[22] Raimund Kirner and Peter Puschner. **Time-predictable computing**. SEUS'10, Berlin, Heidelberg, 2010. Springer-Verlag. 5

[23] Martin Schoeberl. **Is time predictability quantifiable?** In *Embedded Computer Systems (SAMOS), 2012 International Conference on*, pages 333–338, July 2012. 5

[24] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. **Memory Access Control in Multiprocessor for Real-Time Systems with Mixed Criticality**. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 299–308, July 2012. 9

[25] Thomas D Richardson, Chrysostomos Nicopoulos, Dongkook Park, Vijaykrishnan Narayanan, Yuan Xie, Chita Das, and Vijay Degalahal. **A hybrid SoC interconnect with dynamic TDMA-based transaction-less buses and on-chip**

networks. In *VLSI Design, 2006. Held jointly with 5th International Conference on Embedded Systems and Design., 19th International Conference on*, pages 8 pp.–, Jan 2006. 10

[26] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. **Hardware support for WCET analysis of hard real-time multicore systems**. *SIGARCH Comput. Archit. News*, **37**:57–68, June 2009. 10, 16

[27] Francesco Poletti, Davide Bertozzi, Luca Benini, and Alessandro Bogliolo. **Performance analysis of arbitration policies for SoC communication architectures**. *Design Automation for Embedded Systems*, **8**(2-3):189–210, 2003. 10

[28] Hardik Shah, Andreas Raabe, and Alois Knoll. **Priority division: A high-speed shared-memory bus arbitration with bounded latency**. In *Proc. DATE*, 2011. 10, 12, 30

[29] Kanishka Lahiri, Anand Raghunathan, and Ganesh Lakshminarayana. **The LOTTERYBUS on-chip communication architecture**. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, **14**(6):596 –608, june 2006. 11

[30] Javier Jalle, Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. **Bus Designs for Time-probabilistic Multicore Processors**. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, pages 50:1–50:6, 3001 Leuven, Belgium, Belgium, 2014. European Design and Automation Association. 11, 17

[31] Chien-Hua Chen, Geeng-Wei Lee, Juinn-Dar Huang, and Jing-Yang Jou. **A real-time and bandwidth guaranteed arbitration algorithm for SoC bus communication**. In *Design Automation, 2006. Asia and South Pacific Conference on*, pages 6–pp. IEEE, 2006. 11

[32] Benny Akesson, Liesbeth Steffens, Eelke Strooisma, and Kees Goossens. **Real-Time Scheduling Using Credit Controlled Static Priority Arbitration**. In *RTCSA '08*. 11, 12, 51

[33] Marcel Steine, Marco Bekooij, and Maarten Wiggers. **A Priority-Based Budget Scheduler with Conservative Dataflow Model**. In *Proceedings of the 2009*

12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools, DSD '09, pages 37–44, Washington, DC, USA, 2009. IEEE Computer Society. 11, 12

[34] Roman Bourgade, Christine Rochange, Marianne De Michiel, and Pascal Sainrat. **MBBA: a Multi-Bandwidth Bus Arbiter for hard real-time**. In *Embedded and Multimedia Computing (EMC), 2010 5th International Conference on*, pages 1–7. IEEE, 2010. 11

[35] Madhavapeddi Shreedhar and George Varghese. **Efficient fair queueing using deficit round robin**. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '95, New York, NY, USA, 1995. ACM. 11

[36] Hardik Shah, Andreas Raabe, and Alois Knoll. **Bounding WCET of Applications Using SDRAM with Priority Based Budget Scheduling in MPSoCs**. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12, pages 665–670, San Jose, CA, USA, 2012. EDA Consortium. 11, 12, 39, 40

[37] EAW Tomlow. **Priority budget scheduling using dataflow.** `http://alexandria.tue.nl/extra1/afstversl/wsk-i/tomlow2013.pdf`. Accessed: 2014-05-30. 12

[38] Dimitrios Stiliadis and Anujan Varma. **Latency-rate servers: a general model for analysis of traffic scheduling algorithms**. 1998. 12, 49

