Many-Core Timing Analysis of Real-Time Systems

Hamza Rihani

Communauté
UNIVERSITÉ Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**

Arrêté ministériel : 25 mai 2016

Présentée par

Hamza RIHANI

Thèse dirigée par **Matthieu MOY**
et co-encadrée par **Claire MAÏZA**

préparée au sein du **Laboratoire VERIMAG**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Analyse temporelle des systèmes temps-réels sur architectures pluri-cœurs

Many-Core Timing Analysis of Real-Time Systems

Thèse soutenue publiquement le **1 décembre 2017**,
devant le jury composé de :

Monsieur Robert I. DAVIS
Directeur de recherche, Université de York - Royaume-Unis, Président

Madame Christine ROCHANGE
Professeur, Université de Toulouse III Paul Sabatier - France, Rapporteur

Monsieur Jan REINEKE
Professeur, Université de la Sarre - Allemagne, Rapporteur

Monsieur Benoît DUPONT DE DINECHIN
Directeur de la Technologie, Kalray SA - France, Examinateur

Monsieur Matthieu MOY
Maître de conférences, Université Lyon 1 - France, Directeur de thèse

Madame Claire MAÏZA
Maître de conférences, Grenoble INP - France, Co-Encadrant de thèse

MANY-CORE TIMING ANALYSIS OF REAL-TIME SYSTEMS

and Its Application to an Industrial Processor

BY

HAMZA RIHANI

To obtain the academic degree of:
Doctor of Philosophy in Computer Science

Verimag
Univ. Grenoble Alpes

Advisor: Dr. Matthieu Moy
Supervisor: Dr. Claire Maïza

December 2017

To my teachers

ABSTRACT

Predictability is of paramount importance in real-time and safety-critical systems, where non-functional properties – such as the timing behavior – have high impact on the system's correctness. As many safety-critical systems have a growing performance demand, classical architectures, such as single-cores, are not sufficient anymore. One increasingly popular solution is the use of multi-core systems, even in the real-time domain. Recent many-core architectures, such as the Kalray MPPA, were designed to take advantage of the performance benefits of a multi-core architecture while offering certain predictability. It is still hard, however, to predict the execution time due to interferences on shared resources (e.g., bus, memory, etc.).

To tackle this challenge, Time Division Multiple Access (TDMA) buses are often advocated. In the first part of this thesis, we are interested in the timing analysis of accesses to shared resources in such environments. Our approach uses Satisfiability Modulo Theory (SMT) to encode the semantics and the execution time of the analyzed program. To estimate the delays of shared resource accesses, we propose an SMT model of a shared TDMA bus. An SMT-solver is used to find a solution that corresponds to the execution path with the maximal execution time. Using examples, we show how the worst-case execution time estimation is enhanced by combining the semantics and the shared bus analysis in SMT.

In the second part, we introduce a response time analysis technique for Synchronous Data Flow programs. These are mapped to multiple parallel dependent tasks running on a compute cluster of the Kalray MPPA-256 many-core processor. The analysis we devise computes a set of response times and release dates that respect the constraints in the task dependency graph. We derive a mathematical model of the multi-level bus arbitration policy used by the MPPA. Further, we refine the analysis to account for (i) release dates and response times of co-runners, (ii) task execution models, (iii) use of memory banks, (iv) memory accesses pipelining. Further improvements to the precision of the analysis were achieved by considering only accesses that block the emitting core in the interference analysis. Our experimental evaluation focuses on randomly generated benchmarks and an avionics case study.

Keywords: shared resource interference, many-core processors, worst-case execution time, response time, timing analysis, real-time systems.

RÉSUMÉ

La prédictibilité est un aspect important des systèmes temps-réel critiques. Garantir la fonctionnalité de ces systèmes passe par la prise en compte des contraintes temporelles. Les architectures mono-cœurs traditionnelles ne sont plus suffisantes pour répondre aux besoins croissants en performance de ces systèmes. De nouvelles architectures multi-cœurs sont conçues pour offrir plus de performance mais introduisent d'autres défis. Dans cette thèse, nous nous intéressons au problème d'accès aux ressources partagées dans un environnement multi-cœur.

La première partie de ce travail propose une approche qui considère la modélisation de programme avec des formules de satisfiabilité modulo des théories (SMT). On utilise un solveur SMT pour trouver un chemin d'exécution qui maximise le temps d'exécution. On considère comme ressource partagée un bus utilisant une politique d'accès multiple à répartition dans le temps (TDMA). On explique comment la sémantique du programme analysé et le bus partagé peuvent être modélisés en SMT. Les résultats expérimentaux montrent une meilleure précision en comparaison à des approches simples et pessimistes.

Dans la deuxième partie, nous proposons une analyse de temps de réponse de programmes à flot de données synchrones s'exécutant sur un processeur pluri-cœur. Notre approche calcule l'ensemble des dates de début d'exécution et des temps de réponse en respectant la contrainte de dépendance entre les tâches. Ce travail est appliqué au processeur pluri-cœur industriel Kalray MPPA-256. Nous proposons un modèle mathématique de l'arbitre de bus implémenté sur le processeur. De plus, l'analyse de l'interférence sur le bus est raffinée en prenant en compte : (i) les temps de réponse et les dates de début des tâches concurrentes, (ii) le modèle d'exécution, (iii) les bancs mémoires, (iv) le *pipeline* des accès à la mémoire. L'évaluation expérimentale est réalisé sur des exemples générés aléatoirement et sur un cas d'étude d'un contrôleur de vol.

Mots clés : interférences sur ressources partagées, processeurs pluri-cœurs, temps de réponse, temps d'exécution pire-cas, analyse temporelle, système temps-réel.

PUBLICATIONS

Some ideas and figures have appeared previously in the following publications:

[RMM16] Hamza Rihani, Claire Maiza, and Matthieu Moy. "Efficient Execution of Dependent Tasks on Many-Core Processors." In: *RTSOPS 2016*. 7th International Real-Time Scheduling Open Problems Seminar. Toulouse, France, July 2016.

[Rih+15] Hamza Rihani, Matthieu Moy, Claire Maiza, and Sebastian Altmeyer. "WCET analysis in shared resources real-time systems with TDMA buses." In: *RTNS 2015*. 23rd International Conference on Real-Time Networks and Systems. Nov. 2015.

[Rih+16] Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I. Davis, and Sebastian Altmeyer. "Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor." In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS)*. 2016, pp. 67–76.

ACKNOWLEDGMENTS

This thesis has been a journey of learning and self-improvement through hard work. The work presented here would not be possible without the participation of many people.

To my advisors Matthieu Moy and Claire Maïza who remind of what Stephen Covey wrote in his book: *"Be a light, not a judge, be a model not a critic"*. They were a guiding light and inspiring models throughout my journey. I have an endless gratitude toward them for their trust and encouragement. They have been my mentors whom I look up to. Thanks to Matthieu for showing me that working with passion leads to higher quality results. He amazed me with his intelligence and brilliance in facing hard challenges. Thanks to Claire who knew how to motivate and push me when I needed it the most. Her vibrant energy and the synergy she brought in this work helped it to progress and result in interesting collaborations.

My gratitude goes to the jury members who took the time to evaluate my PhD thesis and for their decision to entitle me with the Doctor degree. I thank Christine Rochange, Jan Reineke, for reviewing the manuscript. Thanks to Benoît Dupont de Dinechin and Robert I. Davis for being examiners.

This PhD thesis is funded by the grant *CAPACITES* from the french *Ministère de l'économie, des finances et de l'industrie*. I would like to thank all the partners and participants in this project; I had fruitful collaborations with Isabelle Puaut and Damien Hardy from IRISA, as well as Christine Rochange, Hugues Cassé, and Wei-Tsun Sun from IRIT. Many thanks to Benoît Dupont de Dinechin, CTO of Kalray, who was always available for discussions despite his busy schedule. Without the openness and transparency of Kalray, the work of this thesis would not have been possible. I would like to salute all the employees of Kalray who are always ready to help and to answer any request.

Special thanks go to all the people of the Real-Time Systems community whom I met during academic events. Thanks to my co-authors Sebastian Altmeyer and Robert I. Davis. The discussion we had during RTNS 2015 was a starting point of what gave a large part of the work presented in this thesis. Thanks to Sophie Quinton, Joël Goosens, Vincent Nélis, Jan Reineke, for the many meetings and insightful discussions we had.

Many thanks to the members of Verimag lab for all the formal and informal discussions and for simply being a part of this adventure. I was fortunate to be surrounded with wonderful and inspiring people. In particular, thanks to Florence Maraninchi and Nicolas Halbwachs, successive directors of Verimag, who warmly welcomed me in this family. Thanks to Oded Maler and Susanne Graf for their collections of interesting books. Thanks to David Monniaux and Julien Henry for their help with the tool PAGAI. Thanks to the administrative staff and the system administrators of Verimag whose support provides a good environment for efficient work.

Furthermore, thanks to my colleagues and friends: Denis Becker, my office mate with whom I shared the stress of writing, Amaury Graillat, Vera Shalaeva, Thomas Rubiano, Maxime Puys, Anaïs Durand, Alexandre Maréchal, Valentin Touzeau, Alexandre Rocca, Irini Mens, Hang Yu, Mahieddine Dellabani, Lotfi Mediouni, Laurent Lemke, and all the current and future doctors at Verimag with whom I shared lunches and coffee breaks.

My life long friends: Ismail, Amine, Sami, and Farouk. To whom I say thanks for always being there for me. It is rare to have such persons who can ignite enthusiasm and ambition back whenever it is needed.

To Yuliia, who shared the bitter and the sweet of this thesis. She proofread my chapters and motivated me throughout the writing process. I am endlessly grateful to her.

Finally, to my parents, my sister, and my brothers, for all their unconditional love and support. Without them, I would not be who I am today.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

CFG Control-Flow Graph

COTS Common-Off-The-Shelf

CRPD Cache Related Preemption Delay

DAG Directed Acyclic Graph

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

DSU Debug Support Unit

FIFO First-In-First-Out

ILP Integer Linear Programming

IPET Implicit Path Enumeration Technique

LSU Load Store Unit

MBTA Measurement-Based Timing Analysis

MD Memory Demand

MPPA Massively Parallel Processor Array

MRTA Multi-core Response Time Analysis

NoC Network-on-Chip

RM Resource Manager

RR Round-Robin

RTOS Real-Time Operating System

SDF Synchronous Data-Flow

SDRAM Synchronous Dynamic Random Access Memory

SMEM Shared Memory

SMT Satisfiability Modulo Theory

SRAM Static Random Access Memory

TDMA Time Division Multiple Access

VLIW Very Long Instruction Word

WCET Worst-Case Execution Time

WCRT Worst-Case Response Time

Chapter 1

INTRODUCTION

1.1 CONTEXT AND MOTIVATION

Time matters in safety-critical real-time systems. The predictability of these systems is needed in order to guarantee certain security and safety requirements. Determining Worst-Case Execution Times (WCET) of a software (or a piece of code) has received a major focus of research in the field of embedded systems.

Critical embedded systems (e.g., avionics, medical devices, automotive, etc.) have traditionally used simple hardware systems in order to control their predictability: not only the computations done in these systems must yield the correct result, but they also must deliver this result before a given deadline. Many critical embedded systems have an increasing demand in computing speed. For example, self-driving cars require more performance to process video streams, and make decisions in real-time according to the surrounding environment.

Old and simple processors are not sufficient anymore. Unfortunately, common optimizations done in general-purpose electronic systems (e.g., complex cache policy, branch prediction, etc.) are meant to optimize the average case, but not the worst case. As a consequence, general-purpose processors are not suitable.

WCETs must be computable in order to ensure that a program has enough time to finish before its deadline. The execution time depends on the inputs of the program, that may determine the execution path, and the state of the hardware architecture. The following example illustrates the problem:

Example 1. *Function f in Figure 1.1 represents a simple toy task running on a single-core. Its control-flow graph is annotated with timing information. The execution path in the function depends on the input value of x. For instance, when $x = 3$, the execution time is $b_1 + b_3 + b_4 = 5 + 7 + 3 = 15$ time units. On the other hand, for $x = 41$ the execution time is $b_1 + 10b_2 + b_4 = 5 + 9 \times 10 + 3 = 98$ time units. To allocate a proper time budget for the function f, its worst-case execution time needs to be known in advance. For this example, the worst-case execution time is $b_1 + 39b_2 + b_4 = 5 + 39 \times 10 + 3 = 398$ time units when $x = 11$.*

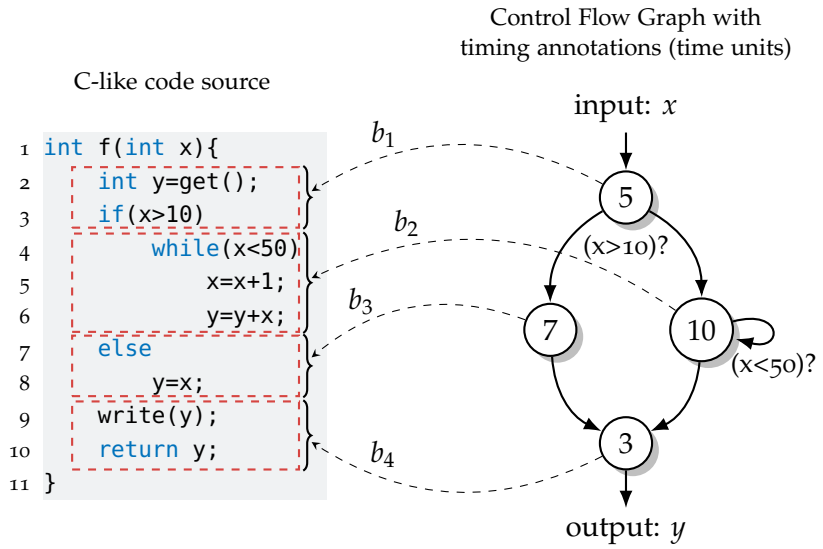


Figure 1.1: Role of execution time in timing constrained systems

Example 1 assumes known and deterministic worst-case execution times of each instruction. On a single-core processor this is done by analyzing the states of the processor pipeline and the cache memory. The challenge here is to find the longest execution path that maximizes the execution time.