[39] Jan Staschulat and Marco Bekooij. **Dataflow models for shared memory access latency analysis**. In *Proc. EMSOFT*, 2009. 12

[40] Firew Siyoum, Benny Akesson, Sander Stuijk, Kees Goossens, and Henk Corporaal. **Resource-Efficient Real-Time Scheduling Using Credit-Controlled Static-Priority Arbitration**. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2011 IEEE 17th International Conference on*, **1**, pages 309–318, Aug 2011. 12

[41] Hardik Shah, Alois Knoll, and Benny Akesson. **Bounding SDRAM interference: detailed analysis vs. latency-rate analysis**. In *Date '13, Grenoble, France.* 12, 39, 50, 53, 62

[42] Alok Lele, Orlando Moreira, and Pieter JL Cuijpers. **A new data flow analysis model for TDM**. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 237–246. ACM, 2012. 12

[43] Christof Pitter and Martin Schoeberl. **A real-time Java chip-multiprocessor**. *ACM Transactions on Embedded Computing Systems (TECS)*, **10**(1):9, 2010. 12

[44] Hermann Kopetz and Günther Bauer. **The time-triggered architecture**. *Proceedings of the IEEE*, **91**(1):112–126, 2003. 12

[45] Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. **Evaluation of resource arbitration methods for multi-core real-time systems**. In *OASIcs-OpenAccess Series in Informatics*, **30**. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2013. 12

[46] Javier Jalle, Jaume Abella, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. **Deconstructing bus access control policies for Real-Time multicores.** In *SIES*, pages 31–38, 2013. 12

[47] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. **Otawa: An open toolbox for adaptive wcet analysis**. In *Software Technologies for Embedded and Ubiquitous Systems*, pages 35–46. Springer, 2011. 13

[48] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. **Chronos: A timing analyzer for embedded software**. *Science of Computer Programming*, **69**(1):56–67, 2007. 13

[49] Gernot Gebhard. **Timing Anomalies Reloaded**. In *WCET 2010*, Dagstuhl, Germany. 13

[50] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. **Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems**. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, **28**(7):966–978, July 2009. 13

[51] Sudipta Chattopadhyay, Lee Kee Chong, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. **A Unified WCET Analysis Framework for Multi-core Platforms**. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2012 IEEE 18th*, pages 99–108, April 2012. 13

[52] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. **Bus-Aware Multicore WCET Analysis through TDMA Offset Bounds**. In *Real-Time Systems (ECRTS), 2011 23rd Euromicro Conference on*, pages 3–12, July 2011. 13

[53] Huping Ding, Yun Liang, and Tulika Mitra. **Shared cache aware task mapping for WCRT minimization**. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 735–740. IEEE, 2013. 13

[54] Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. **Using Measurements as a Complement to Static Worst-Case Execution Time Analysis**. In *Intelligent Systems at the Service of Mankind*, **2**. UBooks Verlag, Dec. 2005. 14

[55] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. **Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software**. In *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, pages 339–349, Nov 2010. 14

[56] Rodolfo Pellizzoni and Marco Caccamo. **Impact of Peripheral-Processor Interference on WCET Analysis of Real-Time Embedded Systems**. *Computers, IEEE Transactions on*, **59**(3):400–415, March 2010. 14

[57] Jan Nowotsch, Michael Paulitsch, Arne Henrichsen, Werner Pongratz, and Andreas Schacht. **Monitoring and WCET Analysis in COTS multi-core-SoC-based Mixed-criticality Systems**. In *DATE '14*. 15

[58] Jingyi Bin, Sylvain Girbal, Daniel Gracia Pérez, Arnaud Grasset, and Alain Merigot. **Studying co-running avionic real-time applications on multi-core COTS architectures**. In *Embedded Real Time Software and Systems conference*, 2014. 15

[59] Ahmed Alhammad and Rodolfo Pellizzoni. **Schedulability Analysis of Global Memory-predictable Scheduling**. In *Proceedings of the 14th International Conference on Embedded Software*, EMSOFT '14, pages 20:1–20:10, New York, NY, USA, 2014. ACM. 15