Multi-core and many-core processors are considered to be a fit solution to increasing computing speed demand, energy consumption, and area efficiency in embedded systems. Such processors typically consist of several cores sharing some hardware resources such as cache memories, buses, shared global memory, etc. The execution time becomes less predictable due to potential contention on shared resources. In Figure 1.1, reading and writing of y (lines 2 and 9 respectively) may be subject to timing interference from co-runner tasks on the platform, especially if y is in a shared memory and not in a private cache memory.

Some architectures were designed with both performance and predictability in mind, and would be good candidates to run critical, real-time and high-performance embedded software. One of them is the industrial MPPA [1] many core architecture. It offers a very high degree of parallelism (256 cores), and allows several computations to be done in parallel with limited interference. The individual cores of the MPPA architecture have already been studied and showed to behave well with respect to real-time.

Due to the shared resources, two additional problems remain in multi-core and many-core systems: (i) how to control the interference between two computations running in parallel and sharing the same memory (i.e., how much can one computation slow down the other?), (ii) how to analyze the performances of the communications between cores.

1 **M**assively **P**arallel **P**rocessor **A**rray, a many-core processor from Kalray http://www.kalrayinc.com

1.2 SUMMARY OF CONTRIBUTIONS

This thesis addresses the problem of shared resources access delays. The main contributions consist of two different approaches for shared resource timing analysis. We first propose an accurate analysis based on *Satisfiability Modulo Theory* (SMT) to encode a program's semantics and accesses to shared resources with a *Time Division Multiple Access* (TDMA) bus. This work is published in [Rih+15], in which we propose:

- **SMT Encoding of Shared TDMA Bus:** We propose an SMT expression that returns a delay of a bus access taking into account TDMA bus.
- **Evaluation of Several Encodings:** We show that several encodings are possible. Although they are equivalents and functionally give the same results, their solving time may vary greatly. We determine the best encoding that optimizes the solving time.

In our second approach, we address other arbitration policies with an application to an industrial many-core processor. Such architectures can execute larger applications where the interference is greater. We propose a scalable analysis that accurately accounts for the interference. Here, we focus on dependent task graphs with an application to periodic *Synchronous Data Flow* applications. We consider static non-preemptive scheduling. To accurately account for the interference, we use the dependency graph to exclude tasks that cannot execute at the same time.

- **Double Fixed-Point Algorithm:** To statically schedule a task graph on a multi-core (i.e., determine start and finish dates of each task), worst-case bounds on execution times are required to ensure that dependencies are respected. The execution time of a task depends on the interference on shared resources. This in turn depends on the number of co-runner tasks that potentially access the shared resource. To determine the co-runners, the start and finish dates of each task are required. We provide a fixed-point algorithm that solves this cyclic dependency. Our algorithm starts with an initial mapping and execution order, and produces a scheduling that tightly accounts for the interference.
- **Proof-of-Termination:** We provide a proof of termination of the above algorithm.

To accurately account for the interference on shared resources, we target a specific architecture model. Particularly, we focus on the industrial many-core processor Kalray MPPA-256 that fits this model.

- **Model of an Industrial Multi-Level Arbiter:** We present a mathematical model of an industrial shared resource arbiter. The arbiter is implemented with several levels of round-robin and fixed-priority policies.
- **Hardware Features:** Our model supports delays from access bursts and access pipelining. It also takes into account partitioned shared memory configuration.

Part of the contributions above appears in [Rih+16]. This work is used in the framework of the *CAPACITES* project [Cap]. Our contribution on task graphs is used in a tool-chain that transforms a synchronous data-flow program into a binary with a mapping and a scheduling on a many-core processor.

1.3 THESIS OUTLINE

The thesis is organized in three parts.

Part i presents the state-of-the-art of timing analysis of multi-core and many-core systems.

- We present basic notions and background in Chapter 2.
- Chapter 3 gives an overview of existing and related work.

The core content of the thesis is presented in Part ii.

- Our contribution on TDMA buses is in Chapter 4.
- Chapter 5 presents our response time analysis.
- Our model of shared bus arbitration policy applied to the Kalray-MPPA many-core processor is given in Chapter 6.

Finally in Part iii, we evaluate the analysis of the Kalray-MPPA :

- Experimental evaluation and methodology is presented in Chapter 7.
- Chapter 8 gives a positioning of the work within the *CAPACITES* project.

The conclusion and future work are given in Chapter 9.

Part I

STATE-OF-THE-ART

Chapter 2

BACKGROUND

In this chapter, we present the context and foundation of the thesis. We target hard real-time systems on multi-core and many-core architectures. The organization of this chapter is as follows: In Section 2.1, we globally define real-time systems and their requirements. In Section 2.2, we present different application models and the hardware architectures in Section 2.3. Section 2.4 demonstrates how the software can run on the hardware. Section 2.5 introduces static timing analysis of such systems with illustrations of a timing analysis framework. Finally, Section 2.6 sets the context of this thesis.

2.1 Real-Time Systems

A real-time system is a system where the timing behavior is as important as the functional one. The main characteristic of a real-time system is the execution under timing constraints or deadlines; not only the computed results must be correct, they also must be provided within a defined time delay. Such systems are typically found in reactive environments, where a computing system must interact with and control phenomena in the physical world.

A recurring example of such systems is a flight management system illustrated in Figure 2.1 and Example 2; it reads the position's values from sensors alongside with the pilot's inputs, and control the aircraft accordingly. These reactive systems exist in many fields, from automotive systems, to avionics, and through nuclear plants and medical equipment.

We distinguish real-time systems depending on their requirements into *hard* and *soft* real-time systems. A system can be hard or soft depending on the damages of failures to respond within the deadlines. A hard real-time system is a system where meeting the deadlines is a strong requirement; missing the deadline may lead to a catastrophic loss. This is by definition what is called a *safety-critical system*. A soft real-time system is a system where meeting the deadline is a desirable requirement but not necessary; the system may not fail if a deadline is missed and the consequences are not critical. In-between soft and hard, a *firm* system is a system where the outputs expire when the deadline is missed but the consequences are not critical and the system may not fail. The context of this thesis falls in the hard real-time and safety-critical systems field.

A real-time system is able to react to external signals (usually from sensors) *during* its evolution and within a time deadline. Timing constraints are imposed by the interactions with the physical world. As an illustration, the flow of time is modeled with a directed time line representing the system evolution. An *instant* is a point on this line. A *duration* is an interval in time defined by a *starting event* and a *termination event*. We define here an *event* as any occurrence at an instant of time. Some real-time systems have one or several clocks that tick at equally distant instants [Kop11].

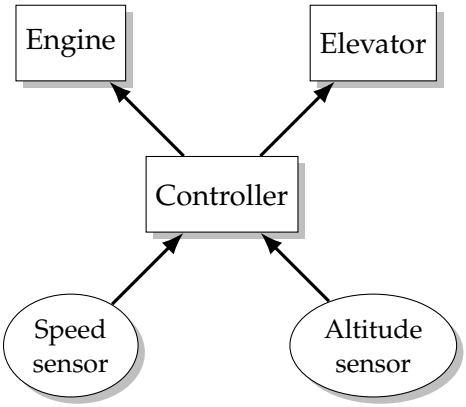


Figure 2.1: Example of a real-time system: Flight Management System

Example 2 (Flight Management System)**.** *Figure 2.1 illustrates a real-time system with the example of a flight management system. This is the system responsible for controlling and maintaining the altitude of the airplane. Inputs are obtained from dedicated sensors: the speed/acceleration sensor and the altitude sensor. Information in the physical world are continuous values which requires a discretization through sampling. The controller then computes outputs and sends commands accordingly to the actuators. The controller also responds to inputs from the operator to adjust the airplane's position. The system is expected to react within time limits and maintain certain performance properties. The controller must respond to all input commands from the operator and also maintain a steady state throughout the evolution of the physical inputs from the sensors.*

The example above, also illustrated in Figure 2.1, is called a *reactive system*. Outputs of such systems must be produced within deadlines relative to the occurrence of inputs. In contrast to this, a system can be *time-aware*. In these systems, there is an explicit reference to time, for instance, a scheduling table that has explicit times of when each task/procedure is executed.

2.1.1 *Requirements*

The design, development, and verification of safety-critical real-time embedded systems are subject to specific requirements that follow from guideline and standards such as DO-178B/C for avionics and ISO26262 for automotive systems. For example, Table 2.1 illustrates the assurance levels as defined by the DO-178B and according to their failure consequences. Hard real-time systems are certified at Levels `A` and `B`. Both the functional and the timing behavior of such systems is required to be correct. In order to ensure that applications meet their deadlines, predictable upper bounds are required on the execution times of software components. This enables the derivation of sound upper bounds on the worst-case response times (WCRT), from input stimulus to output response, and the verification of their compliance with timing constraints.

Assurance level	**Failure condition**	**Details**
Level A	Catastrophic	Crash of the system. Loss of human lives
Level B	Hazardous/Severe	Large performance degradation that may leads to serious/fatal injuries
Level C	Major	Failure is noticeable but with a lesser impact than Hazardous.
Level D	Minor	Failure is noticeable but the impact is lesser than Major
Level E	No effect	Failure is not noticeable

Table 2.1: Design Assurance Levels in DO-178B[1]

2.1.1.1 *Functional Requirements*

Functional requirements consider the output result produced by a real-time system. Failure of the system due to bugs might not be tolerated at certain assurance levels. Unfortunately there have been real-world cases where this had occurred. In 1996, the satellite launcher *Ariane 5* exploded mid-air in less than 40 seconds after its launch [JM97]. Among other identified problems, a conversion from 64-bit floating-point to 16-bit signed integer resulted in a register overflow. A most recent example from 2015, where failure of the generator control unit in the *Boeing 787* airplane occurred after a continuous powering for 248 days [Mou15]. This was caused by a counter overflow in software. Such failures cause financial costs and

1 RTCA/DO-178B "Software Considerations in Airborne Systems and Equipment Certification", December 1, 1992.

may endanger human life, although it could be detected at an early stage of the software design process.

There exist several techniques to verify the correctness of software. Ranging from static verification with formal methods, such as *abstract interpretation* [CC77] and *model checking* [EC80], to more dynamic and on-line testing such as Assertion Based Verification [Dah+05; Fos09]. While the latter technique is usually faster, it may exhibit coverage related problems. The former technique is sometimes tedious and more complex although it can achieve a high confidence assurance of the system correctness.

2.1.1.2 *Extra-Functional Requirements*

The extra-functional requirements include for example, power consumption, temperature, and execution time. In this work, we focus on the execution time in a real-time system. System's outputs must be produced within a bounded time frame otherwise the system fails. These bounds are often imposed by the physical world. In hard real-time systems, the computations must be finished and the output produced before the occurrence of the next physical event. Thus, it is required that for any input values, the system can still produce outputs before the deadline. To do so, one must prove that the *Worst-Case Execution Time (WCET)* is always smaller than the deadline. In techniques relying only on on-line testing, it is hard (and very unlikely) to reach the worst case. Figure 2.2 illustrates that testing the system represents only a subset of all possible values of the execution time [Wil+08]. Probabilistic techniques [BCP02] can enhance the analysis by giving a probabilistic upper-bound. This result, however, is not guaranteed but limited to a confidence interval.

Static timing verification relies on conservative analyses of software and hardware. By making over-approximations and pessimistic assumptions, the estimated WCET is a guaranteed upper-bound on the worst-case execution time. In this thesis, we refer to the estimated WCET simply as WCET. The context of this thesis does not cover on-line techniques and falls only in the formal verification method. We also consider only the timing aspects of the real-time systems. Functional correctness and other extra-functional aspects are out of scope of this thesis.

2.1.2 *Challenges in the Verification of Real-Time Systems*

Many critical embedded systems have increasing demand in computing speed, and old, simple processors are not sufficient anymore. Unfortunately, common optimizations done in general-purpose electronic systems are meant to optimize the average-case, but not the worst case. As a consequence, general-purpose processors are not suitable. Besides this, the behavior of such processors is hardly, if at all, time-predictable; time predictability is the ability to tell how much time a program will execute in the worst case. Features such as complex cache policy, Out-of-Order (OoO) execution, branch predictions, or speculations allow to speedup the average-case performance but not the worst case. A model of such architectures must cover all the possible outcomes of the hardware features. This makes the analysis very complex. New multi-core platforms were designed to offer high performance in real-time systems while keeping the predictability in mind. Depending on abstract

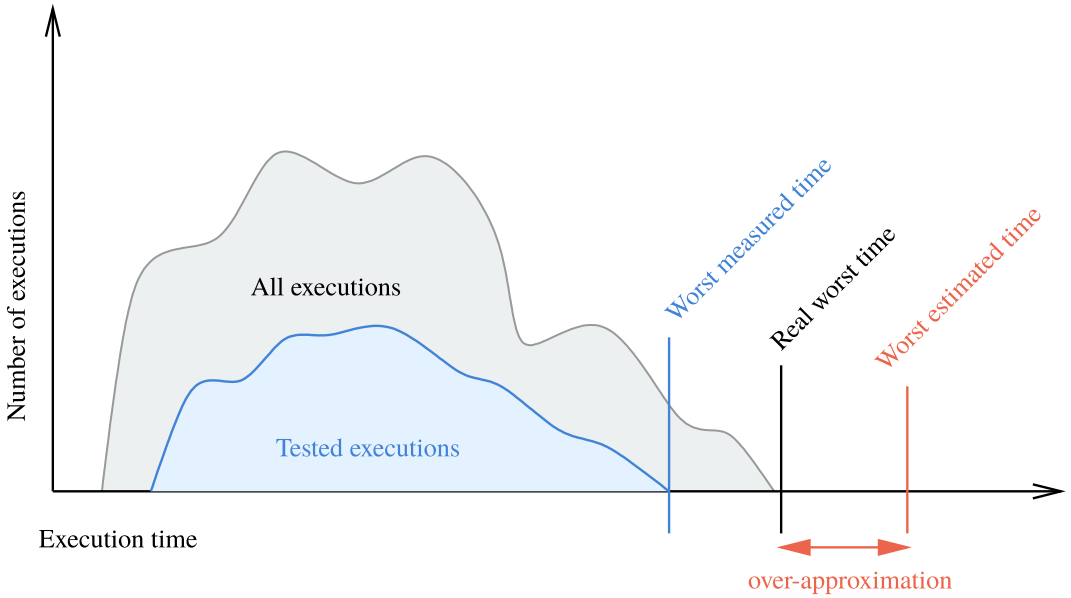


Figure 2.2: Worst-case execution time

models of such architectures, their analysis can achieve a trade-off between precision and scalability.