[60] Benny Akesson, Anca Molnos, Andreas Hansson, Jude Ambrose Angelo, and Kees Goossens. **Composability and Predictability for Independent Application Development, Verification, and Execution**. In Michael Hübner and Jürgen Becker, editors, *Multiprocessor System-on-Chip — Hardware Design and Tool Integration*, chapter 2. Springer, December 2010. 15, 16

[61] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. **Predictable Programming on a Precision Timed Architecture**. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '08, pages 137–146, New York, NY, USA, 2008. ACM. 16

[62] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. **PRET DRAM controller: bank privatization for predictability and temporal isolation**. CODES+ISSS '11, NY, USA. 16

[63] Isaac Liu, Jan Reineke, David Broman, Michael Zimmer, and Edward A Lee. **A PRET microarchitecture implementation with repeatable timing and competitive performance**. In *Computer Design (ICCD), 2012 IEEE 30th International Conference on*, pages 87–93. IEEE, 2012. 16

[64] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. **CoMPSoC: A Template for Composable and Predictable Multi-processor System on Chips**. *ACM Trans. Des. Autom. Electron. Syst.*, **14**(1):2:1–2:24, January 2009. 16

[65] Benny Akesson, Andreas Hansson, and Kees Goossens. **Composable Resource Sharing Based on Latency-Rate Servers**. In *Digital System Design, Architectures, Methods and Tools, 2009. DSD'09. 12th Euromicro Conference on*, pages 547–555. IEEE, August 2009. 16

[66] Stijn Goossens, Jasper Kuijsten, Benny Akesson, and Kees Goossens. **A reconfigurable real-time SDRAM controller for mixed time-criticality systems**.

In *Hardware/Software Codesign and System Synthesis (CODES+ISSS), 2013 International Conference on*, pages 1–10, September 2013. 16

[67] Theo Ungerer, Christian Bradatsch, Mike Gerdes, Florian Kluge, Ralf Jahr, Jörg Mische, and J. Fernandes *et al.* **parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability**. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 363–370, Sept 2013. 16

[68] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. **An Analyzable Memory Controller for Hard Real-Time CMPs**. *Embedded Systems Letters, IEEE*, **1**(4):86–90, Dec 2009. 16

[69] Miloš Panić, German Rodriguez, Eduardo Quiñones, Jaume Abella, and Francisco J. Cazorla. **On-chip Ring Network Designs for Hard-real Time Systems**. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, RTNS '13, pages 23–32, New York, NY, USA, 2013. ACM. 16

[70] Boris Motruk, Jonas Diemer, Rainer Buchty, Rolf Ernst, and Mladen Berekovic. **IDAMC: A Many-Core Platform with Run-Time Monitoring for Mixed-Criticality**. In *High-Assurance Systems Engineering (HASE), 2012 IEEE 14th International Symposium on*, pages 24–31, Oct 2012. 16

[71] Jonas Diemer and Rolf Ernst. **Back Suction: Service Guarantees for Latency-Sensitive On-chip Networks**. In *Networks-on-Chip (NOCS), 2010 Fourth ACM/IEEE International Symposium on*, pages 155–162, May 2010. 16

[72] Jens Sparso, Evangelia Kasapaki, and Martin Schoeberl. **An area-efficient network interface for a TDM-based Network-on-Chip**. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, pages 1044–1047, March 2013. 16

[73] Rasmus Bo Sorensen, Martin Schoeberl, and Jens Sparso. **A light-weight statically scheduled network-on-chip**. In *NORCHIP, 2012*, pages 1–6, Nov 2012. 16

[74] Evangelia Kasapaki, Jens Sparso, Rasmus Bo Sorensen, and Kees Goossens. **Router Designs for an Asynchronous Time-Division-Multiplexed Network-on-Chip**. In *Digital System Design (DSD), 2013 Euromicro Conference on*, pages 319–326, Sept 2013. 16

[75] Florian Brandner and Martin Schoeberl. **Static Routing in Symmetric Real-time Network-on-chips**. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, RTNS, pages 61–70, New York, NY, USA, 2012. ACM. 16