With the relatively simple hardware (single processor, no advanced hardware acceleration features, etc.), it was possible to ensure predictability of execution times, to tightly bound the WCET by static analysis. Due to an increasing demand for compute performance in real-time systems, combined with Size, Weight, and Power consumption (SWaP) requirements, the emphasis has shifted from ever faster single-core processors, which had reached physical limitations due to issues with heat dissipation, to more complex multi-core and many-core architectures. The shift is also motivated by the ability to reduce the number of hardware platforms in a real system; a single chip with several cores can run different applications with a proper isolation between them.

Ensuring predictable and tightly bounded timing behavior in such systems is very challenging. This is due to the contention for multiple shared hardware resources between co-running applications on the different processors. Examples include contention for cache, network or memory bus, memory controllers, etc. For example, on the Freescale P4080, the latency of a read operation varies from 40 to 600 cycles depending on the total number of cores running competing tasks [Now+14]. Similarly, a 14 times slowdown has been reported [Rad+12] due to interference on L2-cache for tasks running on Intel Core 2 Quad processors. Recent work [VYF16] shows that even with cache partitioning, contention for registers accessed on both cache hits and misses can cause a 21 times slowdown due to contention caused by co-runners on the ARM cortex A-15 multi-core architecture.

Despite the challenges described above, efforts were made to simplify the hardware architecture used in real-time systems and improve its predictability. Along this direction in hardware, static timing analysis with correct assumptions can give strong correctness guarantees on the system behavior. The over-approximations, as shown in Figure 2.2 and pessimistic assumptions ensure that the worst case is covered in the analysis. Nevertheless,

if the analysis is too pessimistic, the system may be considered unschedulable while in fact it is. The main challenge here is to reduce the over-approximation. One way to do this is to avoid considering generic hardware or software; for the best results, the analysis must be tailored to take into account the software and the underlying architecture.

In the remainder of this chapter, we present the different application models and hardware architectures used in real-time systems. This is followed by the definition of the context of this thesis.

2.2 APPLICATION MODELS

Due to the particularity and requirements of real-time systems, real-time software follows a certain model which is different from desktop applications. In this section, we present the real-time application models and the different notions used in this thesis.

2.2.1 *Task Models*

Real-time systems are characterized by a set of timing properties. Let's assume a set of tasks $\Gamma = \{\tau_0, \tau_1, \ldots, \tau_n\}$ executing on a processor. We define the *job* J_k^i of task τ_i as the k^{th} instance of τ_i. In this thesis, we use the terms *job* and *task* interchangeably unless it is explicitly specified in the context of speech.

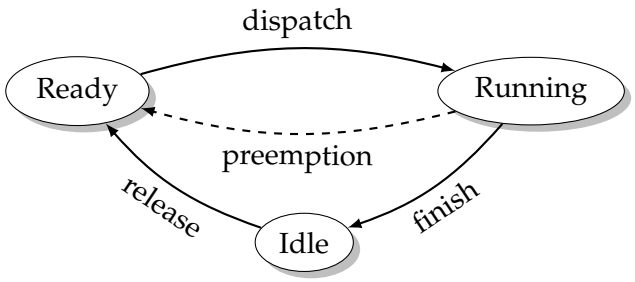


Figure 2.3: Task's state transitions [Alt13]

Figure 2.3 shows transition state in a task scheduling. When a task is ready for execution, it waits for the scheduler to dispatch it. Depending on the scheduling policy, a task may get preempted and return to the ready state. When the task finished it goes to an idle state and awaits to get released in the next cycle.

2.2.1.1 *Definitions*

In the following, we define a set of notions common to real-time systems [But04]. Figure 2.4 illustrates these notions on a time line.

Definition 1 (Release date)**.** *Denoted by* rel_i *and also called* arrival time*, it is the time when* τ_i *is eligible for execution.*

In practice, the release date may not be exactly defined but occurs within a bounded time interval. This variation in delay is called *jitter*. The time duration between the release and

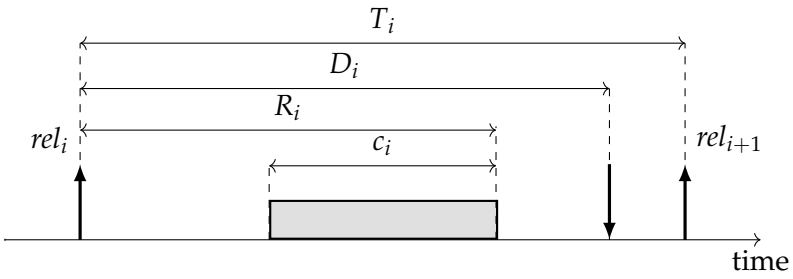


Figure 2.4: Illustration of a task/job execution

the start of execution of τ_i may be fixed or bounded within an interval of a *minimum delay* and a *maximum delay*.

Definition 2 (Execution time)**.** *Denoted by* C_i*, it is the duration that takes* τ_i *to finish if executed without any interference or preemption. Lower and upper bounds on the execution time are called* BCET *and* WCET *(best-case and worst-case execution times) respectively.*

Definition 3 (Response time)**.** *Denoted by* R_i*, the time duration between the release date of* τ_i *and its completion time (finish time).*

Definition 4 (Deadline)**.** *Denoted by* D_i*, a timing constraint by which* τ_i *must complete its execution.*

Definition 5 (Inter-arrival time)**.** *Denoted by* T_i*, the time duration between two successive release dates of* τ_i*.*

Regarding the inter-arrival time, a task τ_i is *sporadic* when its jobs arrive at variable times with a guaranteed minimum interval. We find such configuration in event-triggered systems [BW16]. On the other hand, a task is periodic when its jobs arrive at a regular interval of time. The inter-arrival time T_i, in this case, is also called a *period*. This configuration is found in time-triggered systems [BW16]. The focus of this thesis is only on periodic tasks.

2.2.2 *Synchronous and Asynchronous Task Models*

In the current practices of real-time systems, there exist two programming paradigms: *asynchronous* and *synchronous* paradigms. In the asynchronous task model, events occurs in an undeterministic order. In most cases, they are queued upon reception and treated at some point during the task execution. Parallel and concurrent programming can be asynchronous relying on some asynchronous communication protocol.

Synchronous task models offer a more deterministic behavior, hence they are easier to analyze. In this model, the system's execution is based on the notion of execution steps. Within a step, the components (or tasks) in the system, which met their activation conditions, are expected to progress in their execution. Inputs from the external environment are treated at the next step. Any effect of computation is then propagated through the components in the system. To implement this concept, the assumption is that the real-time system evolves fast enough with regard to the external environment.

The definition above implies that all computations should terminate within the step, i.e., an upper-bound on the execution time should fit within the step bounds. Nonetheless, sources of asynchronous behavior may occur and introduce unexpected delays. Such sources may come from the operating systems' interrupts, the interference on shared resources, variations in the execution times,... etc. To ensure the assumption on the synchronous behavior, it is important that the WCET analysis takes into account such delays.

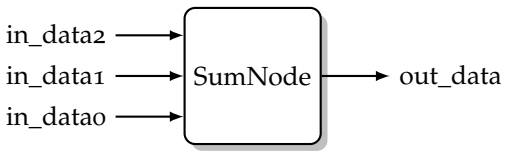
2.2.3 *Synchronous Data-Flow Model*

The *Synchronous Data-Flow (SDF)* model is a paradigm based on *Kahn Process Networks*. It is an application of data-flow programs combined to the synchrony assumption. As a result, there is no notion of time at this level of abstraction; all operations and communications within a step are considered instantaneous. Figure 2.6 illustrates a simple example of a data-flow program; nodes represent the tasks whereas the directed edges represent the communication channels from the producers to the consumers.

In terms of implementation, there exist several languages to write SDF programs. SIGNAL [BGJ91], SCADE [Ber07], and Lustre [Hal+91] are examples of languages specialized in the synchronous programming paradigm. While SCADE is used in industry, SIGNAL and Lustre are mostly used in academic research. Figure 2.5 shows a snippet of a Lustre code. At each step (represented with clock ticks), the values in the input data `in_data0`, `in_data1`,`in_data2` are summed and returned in the output data `out_data`. This operation is assumed to execute in a null time. The communications are implemented with memory buffers (for example FIFOs) and are also assumed instantaneous.

```
node SumNode (in_data0: int,
              in_data1: int,
              in_data2: int)
returns (out_data: int);
let
    out_data=in_data0+in_data1+in_data2;
tel
```

Listing (2.1) Lustre programming language



(b) Dataflow node

Figure 2.5: Example of a Lustre node and its high-level graphical representation

In terms of scheduling, we consider that all tasks in a cycle must complete before the end of the cycle. As a consequence, scheduling can be done on one period; the same schedule is then repeated indefinitely. Therefore, the tasks in the data-flow program are seen as an acyclic dependency graph. A task is released only when all its predecessors have finished their execution, i.e., when they produce tokens for the next tasks. In the example given in Figure 2.6, the output data of task τ_1 must be available to task τ_4 before it can execute. Hence, the release date of task τ_4 should be greater than the finish time of task τ_1. The data

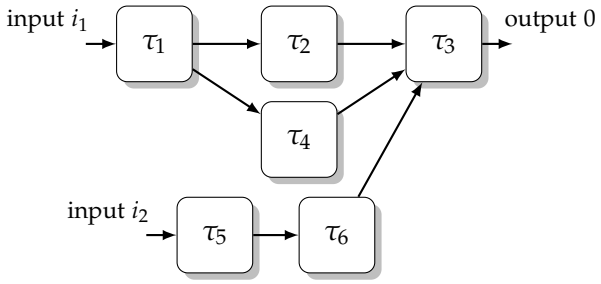


Figure 2.6: Example of a data-flow program

produced is written into a memory location where the consumer task can read it. In the example, tasks τ_2, τ_4, and τ_6 write to the memory of task τ_3.

2.2.4 *Task Scheduling*

Task scheduling is considered as a resource allocation problem. The goal is to allocate a sufficient amount of resources to all the tasks in the system in order to meet their requirements. This processes aims at optimizing objective function according to the system requirements. In this context, task scheduling is the problem of allocating a sufficient amount of processor clock cycles within a scheduling period to each task according to their priorities. For this purpose, there exist several policies among them: *Earliest Deadline First (EDF)*, *Deadline Monotonic (DM)*, *Rate Monotonic (RM)*. Each policy is applied dynamically or statically.

A dynamic scheduling policy is an online policy where the set of tasks might be unknown at the start of the system. Tasks are assigned to an available core *during* run-time and when they are released. The assignment follows a scheduling policy, for instance EDF. A static scheduling policy is performed off-line. This implies that the set of tasks is known in advance in order to compute a scheduling table.

The execution of tasks can be *preemptive*; a task is preempted whenever a higher priority task needs to be executed. Preemption introduces a context-switch overhead and other costs such as the *Cache Related Preemption Delay (CRPD)* [Alt13]. The analysis of such costs often increases the pessimism in the WCET. In the *non-preemptive* scheduling, a dispatched (running) task cannot be stopped until its completion.

A *schedulability test* checks whether a task set is schedulable on a processor. With explicit deadlines, this is can be achieved by checking whether all tasks meet their deadlines. Particular tests exist such as in the case of EDF; a task set Γ is schedulable if its utilization U_Γ is smaller or equal to 1, where $U_\Gamma = \sum_i^n c_i/T_i$ (assuming $D_i = T_i$). All potential interference must be accounted for in the schedulability test and therefore added to c_i when computing the utilization.

2.3 HARDWARE ARCHITECTURES

Execution times dependent on the hardware platform that executes the software. The methodology for a tight analysis of real-time systems requires knowledge on the underlying micro-architecture at a certain abstraction level.