[76] David May. **The XMOS architecture and XS1 chips**. *IEEE Micro*, (6):28–37, 2012. 16

[77] Andy Lindsay. **Propeller Education Kit Labs**. `https://www.jameco.com/Jameco/Products/ProdDS/2109552.pdf`, 2006. Accessed: 17.06.2015. 16

[78] **xCORE multi-core micro controllers overview.** `http://www.xmos.com/published/xcore-multicore-microcontrollers-overview?version=latest`. Accessed: 17.06.2015. 17

[79] Stewart Edgar and Alan Burns. **Statistical analysis of WCET for scheduling**. In *Real-Time Systems Symposium, 2001. (RTSS 2001). Proceedings. 22nd IEEE*, pages 215–224, 2001. 17

[80] Guillem Bernat, Anotione Colin, and Stefan M. Petters. **WCET analysis of probabilistic hard real-time systems**. In *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pages 279–288, 2002. 17

[81] Francisco J Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, et al. **Proartis: Probabilistically analyzable real-time systems**. *ACM Transactions on Embedded Computing Systems (TECS)*, **12**(2s):94, 2013. 17

[82] Robert I Davis, Tullio Vardanega, Jan Andersson, Francis Vatrinet, Mark Pearce, Ian Broster, Mikel Azkarate-Askasua, Franck Wartel, Liliana Cucu-Grosjean, Mathieu Patte, et al. **PROXIMA: A Probabilistic Approach to the Timing Behaviour of Mixed-Criticality Systems**. *Ada User Journal*, **35**(2), 2014. 17

[83] Eduardo Quinones, Emery D. Berger, Guillem Bernat, and Francisco J. Cazorla. **Using Randomized Caches in Probabilistic Real-Time Systems**. In *Real-Time Systems. ECRTS '09.* 17

[84] Luciano Bononi, Nicola Concer, Miltos Grammatikakis, Marcello Coppola, and Riccardo Locatelli. **NoC topologies exploration based on mapping and simulation models**. In *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*, pages 543–546. IEEE, 2007. 22

[85] James Ball. **The Nios II Family of Configurable Soft-Core Processors**. *Hot Chips, Altera*, 2005. 22, 85, 95

[86] Hardik Shah, Kai Huang, and Alois Knoll. **Timing anomalies in multi-core architectures due to the interference on the shared resources**. In *Design Automation Conference (ASP-DAC), 2014 19th Asia and South Pacific*, pages 708–713, Jan 2014. 24, 85, 86, 88

[87] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. **The Mälardalen WCET Benchmarks - Past, Present and Future**. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis*, July 2010. 32, 77, 91

[88] Antonino Tumeo, Christian Pilato, Gianluca Palermo, Fabrizio Ferrandi, and Donatella Sciuto. **HW/SW methodologies for synchronization in FPGA multiprocessors**. In *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*, pages 265–268. ACM, 2009. 37

[89] Jianguo Xing, Wenmin Zhao, and Hua Hu. **An FPGA-based experiment platform for multi-core system**. In *Young Computer Scientists, 2008. ICYCS 2008. The 9th International Conference for*, pages 2567–2571. IEEE, 2008. 37

[90] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Madanlal Musuvathi, Shaz Qadeer, and Thomas Ball. **Chess: A systematic testing tool for concurrent software**. *Microsoft Research*, 2007. 37

[91] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott Mahlke. **The theory of deadlock avoidance via discrete control**. In *ACM SIGPLAN Notices*, **44**, pages 252–263. ACM, 2009. 37

[92] Sven Goossens, Tim Kouters, Benny Akesson, and Kees Goossens. **Memory-map selection for firm real-time SDRAM controllers**. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 828–831. EDA Consortium, 2012. 41

[93] Maarten H. Wiggers, Marco J. G. Bekooij, and Gerard J. M. Smit. **Modelling Run-time Arbitration by Latency-rate Servers in Dataflow Graphs**. In *Proceedingsof the 10th International Workshop on Software &Amp; Compilers for Embedded Systems*, SCOPES '07, pages 11–22, New York, NY, USA, 2007. ACM. 50