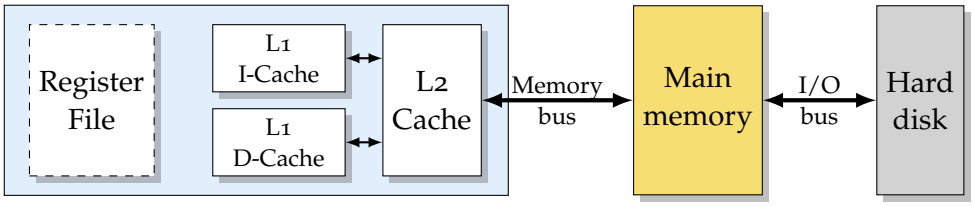


Figure 2.7: Memory hierarchy of a single-core processor [Reio8]

2.3.1 *Multi-core and Many-core Architectures*

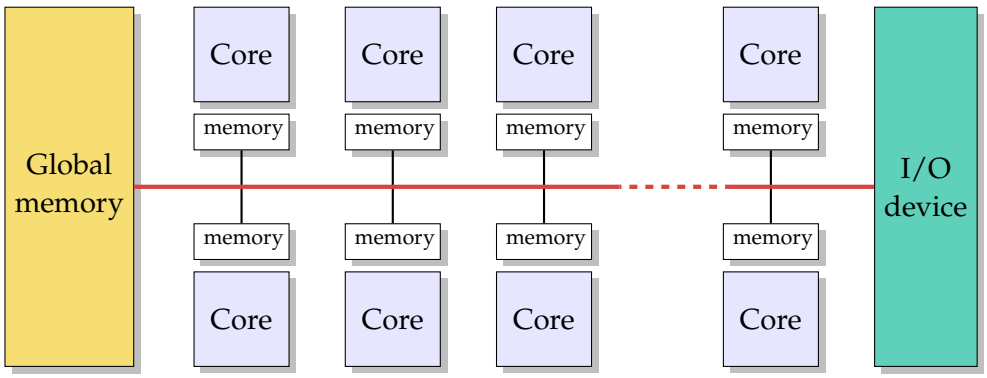


Figure 2.8: An example of a multi-core platform

Multi-core and many-core systems offer more opportunities and a new set of challenges. These platforms have shown capabilities of energy efficiency with a higher computing power in comparison to an equivalent single-core platform. Moreover, a single multi-core chip is capable to run several applications which reduces the number of required platforms. This is important in systems where the space and weight are constrained.

One of the challenges on multi-core and many-cores is the analysis of interference on shared resource. To illustrate this, we take the example of the memory hierarchy as shown in Figure 2.7. Access time varies whether the accessed resource is the cache memory or the main memory. On a single-core, where only one task is running at a time, the access time to the memory is deterministic and easily bounded. On a multi-core, where some levels of the memory hierarchy are shared, the access time depends on the contention from co-runner tasks.

Figure 2.8 illustrates a multi-core system where cores may share one or several resources such as caches, main memory, I/O devices, buses, etc. The presence of shared resources is a source of interference. The execution time does not only depend on the task under analysis but also on other co-runners. Depending on the arbitration policy used at each shared resource, the delay may vary from task to task; examples of arbitration policies are:

- *fixed-priority* arbitration policy assigns priorities to each core (or any other requester of the shared resource) When there is a contention, a higher priority core always gets the resource before lower priority cores. Therefore, the highest priority core suffers less from the delay compared with the lowest priority processor.

- *round-robin* arbitration policy is where all cores have the same priority. The delay due to interference is distributed almost fairly between the cores. One of the main challenges when analyzing such platforms is to tightly and efficiently find upper-bounds of interference on shared resources.
- *first-come-first-served* arbitration policy grants access requests in the order of which they are issued.
- *time division multiple access* arbitration policy assigns a periodic communication slot to each core. Access requests are granted only during their corresponding slots, otherwise, they are stalled.

The fixed-priority, round-robin, and first-come-first-served policies are said to be *work-conserving*. A work-conserving bus arbiter will not idle the bus as long as there are pending requests. In contrast, the time division multiple access policy is *non-work-conserving*, since access requests are stalled until their corresponding slots even when there are no concurrent accesses.

There are other functional challenges introduced in the multi and many-core platforms. Since the code can be executed in parallel, the question is how to split and run parallel segments. The presence of critical sections and concurrency management techniques (such as locks, semaphores,...) introduces waiting times in the execution. Moreover, the parallel tasks can be mapped to different cores where their execution time may depend on the core they are assigned to. In the following, we define $\Pi = \{P_0, P_1, ...P_N\}$ as the set of cores in the platform. The mapping function $Map : \Gamma \rightarrow \Pi$, maps a task in Γ to a core in Π.

Let's assume a platform with 3 cores $\Pi = \{P_0, P_1, P_2\}$. A possible mapping (among others) of the example in Figure 2.6 is $Map(\tau_0) = Map(\tau_1) = Map(\tau_2) = P_0$, $Map(\tau_4) = P_1$, $Map(\tau_5) = Map(\tau_6) = P_2$. Notice that core P_1 runs only task τ_4 which depends on τ_1. Depending on the execution time of τ_4, this may result in P_1 being most of the time idle. Several works were done in the area of task mappings [PNP13; Gia+14; WN15; Per+16a; Ten14] in which the proposed approach finds an "optimal" mapping that leads to specific objectives in terms of performance. In this work, we consider that the mapping is given and fixed (i.e., no task migration).

The terms multi-core and many-core both refer to architectures with more than 2 cores. Multi-core systems have typically between 2 to 32 cores whereas many-core systems have more than 32 cores. At the architectural level, many-cores platforms have their cores disposed in *clusters* (also called *tiles*). A common feature between many-cores is the use of *Networks-on-Chip (NoC)* to ensure high bandwidth and low latency communication channels between clusters. Examples of many-core platforms are: Intel Xeon Phi [Chr12], Tilera 64 [Til12], the STHorm platform [Mel+12], and Kalray MPPA-256 [Din+14b].

2.3.2 *Timing Compositionality*

Predictability of the hardware architecture is one important aspect in hard real-time system. Many of the optimization techniques are used to improve the average case but not the worst case. An architecture with many states may lead to a very pessimistic analysis which increases the over-approximation on the WCET. For this reason, efforts were made to keep

the architecture as simple as possible while maintaining a certain level of performance. It turns out that simple hardware may not be enough to make the platform predictable. While one might think that assuming a local worst-case everywhere leads to the global worst-case, this assumption is not always correct. This is known as a *timing anomaly* [LS99]. There are many sources of timing anomalies. In the following we discuss some examples of timing anomalies.

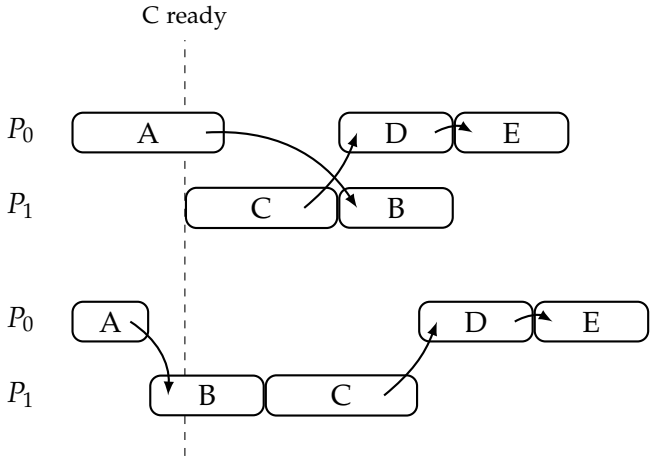


Figure 2.9: Scheduling anomaly

Figure 2.9 illustrates a case of a timing anomaly due to resource scheduling. Assuming a two-core platform executing depending tasks using a list scheduling[2]. The mapping is as follows: $Map(A, D, E) = P_0$ and $Map(C, B) = P_1$. In the first scenario task C is ready during the execution of A and is scheduled to execute on P_1. This, in turns, delays the execution of B which is scheduled after C. In the second scenario, A finishes earlier allowing B to start executing before C is ready. C executes later which delays the execution of D that depends on C. This scenario leads to a greater global execution time despite the fact that A is faster.

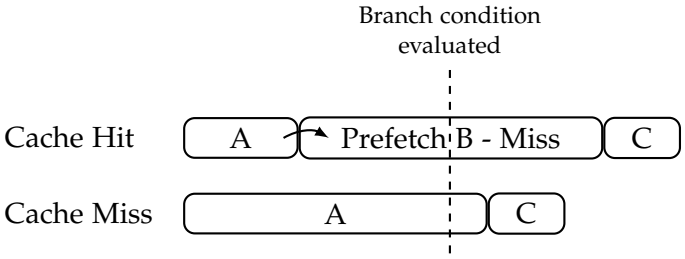


Figure 2.10: Branch speculation anomaly

Branch prediction may also lead to timing anomalies. Figure 2.10 describes a case where a cache miss lead to a faster execution due to branch prediction. A, B, and C are instructions executing on the processor. Branch mispredictions results in unnecessary fetches which may not happen when instruction A hits in the cache.

2 In a list scheduling, tasks are executing as soon as they are ready

Another potential source of anomalies is the cache memory and its replacement policy. The work in [Rei08] focuses on the caches and discusses their predictability. It has shown that some replacement policies such as *pseudo Least Recently Used*, *First-In-First-Out*, *pseudo round-robin* show behavior anomalies where an eligible entry for eviction does not get evicted. This fact affects the cache analysis that aims at determining whether cache accesses result in a hit or miss. [Rei08] concludes that the *Least Recently Used* policy have a sound behavior and does not exhibit this kind of anomalies. Interested readers may refer to [Rei08] for a complete description of the mentioned policies and their timing anomalies.

The timing anomalies may also create new timing anomalies, which in turn creates more anomalies. This effect is known as the *domino effect* [LS99]. By definition, the domino effect leads to unbounded differences in execution times the same piece of code with different hardware states. An example where this may happen is when the program executes a loop with the same body but may lead to a non-convergent states of the pipeline (or caches, or other hardware resource) at each iteration.

With the above definition and example, architectures can be classified as the following [Cul+10; Axe+14]:

- *Fully timing compositional architecture:* There is no timing anomalies and the analysis may follow the local worst-case.
- *Compositional architectures with bounded effect:* The architecture may exhibit some timing anomalies without domino effects. The analysis is done by compensating the anomalies with a constant overhead.
- *Non-compositional architectures:* Such architectures exhibit timing anomalies with domino effects. The analysis must takes into account all possible states of the architecture.

2.3.3 *Predictable Multi-core and Many-core Architectures*

Considering all the challenges described above, the design of multi-core and many-core systems was influenced by the predictability considerations. The work in [Cul+10] propose guidelines to follow when designing a predictable architecture.

To design a predictable multi-core, it is necessary that the cores, which are the base units, are predictable. Efforts in academia and industry have been taken toward this direction. For example, the time-predictable processor Patmos [Sch+14]. It is a *RISC* [3] processor with a *Very Long Instruction Word (VLIW)* pipeline. This processor is designed along with the timing analysis which makes it efficient. The core relies on separate caches for instructions, stack data, and heap data. The processor can run in simulation or on an FPGA [4].

In the industry, the ARM7 core is designed as a simple architecture which exhibit predictable timing properties [ARM04]. In fact, it is claimed to be a fully timing compositional architecture. The core is designed such that in the case of a timing accident, the pipeline stalls until the issue is resolved. The analysis of this system becomes simple, and one could analyze each component separately and add penalties when necessary. Similarly, the newer ARM Cortex-R series, targeting for hard real-time applications, provides a higher performing processor while limiting the complexity of the analysis [ARM11].

3 Reduced Instruction Set Computing
4 Field-Programmable Gate Array: a re-programmable integrated circuit

Designing predictable single-cores is not enough to make the multi-core that uses them fully timing compositional. Shared resources, such as memory controllers, are accessed concurrently which may complicate their predictibility. Designer of timing predictable multi-cores already address such issues. For example, the T-CREST [Sch+15] project aims at designing a time predictable multi-core architecture based on Patmos cores. The memory hierarchy and the core-to-core communications are particularly discussed in order to ensure a time-predictable architecture.

In the area of many-core systems, the Tilera Tile processors have received some attention from the real-time community. Such a platform however are not designed for real-time but for high performance or networking. The authors in [Pag+14] describe how to configure the processor in a predictable execution environment. Figure 2.11 gives a global overview of the Tilera TILE-Gx36 processor. It has 36 clusters, each containing a single 1.2 GHz core.

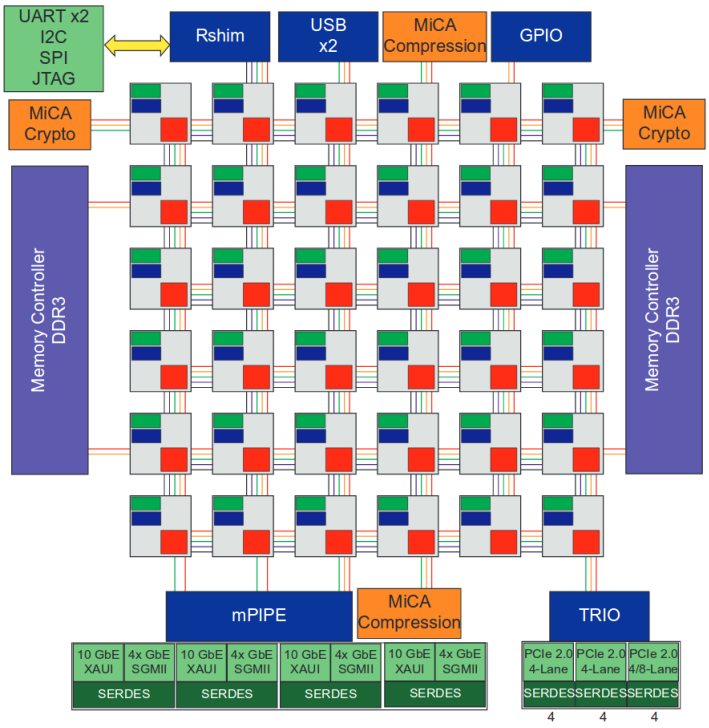


Figure 2.11: Overview of the Tilera TILE-Gx36 processor (taken from [Til12])

The MPPA-256 is another many-core architecture implemented by Kalray [Din+14b]. This processor is composed of 16 *compute clusters* with 16 cores and 2 *I/O clusters* with 4 cores. The clusters are connected through a dual NoC (for data and control) in a 2-D torus topology. Based on a VLIW architecture, the cores (named *k1-core*) are claimed fully timing compositional [Din+14b]. The k1-core is clocked at 800MHz, has an in-order, 5 stages pipeline with no branch prediction. This is the processor considered in this thesis for the experimental evaluation. A detailed description of the Kalray MPPA-256 is given in Chapter 6.

2.4 EXECUTION MODELS

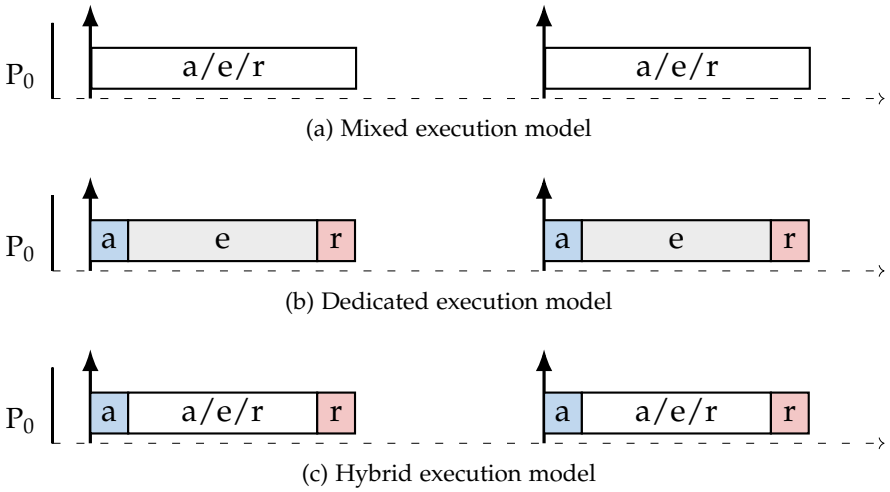


Figure 2.12: Execution models, where: *a*, *e*, and *r* stand for acquisition, execution, and replication respectively

The hardware architecture offers many capabilities that affect the application and their analysis. The interaction between software and the underlying hardware should be considered in the analysis. Some architectures, such as Common-Off-The-Shelf (COTS) processors, can nonetheless be used by enforcing a certain configuration and execution model. This is the case, for example, of the Tilera processor in [Pag+14].

A proposed execution model in [Pel+11] aims at enforcing the predictability in COTS-based systems. The idea is to follow a set of programming guidelines in terms of accesses to shared resources. A dedicated scheduler ensures a temporal isolation to reduce (or eliminate) any unpredictable interference. Note that this might not be enough since some hardware components are not designed for real-time requirements. In this case, extra components might be added to the system to enforce a predictable behavior.

Execution phases can be used to improve the predictability of the system. Figure 2.12a represents the traditional execution model where computations and accesses to the shared resource are mixed. The computations are instructions that do not request any shared resource but can access the cache or a local resource. The shared resource accesses can be explicit, as in the case of communications, or implicit, as in the case of cache misses. In this model, the analysis assumes the worst-case scenario; accesses from co-runner tasks always interfere as long as the tasks are executing at the same time.

According to this behavior, tasks can be split into phases. Figure 2.12b illustrates the different phases. A task starts with an *acquisition phase* where all shared data is copied in a local memory. The *execution phase* accesses, computes, and stores the data locally. Finally, a *replication phase* copies the data back to a shared global memory. The advantage of this model is that some system, with powerful DMA engines[5], can (i) emit bursts of accesses, (ii)

5 Direct Memory Access engine: a hardware feature allowing direct accesses to the main memory independently from the CPU

offload memory transfers from the CPU to the DMA. Thus improving the performance and reducing the overhead of single memory accesses. Moreover, any potential interference from co-runner is limited to the acquisition and the replication phases. This also allows more freedom in scheduling; each phase can be scheduled separately in such a way that the interference on shared resources is avoided [Mel+15; Bec+16; Pel+08].

Figure 2.12c shows a hybrid execution model. Accesses to shared resources are allowed outside of the acquisition and replication phases. This happens in the case where, for example, data can be altered and needs to be reloaded during the execution phase. In this model, the interference on shared resource may occur at any point during the execution of the task. Unlike the general model, a (partial) distribution of accesses across the phases is known and an adequate analysis can use this information to derive a tighter approximation on the interference. The work done in [SCT10] shows how the execution models affect the results of the timing analysis in a shared resource environment.

2.5 STATIC TIMING ANALYSIS

Static timing analysis aims at finding upper-bounds on the WCET as it is illustrated above in Figure 2.2. It is carried out in an input independent matter; the estimated WCET should be absolute and valid for any input values. The analysis is performed on an abstract model of the system. Depending on how much information is available in the model, the analysis can achieve tight estimations but at the cost of complexity.

Figure 2.13 illustrates a common framework for static timing analysis [Wil+08]. As input, the framework takes an executable binary and reconstructs a *Control Flow Graph (CFG)*. A CFG is representation (in the form of graph) of the execution paths in the program's flow. Figure 2.14 illustrates an example of a pseudo-C code and its CFG; each circle in the CFG, called *basic block*, is a set of sequential instructions.

The second step in the timing analysis framework is to annotate the CFG with information that helps to derive timing values. The *value analysis* [Sou+07; The+03] verifies the accessed memory addresses. The *loop bound analysis* extracts bounds on the execution of loops. Note that unbounded loops are avoided in hard real-time systems. Finally, the *Control-flow Analysis* finds infeasible paths in the CFG. These analyses rely on formal methods such as abstract interpretation.

The described analyses rely on the reconstructed CFG from the binary. A dedicated compiler may generate an annotated CFG with the necessary information. It happens, however, that such information, such as variable types, is abstracted by the compiler and optimized. This is usually resolved by performing the analysis on a CFG obtained from an intermediate representation of the source code. The intermediate representation form allows inferring more information on the execution paths, values types, loop bounds,... This CFG is then compared against the one from the executable binary using a *block matching* technique to identify the relevant basic block.

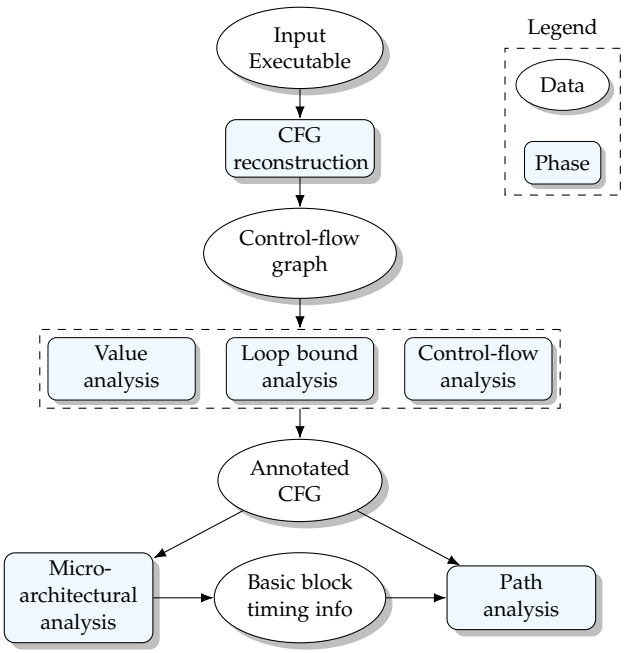


Figure 2.13: Components of a timing analysis framework [Cul+10]

2.5.1 *Micro-architectural Analysis*

The purpose of this analysis is to derive timing bounds on instructions' execution by taking into account the state of the hardware. In fact there are many factors that intervene and affect the execution of an instruction. As a starter, the state of the pipeline and the previously executed instructions may delay (or speedup) the current instruction. This is also correlated with the number of stages in the pipeline and whether there are dependencies between instructions. The analysis of the pipeline derives upper-bounds on the execution of a basic block in the analyzed CFG.

The caches greatly affect the execution time. The commonly used analysis techniques are known as *May*, *Must* analyses [Fer+99]. The former over-approximates the cache content whereas the latter under-approximates it. The combination of the cache analyses allows the annotation of each `store` and `load` instruction with `HIT`, `MISS`, or `UNKNOWN`. The timing information are derived depending whether the accessed data is in the cache or in the main memory. The case of `UNKNOWN` is considered in the analysis as both `MISS` and `HIT`, since it is hard to know which value leads to the worst-case scenario (for instance, due to potential timing anomalies). This inconveniently increases the number of states that must be taken into account. An effort in this area aims at reducing the number of `UNKNOWN`s obtained from the analysis, allowing for tighter estimations and simplifying the analysis [Tou+17].

Another aspect of the micro-architectural analysis is the accesses to shared resources. The time required to access a shared resource, such as the main memory, depends on accesses from co-runners as well as the used arbitration policy. An accurate timing model of such platforms might be costly since it has to take into account all tasks running in the system and accessing the memory. A way to simplify the analysis is to always consider a constant penalty, which might be too pessimistic and sometimes not feasible (for instance, in fixed-

```
#define N 10
int main(){
    /*init*/
    int A[N];
    int cond=0;
    for(int i=0;i<N;i++){ /* b0 */
        if(A[i]) { /* b1 */
            cond += A[i]; /* b2 */
        }else {
            cond =0; /* b3 */
        }
        if(!cond){
            /* b5 */
        }else{
            /* b6 */
        }
        /* b7 */
    }
    /* exit */
}
```

Listing 2.2: Example of C-like code with a loop with two conditions

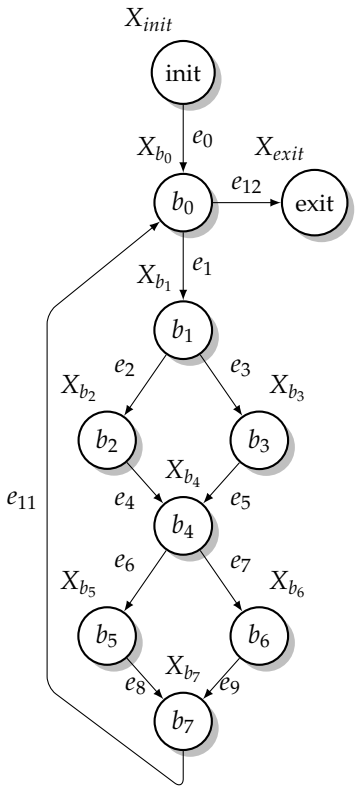


Figure 2.14: Example of a simplified C code and its (simplified) CFG

priority arbitration). Another option is to consider simpler arbitration policies that allow separate analysis of each task. This, however, might not be the case in all systems. There is always a trade-off between the precision and the effort/complexity of the analysis. The interference analysis, which is the focus of this thesis, is discussed in Part ii.

2.5.2 *Path Analysis*

The path analysis strives at finding all execution paths in the CFG. An execution path is a sequence of basic blocks. One of the existing approaches for path analysis is *Implicit Path Enumeration Technique (IPET)* [LM95]. IPET is a method that considers a list of basic blocks in an execution path and their execution counts. The execution time of a path is computed by adding execution times of its basic blocks weighted by their execution count. The WCET is the maximum of all execution times which can be efficiently found using *Integer Linear Programming (ILP)*.

ILP allows the encoding of problems in the form of linear constraints. Thus, the basic blocks are annotated with timing information and modeled in the form of a linear program. A solver is used to optimize an objective function. In this case, the objective function is to maximize the global execution time. We illustrate this in the example given in Figure 2.14. Let $\mathcal{B} = \{\text{init}, b_0, b_1, \ldots, \text{exit}\}$ be the set of basic blocks in the CFG. Let $\{e_1, e_2, e_3, \ldots\}$ be a set of variables representing the number of times edges of the CFG are executed. Similarly,

Let $\mathcal{X} = \{X_b | b \in \mathcal{B}\}$ be the set of variable representing the number of times basic blocks are executed. The linear program representing all the execution paths in the CFG is:

$$\left\{ \begin{array}{l} X_{init} = e_0 = 1,\ X_{exit} = e_{12} = 1 \\ X_{b_0} = e_0 + e_{11} = e_{12} + e_1 \\ X_{b_1} = e_1 = e_3 + e_2 \\ X_{b_2} = e_2 = e_4,\ X_{b_3} = e_3 = e_5 \\ X_{b_4} = e_4 + e_5 = e_6 + e_7 \\ X_{b_5} = e_6 = e_8,\ X_{b_6} = e_7 = e_9 \\ X_{b_7} = e_9 + e_8 = e_{11} \\ e_{11} \leq 10 \end{array} \right\}$$

The formula above states that: (i) the program must be started and exited only once, (ii) each basic block is exited the same number of times as it is executed, (iii) loop bounds analysis adds constraints on the number of time the loop head is executed. Let w_b be the local WCET of basic block b obtained by, for instance, the techniques discussed in Section 2.5.1. The objective function given to the ILP solver is:

$$wcet = \max \Big(\sum_{b \in \mathcal{B}} w_b \times X_b \Big)$$

An infeasible path analysis can be used to add new constraints to the linear systems and refine the WCET. These constraints can be inferred from the semantics; for instance, two edges are incompatible or conflicting thus cannot exist on the same execution path [Ray14].

Other path analysis techniques are based on *Model Checking* with *Timed Automata* or *Satisfiability* Modulo Theory (SMT). We discuss them as related work in Chapter 3.

2.5.3 *Some WCET Tools*

OTAWA [Bal+10] is an open source, academic tool for WCET analysis. It allows the use of different techniques at each level of the analysis (value analysis, loop bound analysis, flow analysis,...) and combines them to generate the integer linear program required in IPET. It also provides the micro-architectural analysis that includes (among others) the instruction caches, the pipeline, and dynamic branch predictors. *OTAWA* targets the analysis at the level of a single-core. Thus, it does not include the interference on shared resource. It can, however, be used to derive local WCETs on tasks (in isolation without interference) and completed with an interference analysis. Another academic tool is *Heptane* [HRP17], a research prototype for WCET analysis based on IPET. It supports several cache replacement policies and cache hierarchies as well as shared caches on multi-core platforms.