[94] **Dynamic Priority Queue: An SDRAM Arbiter With Bounded Access Latencies for Tight WCET Calculation**. Technical report, 2012. 72

[95] Hardik Shah, Kai Huang, and Alois Knoll. **Weighted Execution Time Analysis of Applications on COTS Multi-core Architectures**. Technical Report TUM-I1339, 2013. 75

[96] Jan Nowotsch and Michael Paulitsch. **Leveraging Multi-core Computing Architectures in Avionics**. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 132–143, May 2012. 79, 91

[97] Petar Radojković, Sylvain Girbal, Arnaud Grasset, Eduardo Quiñones, Sami Yehia, and Francisco J. Cazorla. **On the Evaluation of the Impact of Shared Resources in Multithreaded COTS Processors in Time-critical Environments**. *ACM Trans. Archit. Code Optim.*, **8**(4):34:1–34:25, January 2012. 80, 91

[98] Hardik Shah, Andreas Raabe, and Alois Knoll. **Challenges of WCET Analysis in COTS Multi-core due to Different Levels of Abstraction**. *Workshop on High-performance and Real-time Embedded Systems (HiRES 2013)*, 2013. 85, 95

[99] David Flynn. **AMBA: enabling reusable on-chip designs**. *Micro, IEEE*, **17**(4):20–27, 1997. 85, 95

[100] Will Remaklus. **On-chip bus structure for custom core logic designs**. In *Wescon/98*, pages 7–14. IEEE, 1998. 85, 95

[101] Thomas Lundqvist and Per Stenström. **Timing anomalies in dynamically scheduled microprocessors**. In *Proc. RTSS*, 1999. 86

[102] Jan Reineke, Björn Wachter, Stefan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. **A definition and classification of timing anomalies**. In *OASIcs-OpenAccess Series in Informatics*, **4**. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2006. 86

[103] Mikel Fernández, Roberto Gioiosa, Eduardo Quiñones, Luca Fossati, Marco Zulianello, and Francisco J Cazorla. **Assessing the suitability of the NGMP multi-core processor in the space domain**. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 175–184. ACM, 2012. 91

[104] Anton Hattendorf, Hardik Shah, and Andreas Raabe. **TASERS - TIME AND SPACE SEPARATED EMBEDDED REAL-TIME SYSTEM WITH SHARED SDRAM**. In *in University booth, Design, Automation and Test in Europe*, DATE '12, San Jose, CA, USA, 2012. EDA Consortium. 101

[105] Anton Hattendorf, Andreas Raabe, and Alois Knoll. **Shared memory protection for spatial separation in multicore architectures**. In *Industrial Embedded Systems (SIES), 2012 7th IEEE International Symposium on*, pages 299–302, June 2012. 102, 104, 105, 106

[106] **Rosetta code (Mandelbrot set).** `http://rosettacode.org/wiki/Mandelbrot_set`. Accessed: 2015-06-17. 107

[107] Michael Armbruster, Ludger Fiege, Gunter Freitag, Thomas Schmid, Gernot Spiegelberg, and Andreas Zirkler. **Ethernet-Based and Function-Independent Vehicle Control-Platform: Motivation, Idea and Technical Concept Fulfilling Quantitative Safety-Requirements from ISO 26262**. In Gereon Meyer, editor, *Advanced Microsystems for Automotive Applications 2012*, pages 91–107. Springer Berlin Heidelberg, 2012. 109

[108] Carl Bergenhem, Steven Shladover, Erik Coelingh, Christoffer Englund, and Sadayuki Tsugawa. **Overview of platooning systems**. In *Proceedings of the 19th ITS World Congress, Oct 22-26, Vienna, Austria (2012)*, 2012. 112

[109] Gary Bradski and Adrian Kaehler. *Learning OpenCV: Computer vision with the OpenCV library.* " O'Reilly Media, Inc.", 2008. 113

[110] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. **ROS: an open-source Robot Operating System**. In *ICRA workshop on open source software*, **3**, page 5, 2009. 113