The commercial tool *aiT WCET Analyzer* [Wil+08] from *Absint* offers a support for WCET analysis for a broad range of platforms. The tool relies on abstract interpretation for the value analysis and cache/pipeline analyses. It also uses ILP for the path analysis. Although the used methods and techniques may differ, *aiT* and *OTAWA* show the same software architecture and similar to the one in Figure 2.13.

2.6 CONTEXT OF THE THESIS

In this thesis, we address one challenging aspect of multi and many-core platforms; the proposed techniques aims at finding tight upper-bounds on delays of accesses to shared resource. As it was mentioned above, the presence of shared resources creates interference on accesses from co-runner tasks. A conservative approach is to consider a (large enough) constant delay on all accesses to shared resources. This is obviously very pessimistic. The over-approximation increases considerably with the number of accesses, the size of the considered task set, and the number of cores in the system.

2.6.1 *Time Division Multiplexing*

In the real world, two tasks may not interfere at all even if they run concurrently. Their accesses may happen at different instances of time and a very few of them may interfere. Determining which accesses interfere may require a cycle accurate analysis of all the system. This is very costly. Still, in some arbitration policies, such as time multiplexing, tasks can be analyzed separately which considerably reduces the size of the problem.

In this context, we first consider independent tasks running in parallel and accessing a memory bus concurrently. We consider that the bus is arbitrated according to the *time division multiple access (TDMA)* arbitration policy. We show that by combining the semantics of the program under analysis with a model of the bus, we can achieve tighter results than a pessimistic and straightforward approach. Our approach is based on *Satisfiability Modulo Theory (SMT)* to model the shared resource accesses and the program's semantics. More details are given in Chapter 4.

2.6.2 *Response Time Analysis*

We consider a many-core platform with a different arbitration policy than TDMA. Our proposed approach offers a higher scalability, more flexibility and extendability, with tight upper-bounds on WCET. We abstract certain aspects of the task execution and consider only smaller intervals of the accesses' occurrences. The tightness is achieved by taking advantage of a precise consideration of the task model as well as the architecture model.

In this part, we consider dependent task graphs represented as *Directed Acyclic Graphs (DAG)*. In particular, we discuss the case of SDF applications and exploit to their particular properties. We take advantage of the dependencies between tasks to eliminate some interference; two dependent tasks cannot execute at the same time hence they do not interfere on shared resources. Our approach provides a static time-triggered schedule that takes into account the delays due to the interference on shared resources.

The analysis takes advantage of the architecture and the execution models. In this thesis we consider a multi-core architectures with partitioned shared resources. In particular, we consider a system with a partitioned shared memory where each partition (called *memory bank*) is accessed through a dedicated bus arbiter. Such a system corresponds to the commercial many-core Kalray MPPA-256 which is used as an experimentation platform. Our

approach takes advantage of the architecture to deliver a tight estimation on the memory access delays.

More details regarding the application and the hardware models are given in Chapter 5. In Chapter 6, we apply the approach on a concrete platform which is the Kalray MPPA-256.

Chapter 3

RELATED WORK

In this chapter, we introduce the related work to this thesis. First, we give an overview on encountered challenges with multi-core platforms in hard real-time systems. We explain how the presence of shared resources make traditional approaches used with single cores obsolete with multi-cores. We present the different approaches that: (i) enforce architecture and application models to avoid the interference between co-runners, (ii) compute upper-bounds on delays due to shared resource interference, or (iii) combine points (i) and (ii) to optimize the analysis and the execution of the system. Finally, We conclude this chapter by positioning the contribution of this thesis within the state-of-the-art work.

3.1 OVERVIEW ON MANY-CORE PLATFORMS IN HARD REAL-TIME SYSTEMS

Multi-core and many-core platforms become unavoidable in safety-critical and hard real-time systems. This is due to more applications that require more performance but still are subject to constraints such as energy consumption and weight. The shift to multi-cores presents many challenges regarding timing determinism, predictability, or composability [Sai+15]. In fact, the presence of shared resources creates functional interference (such as critical sections) and non-functional interference (due to shared hardware resources) between co-runner tasks.

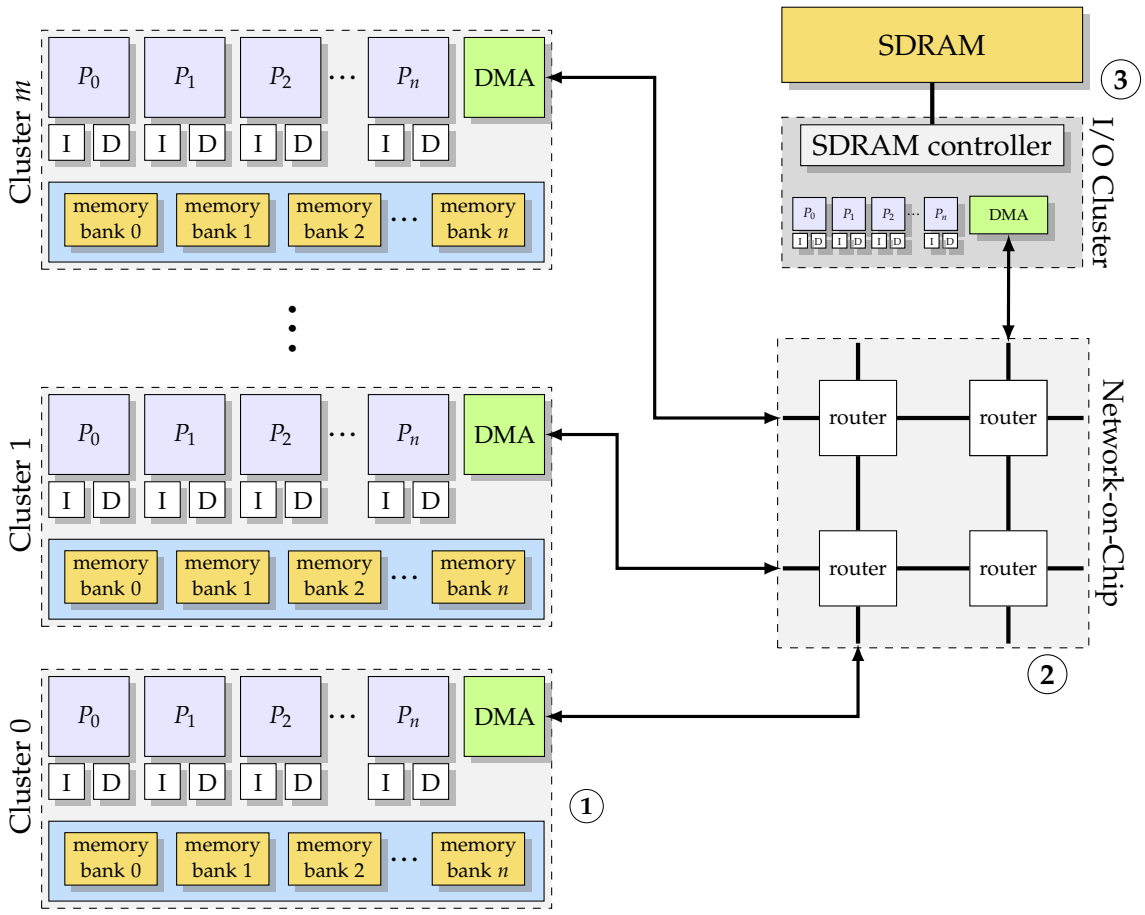


Figure 3.1: Target architecture model of a many-core processor

Figure 3.1 illustrates our target architecture model. This is a typical architecture of a many-core processor [Din+14b; Til12; Sch+15]. Clusters (also called tiles) of cores are connected through a *Network-on-Chip (NoC)*. In each cluster, cores have private caches for data and instruction. A partitioned memory (memory banks) is shared and accessed by all cores. A Direct Memory Access (DMA) engine accesses the shared memory as well as external resources through the NoC. Furthermore, A special cluster (called I/O cluster) connects to an external *Synchronous Dynamic Random Access Memory (SDRAM)*. We identify several sources of interference:

(1) Intra-cluster interference on shared memory banks (and shared buses).

(2) Inter-cluster interference through the NoC.

(3) Interference on the SDRAM.

Considering the sources above, the main challenge is to bound the effects of interference. We present here an overview of the different challenges due to interference on shared resources, to improve the predictability and determinism of multi-cores. In the following we address the problems on each source and present corresponding related work. We also present how the modern architecture and application models are made interference-aware.

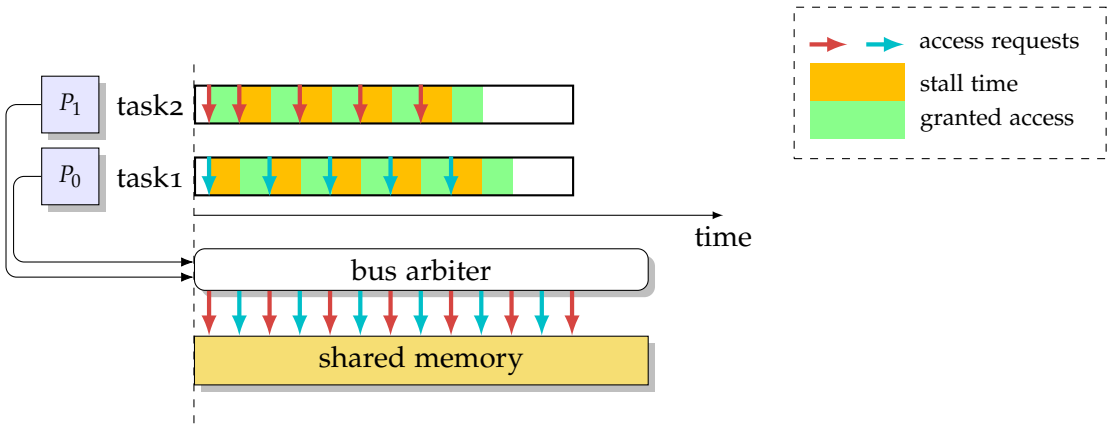


Figure 3.2: Interference on shared memory. Delays occur due to the arbitration of memory accesses which results in a longer execution time for each task than when executing on a single-core

3.1.1 *Shared Resources Interference*

We define *timing interference* as any delay in the execution time suffered by a task due to concurrency on shared hardware resources (e.g., cache memories, shared buses, shared networks, etc.) [Sai+15; Weg17]. In the following we refer to it simply as *interference*. Figure 3.2 illustrates the interference on shared memory. In this example, two tasks running on two different cores access the shared memory at the same time. These concurrent accesses are serialized by an arbiter. The arbiter operates according to a certain policy which directly impacts the execution time of each task by adding extra delays. We define this delay as *interference*. Figure 3.2 shows a *round-robin* arbitration policy.

3.1.1.1 *Intra-Cluster Interference*

Clusters in Figure 3.1 are equivalent to traditional multi-cores and therefore exhibit the same challenges. We identify the following sources of interference: (i) interference on cores due to preemption, (ii) interference on private caches in the case of cache coherency mechanisms, and (iii) interference on shared memories and shared buses. In this thesis, we consider non-preemptive scheduling as well as architectures with private caches and no cache coherency. Thus, we focus on the shared memory and shared buses.

3.1.1.2 *Inter-Cluster Interference*

Inter-cluster communication is performed by means of the NoC. The NoC itself represents a source of interference. It is composed of a set of routers that deliver packets through defined links in the network. The communication route is defined according to a certain packet switching policy (e.g., wormhole switching). This may result in unbounded communication times depending on the switching policy [FF98]. We present the approaches that address the NoC in Section 3.2.

3.1.1.3 *SDRAM Interference*

Finally, SDRAM controllers serialize concurrent accesses to an external memory device, such as the commonly used *Dynamic Random Access Memory (DRAM)*. It constitutes a source of non-determinism which depends on the types of memory accesses (*read* or *write* access) as well as the arbitration policy. Predictable memory controllers are out of the scope of this thesis as we do not address interference on DRAM. Instead, we refer the interested readers to related work addressing predictable hard real time capable memory controllers [Pao+13; Rei+11; AGR07; Pao+09].

3.1.1.4 *Summary*

The interference on shared resources depends not only on co-runners but also on the implemented arbitration policies which affect the timing determinism of multi-cores. A precise analysis of such systems requires a model of all co-running tasks as well as a model of the shared resources, which can be very complex and costly. To solve this challenge, some architectures were designed to provide temporal and spatial isolation between tasks. Predictable bus arbitration schemes are also designed to upper-bound and optimize the memory latency [BRS11; BRS13]. Examples of predictable architectural designs are studied and implemented in the T-CREST project [Sch+15] and the *MERASA* project [Pao+13]. The objective is to partition the shared resources (e.g., partitioned shared memories and separate private cache memories). For instance, the use of *time division multiplexing* arbitration policies allows bounding the inter-task interference. This way, the execution time of shared resource accesses does not depend on co-runners, making it more predictable.

The effort made in predictable architectures improves considerably the scalability and precision of timing analysis techniques. By taking into account the provided hardware capabilities, the timing analysis of multi-cores becomes simpler such that single-core analysis can be easily adapted [Weg17]. Such platforms are designed to improve the worst-case performance rather than the average-case performance by introducing extra delays that guarantee predictable behavior and bounded worst-case delays. In the case where the processor used does not provide mechanics to eliminate the interference, the timing analysis must account for all the delays in the execution time. We present related work to this challenge in Section 3.3.

3.1.2 *Application and Execution Models*

The software can be tailored to take advantage of hardware features in predictable multi-core and many-core systems. For instance by using partitioned shared memory and/or time division multiple access based arbitration policies, the application can follow an execution model that keeps a predictable behavior while optimizing the performance. These functionalities allow deriving a set of guidelines and development rules to bound the effect of interference [Per+16b; Per+16c]. Such guidelines take into account the properties of the target processor to enforce determinism and predictability at the software design level of the application.

The idea of adapting execution models to eliminate the interference was proposed in previous work; for example in [SCT10], tasks are separated into *superblocks*. A superblock is composed of sequential execution phases (e.g, *acquisition*, *execution*, *replication*). The acquisition and replication phases access a global shared memory and copy the data to a local memory such as a private cache memory. This allows the execution phase to run without interference from co-runners. The challenge in such an approach is that the number of superblocks may increase considerably, for example, smaller cache memories limit the size of acquisition and replication phases, and therefore may affect the scalability of the analysis.

The *PRedictable Execution Model (PREM)* [Pel+11] suggests to decouple memory accesses and computations. This allows running concurrent computations in parallel whereas memory phases can be scheduled to run sequentially [Mel+15; Bec+16]. The goal here is to eliminate the interference and therefore to improve the timing analysis of the system. This relies on completely rethinking the programming model of the application. Moreover, for memory intensive applications, this approach may result in a performance drop and/or may result in core under-utilization.

3.1.3 *The Mapping and Scheduling Problem*

One of the main challenges in multi-core systems is the problem of mapping tasks to cores. It consists in allocating tasks to cores and scheduling (ordering) the execution of tasks assigned to the same core. This is similar to the *bin packing* problem [Gar+76] which is known to be NP-complete. Moreover, the mapping and scheduling of tasks must follow some constraints that can be functional (such as deadlines or precedence in a dependency graph) or non-functional (such as resource availability, memory capacity or network bandwidth).

Off-line mapping and scheduling of tasks is widely preferred in safety-critical industrial systems. In this approach, the tasks running on the system are known in advance. Many recent approaches for the mapping problem focus on optimizing the end-to-end response time considering a single conservative WCET that includes the interference delays [YYA16; NHP17; WN15; Gia+17]. The common assumption in these approaches is that the interference on shared resources is negligible. Otherwise, the WCETs can be potentially large and pessimistic which may harm the schedulability test of the system.

Instead of pessimistic upper-bounds on the interference, some mapping and scheduling algorithms aim at completely eliminating the interference on shared resources. This relies on: (i) application models, such as PREM, to schedule memory phase and computation phase separately [Mel+15]; (ii) hardware isolation that relies on partitioned shared resources [PNP13; Bec+16].

An example with partitioned shared resources is the approach of [PNP13] for mapping SDF applications on a many-core processor. The execution model used in this work relies on a spatial isolation of tasks; the shared memory is partitioned and each partition is assigned to a core. Communications are achieved using a *message passing interface* by means of a dedicated buffer. The communications are then scheduled accordingly. This idea is similar to the one in [Bec+16] which aims at completely eliminating the interference on shared memory.

3.1.4 *Summary*

We presented an overview of predictable many-core architectures and related work that addresses them. The presence of shared resources at several layers in the architecture affects the determinism and predictability of such platform due to timing interference. In order to certify these systems, it is necessary to bound the interference delays (as imposed by the DO-178B/C standards) and provide means for spatial and temporal isolation.

Spatial isolation can be achieved by means of partitioned shared and private memories. Temporal isolation can be achieved at the hardware level by means of Time Division Multiple Access (TDMA) arbitration policy of shared resources. This offers a convenient solution to bound the effect of shared resource interference. At the software level, time triggered scheduling policies help to improve the timing analysis of tasks and therefore simplify the mapping algorithms. For other work-conserving policies, the solution seems to either (i) restrict the application at design level to enforce the predictability, or (ii) model the shared arbiters to obtain tight delays on the shared resource accesses. In the following we present the related work to point (i) in Section 3.2 and point (ii) in Section 3.3.

3.2 TEMPORAL ISOLATION: A WAY TO AVOID INTERFERENCE

As mentioned above, single core timing analysis can hardly be adapted in multi-core processors. In single-core, the longest execution path exhibits the WCET. In multi-core systems, several paths are potential candidates to exhibit the WCET depending on the interference on shared resources. A pessimistic but safe solution is to assume that each shared resource access receives the maximum delay. This greatly increases the WCET's over-estimation. Moreover, it is often not possible to obtain upper-bounds on the interference without taking into account concurrent requests to shared resources. For instance, in the case of *fixed-priority* arbiters, it is not possible to bound the interference without taking into account all accesses from higher priority tasks.

We discuss here several approaches to address the issue above. Temporal isolation provides a way to enforce upper-bounds on the delays due to interference. It can be implemented in hardware (by means of TDMA buses) or in software (by means of scheduling policies).

3.2.1 *Time Division Multiplexing*

Time Division Multiplexing (or Time Division Multiple Access) enforces a temporal isolation at the hardware level. This is done by periodically allocating communications slots in the shared bus to each requester (cores or DMA engines) of the shared resource. A bus request is granted only during its allocated slot. Requests that are issued outside of their slots are stalled until the next available slot. This effectively upper-bound the interference delays since slots are fixed and any request is guaranteed to be granted.

The analysis of shared resources accesses with TDMA arbitration can be simplified by considering a safe constant bound on the worst-case delay at each access [Ros+07]. Nevertheless, several approaches were proposed to accurately estimate the waiting time of an ac-

cess for its corresponding communication slot. As an example, *Chattopadhyay et al.* [CRM10] improves the analysis cost of loops by aligning each loop head execution with the TDMA period. A penalty term is added to the WCET of each loop. This allows a better scaling of the analysis at the cost of the precision. The approach by *Kelter et al.* [Kel+14] offers a compromise for loop analysis by modeling the offsets in the TDMA bus with an ILP problem. The proposed solution gives a tighter estimation of the WCET compared to the pessimistic approach. Considering bounded loops, the authors give two methods to estimate the WCET in presence of a TDMA bus with minimal unrolling. The first method unrolls the loop until a fix point of offsets is reached. The second method uses dynamic flow graphs to model loops.

According to the authors, any WCET technique for TDMA buses can be used in this context. We particularly note that the technique from [Kel+14] can also be applied in this case. It relies on abstract interpretation and fixed-point iterations to improve the precision and the scalability of the WCET analysis of TDMA resource arbitration delays.

Schranzhofer et al. [SCT10] propose an efficient analysis of the worst case response time (WCRT) of a shared TDMA bus. The proposed framework uses the access model in periodic tasks to analyze the worst-case response time of the bus and schedulability of tasks. By separating accesses to the bus and computations, this approach exhibits tighter bounds and reduces the WCRT.

Other research works were done to improve the WCET estimation with a shared bus. *Gustavsson et al.* [Gus+10] use timed automata to model the software and the hardware. An upper bound on the clock of the timed automata is obtained with model checking tools such as UPPAAL [BDL04]. This approach suffers from a potential explosion in the number of automata's states. *Lv et al.* [Lv+10] propose a better use of timed automata. With an abstract interpretation of the cache, basic blocks in the CFG are annotated with *cache miss* and *cache hit*. The annotated CFG is then modeled with a timed automaton. TDMA arbitration policy of the shared bus is also modeled with an automaton. The results show a better estimation on the WCET compared with the pessimistic approaches.

To address the scalability issue in timed automata, some approaches are based or inspired from the *Real-Time Calculus* [SCT10; Sch+10; Sch+08]. Such approaches allow analyzing any arbitration policy, including TDMA, given upper-bounds and lower-bounds on the number of shared resource accesses during a time window, expressed in the form of *arrival curves*. The work in [DNA15] follows the same idea. We will detail it in Section 3.3, since it also targets non-TDMA buses.

3.2.1.1 *TDMA Optimizations*

TDMA based arbitration policy offers a predictable behavior of the shared resource interference. It also improves and simplifies the WCET analysis. This comes at the cost of performance. In fact, TDMA policy is *non-work conserving*, meaning that the bus still stalls even when there is no concurrent accesses. In some applications, this may harm the performance, especially in very large systems. For this reason, some approaches were proposed to optimize the TDMA arbitration to improve the worst-case behavior.

Rosèn et al. [Ros+07] propose a system-level scheduling with WCET analysis. The approach considers a task graph with a fixed mapping, and an initial TDMA bus scheduling.

The result is a static periodic bus schedule, i.e., a schedule table of the slots in a bus's periodic cycle. First, tasks are assigned to cores according to a list scheduling policy that uses initial WCETs; through this, one also obtains the critical path in the task graph. Then, the bus schedule is modified to optimize the latest termination time in the critical path. Finally, the new bus schedule is used to re-schedule tasks and re-compute the critical path that includes the bus delays. Each WCET computation (except the first one) uses an ILP system that encodes accesses and availabilities of the bus schedule as well. Since the WCET analysis gives new timing information, the optimization must run again. The process terminates when no shorter critical path is found.

Li et al. [LMS15] also proposes an application specific TDMA schedule of the shared bus. As in [Ros+07], it assumes a programmable TDMA bus to optimize the WCET of shared resource accesses. In contrast to these approaches, *Oehlert et al.* [OLF17] propose an optimization of WCET in a multi-core with TDMA bus, without reprogramming the bus. The target platform contains a shared flash memory and faster but smaller private scratchpad memories[1]. The key idea is to statically allocate some instructions to the size-limited scratchpad memory based on an ILP model to improve the overall WCET.

3.2.2 *Time Frame Isolation*

TDMA-based shared buses offer temporal isolation in the hardware. On processors that do not implement such policy, it is possible to enforce the temporal isolation at the software level, by relying on specific scheduling techniques and/or following specific guideline on the execution models [Per+16b]. We present here the main ideas that follow this approach.

The paper [Gia+16] proposes: (i) a flexible time triggered scheduling, (ii) a response time analysis for the proposed scheduling policy that accounts for the interference, (iii) an optimization of the resource utilization. The analyzed application is a multi-periodic task graph with mixed-criticality levels. The analysis is performed on a *hyper-period* (the least common multiple of the tasks' periods) obtained by unfolding the tasks' execution. Each hyper-period cycle is divided into fixed size frames. Each frame is divided into sub-frames where tasks are assigned to. Sub-frames are synchronized through barriers, i.e., cores wait at the barrier until all tasks in the current sub-frame finish. Therefore, the length of a sub-frame is equal to the maximum worst-case response time of the tasks running in the sub-frame. The interference on shared memory accesses is pessimistically estimated; all tasks executing in the same time sub-frame and accessing the same memory bank interfere with each other. This depends on the task-to-core and data-to-memory bank mappings.

The *mapping design optimization* phase updates the task and data mappings such that: (i) the workload is balanced between the cores and the memory banks, (ii) dependencies and deadlines are respected. The algorithm stops when it finds a solution (i.e., task and data mappings) that minimizes the sum of the time frames' lengths (i.e., barriers' sizes).

Carle et al. [Car+15] describe the design and implementation of data-flow programs on multiprocessors. The authors focus on the optimization of an off-line schedule taking into account timing properties such as release dates and deadlines. In this work, time frames are constructed to enforce temporal isolation between tasks allocated to different frames. The

1 A high-speed memory

tasks that are allocated to the same frame are, however, subject to interference. The authors then assume a conservative upper-bound on WCET that accounts for all delays to access shared resources by tasks in the same time frame. The interference becomes proportional to the number of tasks per time frame.

3.2.3 *Summary*

The techniques discussed above approach the shared resource problem such that the interference is completely eliminated or easily bounded with a constant delay through temporal isolation. This is done by means of the hardware (TDMA bus arbiters) or the software (time frame based scheduling). In the latter, tasks are allocated to time frames (implemented with synchronization barriers) taking into account different constraints such as deadlines, dependencies, and accessed memory banks. The interference on shared resources can be completely avoided [Man+17] or at most bounded by a pessimistic but safe delay either constant [Car+15] or proportional to the number of tasks allocated to the same time frame and accessing the same shared resources [Gia+16].

Other approaches tackle the problem without imposing a global synchronization mechanism to completely eliminate the interference. Instead, the shared resources and their arbiters are modeled and taken into account in the timing analysis of the system. This results in more accurate bounds on shared resource access delay. We present such approaches in the following section.

3.3 SHARED RESOURCES INTERFERENCE ANALYSIS

Temporal isolation is a solution to eliminate or bound the effect of timing interference. Other approaches tackle this challenge by directly analyzing the shared resources to find an upper-bound with tight over-estimation. We present here a set of related work that target shared resource analysis in multi-core and many-core systems.

3.3.1 *Formal Approaches*

Model checking with abstract interpretation is popular in static timing analysis. *Kelter and Marwedel* [KM17] rely on timed automata to model all the concurrent tasks in the program. A parallel execution graph is constructed from the different execution paths of each task's CFG. This allows to precisely detect the interference at a cyclic level and according to the arbitration policy. Despite the optimizations to reduce the size of the analyzed graph, the analysis is vulnerable, in terms of run-time, to the size of the programs as well as the number of cores in the system. Although model-checking based approaches with abstract interpretation yield accurate results, it requires a lot of effort to model the system under analysis. Furthermore, the computational complexity harms the scalability of such approaches especially on architectures with tens or hundreds of cores.

Model-checking based approaches can be combined with Real-Time Calculus in a hybrid approach to improve the scalability of the analysis [LPT09]. Based on this idea, *Lampka et al.* [Lam+14] propose an approach that uses timed automata and abstract interpretation to

model the programs. Several shared bus arbiters (*first-come-first-served*, *round-robin*, TDMA) are modeled with timed automata. The analysis of shared resource interference uses Real-Time Calculus to derive the arrival curves for access requests and the availability curves for the shared resources (in this case the shared bus). This approach also exploits the idea of phase-structured task models [Pel+10; SCT10] to deliver tight WCRT estimation. Although modeling a more complex arbiter can be feasible in timed automata, it increases the complexity of the model and may lead to an explosion of states during analysis. Architectures such as the Kalray MPPA-256 have several shared resources. This adds to the complexity of the analysis and may significantly affect its scalability.

Dasari et al. [DNA15] present an approach for response time analysis taking into account interference on the bus. The number of accesses is obtained from measurements during the task's execution The bus itself is modeled by considering the earliest and latest available communication slots for the task under analysis. This representation depends on the arbitration policy of the bus. The authors give mathematical models of the most widely used bus arbiters; however, it is difficult to see how to represent with this approach less conventional arbitration policies available in commercial platforms [Din+14b].

Giannopoulou et al. [Gia+16] propose a response time analysis in mixed-criticality scheduling on a clustered many-core that fits the architecture model in Figure 3.1. This approach considers a "flexible time-triggered scheduling" model which divides time into frames, and forces a global synchronization barrier between frames. Optimizations were proposed to maximize core utilization and to reduce the interference by mapping cores to memory banks accordingly. Nevertheless, global synchronization potentially creates core under-utilization while they wait for the barrier.

The authors in [CKH16] propose a conservative modelling technique of the shared resource contention based on the event streaming model. The approach targets dependent tasks assuming a fixed task mapping. A tighter upper bound on shared resource requests is computed by considering a cluster of tasks (constructed in the analysis) as a whole instead of considering tasks individually. The task clusters are constructed based on each task's timing information as well as its dependency graph. Bounds on the shared resources' requests are used in the WCRT analysis (such as the one in [Kim+13]) which in turn updates the timing information. The process stops when the task set is stable, i.e., the release and finish dates are fixed. The authors claim that this process converges toward a fixed point, however, no proof of convergence is discussed.

Altmeyer et al. [Alt+15] presents a multi-core response time analysis that accounts for the interference on shared resources in a compositional manner. The idea is to consider a worst-case response time in isolation and add up all the delays from potential sources of interference. This approach is more flexible to extend to a large set of architecture components given an accurate model of the arbitration policies. The genericity of such approach comes at the cost of an over-approximation (pessimism) in the resulting response times.

Note that the approaches in [CKH16; DNA15; Alt+15] rely on known task profiles; WCET in isolation and upper bounds on shared resource accesses. Such profiles can be obtained from a static timing analysis tool of tasks in isolation. There exist many tools targeting different processors, such as: OTAWA [Bal+10], aiT [Ait], Heptane [HRP17], SWEET [Swe],

and Chronos [Chr]. Another way to obtain task profiles is from measurement-based techniques, such as in [DNA15]. We discuss in the following an alternative approach for interference analysis based on probabilistic and full measurement approaches.

3.3.2 *Measurement-Based Approaches*

Measurement-based approaches are an alternative used in timing analysis. *Measurement-Based Timing Analysis (MBTA)* does not require models of the underlying architecture or the application. Instead, the real application and processor are directly used and the timing information is collected through different approaches, such as hardware performance counters and/or code instrumentation. This makes the analysis much simpler and more scalable when compared with formal approaches.

End-to-end measurements, however, do not guarantee that the worst-case execution time is observed. The approaches rely on *Extreme Value Theory*[2] to infer a probabilistic WCET [CG+12; SGM17]. Another way is to measure portions of codes that can be performed and combined to obtain a probabilistic WCET. It is easier to trigger a worst-case behavior on a fine-grained region in the application, however, this requires a heavy code instrumentation and other code tracing methods which can be too intrusive in software execution.

Measurement-based approaches of shared resources is still a challenging problem. The interference depends on co-runners and therefore it is hard to measure it. MBTA can benefit from hardware partitioning and interference-aware mapping/scheduling policy to perform without accounting for interference. In cases where these solutions are not possible, a probabilistic worst-case interference can be added in a compositional way to each task. This has been used, for instance in [Pan+15], to enable measurement-based analysis for TDMA buses. The European project *PROXIMA* [Pro] aims at enabling MBTA for industrial applications on multi-core and many-core processors.

3.3.3 *Summary*

Shared resources in multi-core and many-core systems create timing interference. For this reason, traditional two-step approaches (first timing analysis, then schedulability analysis) cannot perform as efficiently as on single-core. Several approaches aim at addressing the shared resource interference challenge by considering different aspects, such as: shared caches, shared buses, and shared memory banks.

We presented a set of formal methods that address this problem. Although, the result is a tight WCET estimate, the analysis tend to be very complex and sometimes non-scalable. MBTA on the other hand is an alternative to formal methods that relies on probabilitic approaches to estimate the WCET. The extension of MBTA for shared resources is still very challenging and may result in large over-estimation.

Hybrid methods that rely on formal methods and MBTA represent a good compromise between tightness and scalability of the analysis. It is easier to trigger the worst-case be-

2 Statistical theory for extreme events of a stochastic process

havior on a small portion of the analyzed program. MBTA is used on these portions to infer metrics such as local WCETs or worst-case number of memory accesses [Bal+17]. The results are then combined by a formal method to induce the whole system's WCET. This is the case, for example, of the approach in [DNA15] where task profiles (execution time and number of accesses) are measured for code regions in the program under analysis. In the context of the European project P-SOCRATES [Pso], *Nélis et al.* [NYP17] propose an approach for mapping and scheduling tasks on a many-core processor. It relies on timing information based on measurements in isolation and under heavy contention of each task, where the mapping and scheduling is done by a dedicated formal tool.

3.4 CONCLUSION AND POSITIONING

We described in this chapter the impact of the shared resources on multi-core timing analysis. We described the different directions and approaches that were taken to tackle this challenge in real-time systems. Predictable architectures were designed with the idea of reducing the analysis complexity. We discussed some noteworthy related work with solutions that benefit from the hardware-level features (such as the use of TDMA arbiters) or software-level features (such as interference-aware scheduling and mapping).

In the context of the highlighted related work, the contributions of this thesis turn around two topics: (i) a formal method to analyze TDMA buses, (ii) a response time analysis with non-TDMA shared arbiters. While the former topic can be applied to any multi-core with a shared TDMA bus, the latter topic is particularly focused on a commercial many-core that fits our target architecture model in Figure 3.1 (page 30). The following is the positioning of the work presented in this thesis.

3.4.1 *On TDMA-based Buses*

TDMA buses show good characteristics to improve the worst-case behavior of accesses to a shared resource. We discussed several related work that target such arbitration policy and give a tight estimate of the WCET. Such approaches, however, consider already known feasible paths obtained from the semantics [Kel+14; CRM10]. In Chapter 4, we present an approach that combines the infeasible path analysis and the micro-architectural (shared TDMA bus) analysis in the same step. Instead of the usual ILP-based methods, our approach is based on (a more flexible) *Satisfiability Modulo Theory (SMT)* model which allows considering all feasible execution paths without having to enumerate them. We give an overview on SMT in Chapter 4, Section 4.2.3.

3.4.2 *On Shared Resources Interference*

Methods and tools for the design and implementation of real-time systems on single-core architectures are well established. On such architectures, the method used is a two-step process: a timing analysis derives the WCET, which is then used in a scheduling analysis. The only shared resource (and source of interference) here is the processing element. These two steps are independent and hence can be achieved separately.

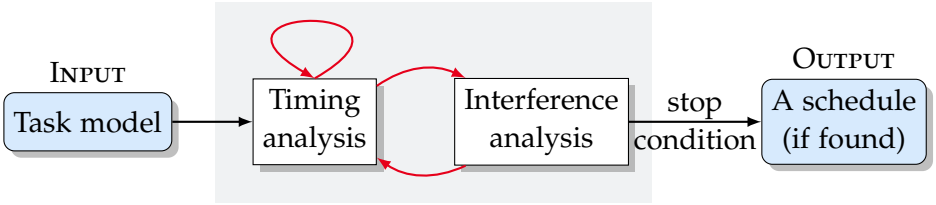


Figure 3.3: Global scheme of multi-core response time analysis

The emergence of multi-core systems brings up new challenges; the boundaries between the traditional two-step analysis are not well-defined. The timing analysis of tasks depends on the interference between cores, which in turns depends on timing information of each task. These cyclic dependencies create loops between the different phases of the design and implementation of real-time systems. Figure 3.3 represents the general scheme of what is observed in the state-of-the-art [Gia+16; Car+15; Ros+07; CKH16]. The *timing analysis* box encompasses the WCET analysis, WCRT analysis, or scheduling analysis that takes into account at least one source of interference. The *interference analysis* box is an iterative process which leads to a more and more precise timing bound for one or more sources of interference. The considered shared resources that lead to interference can be cores, shared memories, or shared bus arbiters. Note that it is necessary to define a stop condition that ensures the termination of the process.

The second contribution of this thesis in Chapter 5 follows a similar flow as in Figure 3.3 to analyze SDF applications on a many-core processor. We assume a given task mapping, and an execution order. The considered shared resources are bus arbiters to a partitioned shared memory. Our proposed algorithm reaches its fixed point (and therefore exits its loop) when the release dates and response times of each task are stable. The result is a static schedule of tasks (fixed release dates) that respects task dependencies and accounts for delays on shared resource accesses.

We address intra-cluster interference on shared bus arbiters. This complements the work in [Per+16a] that, based on ILP, addresses task and data mapping on a many-core taking into account the inter-cluster NoC. Similarly, our approach also may work with [Ten+14] that uses SMT.

Part II

CONTRIBUTIONS

Chapter 4

SHARED RESOURCES WITH A TDMA BUS

We propose in this chapter an approach for interference analysis of accesses to shared resources based on *Satisfiability Modulo Theory (SMT)*. Our proposed techniques is applied to a shared bus with the *Time Division Multiple Access (TDMA)* arbitration policy.

This chapter is organized as follows: in Section 4.1, we give the motivation of this work and summarize our contributions. In Section 4.2, we give a background on TDMA buses and SMT which are core concepts of our approach. Our contribution is in Section 4.3 which explains how a program with accesses to a shared bus with a TDMA arbiter is modeled using SMT expressions. In Section 4.4, we evaluate our model using micro-benchmarks, then we apply our approach to benchmarks taken from real-life applications. Finally, the conclusion and future work are given in Section 4.5.

4.1 MOTIVATION

Determining Worst-Case Execution Times (WCET) has been the focus of research in the field of embedded systems. Static analysis methods have been developed to provide safe bounds on the WCET. The challenge remains in improving the pessimistic approaches that over-estimate the execution time of the analyzed program as well as the scalability of the

analysis. An example of such an approach is the Implicit Path Enumeration Technique (IPET). The initial version of this approach implicitly enumerates all paths including some "obvious" infeasible paths in a program, leading to an over-estimation on the WCET. To illustrate this, we use the following example:

Example of mutually exclusive paths

```
load ...                                    (1)
... /* 3 cycles */                          (2)
if cond then
    ... /* 5 cycles */                      (3)
end if
if ¬cond then
    ... /* 1 cycles */                      (4)
end if
store ...                                   (5)
```

Example 3. *Simple IPET without infeasible path analysis of the program above gives the longest path {(1), (2), (3), (4), (5)}. However, (3) and (4) are mutually exclusive i.e., they cannot be part of the same execution path. The longest path in this case is {(1), (2), (3), (5)}.*

To avoid the problem above, existing work such as [Ray+15; Ray14; Gus+06] extend IPET to exclude infeasible paths. In this chapter, we use another approach with SMT to encode the program's semantics and to perform the feasible path analysis.

In a multi-core environment, the longest path does not necessarily imply the worst-case execution time. The access time to the shared resource can vary depending on the arbitration policy of the shared bus and the access patterns in the execution path of the program.

Example 4. *In the example above, the instruction* `store` *at (5) can access the bus at different instants depending on whether the code at (3) or at (4) is executed. This results in a variation of the bus access delay depending on the arbitration policy of the shared bus; in this case, the worst-case path* $\{(1), (2), (3), (5)\}$ *does not necessarily lead to the worst-case execution time.*

An analysis that does not consider the semantics together with the micro-architecture analysis would have to analyze the WCET of the accesses to shared resources with reduced information about the possible instants when they occur, and may therefore overestimate their execution time. We address this aspect in our proposed approach.

We assume a *Fully Timing Compositional* system architecture [Rei+06] that does not exhibit timing anomalies (see Section 2.3.2 on page 17). We do not support unbounded loops or unbounded recursion: the analysis has to be able to unroll all loops and inline function calls. Note that this is a common restriction for programs where a formal WCET analysis is applied [Kim+14; DNA15; Gia+16; Kel+13; SCT10; Gia+14; Bec+16]. Our implementation is currently a proof of concept that shows the feasibility of the approach. It makes simplifying assumptions: (i) we assume the absence of cache memory which means that each *load* or *store* instruction issues an access to the shared bus, and (ii) we consider that each instruction takes 1 cycle. As future work, we intend to incorporate static cache and timing

analysis tools, such as OTAWA [Bal+10], to include realistic execution time bounds for the instructions and to model the behavior of local caches.

To sum up, our contribution is a way to encode both the semantics of the program and a TDMA arbitration policy in a single SMT expression, and to use it to compute a safe bound on the WCET of the program. The encoding is carefully optimized to avoid the performance issues of a naive encoding.

4.2 FOUNDATIONS

Multi-core platforms offer capabilities that respond to the growing performance demands of embedded real-time systems. However, predictability of these architectures remains a challenge. Shared resources represent the main hot topic in the predictability of such systems. In this work we are interested in shared buses with Time Division Multiple Access arbitration policy.

4.2.1 *Time Division Multiple Accesses (TDMA)*

Time Division Multiple Access (TDMA) is an arbitration policy for shared buses. It allows cores to share a bus by dividing the accesses into periodic time slots. Cores may receive different slot lengths or different number of slots in a period which gives more or less priority to some cores over the others. For simplicity, we consider in our work a TDMA policy where (i) all slots are of the same length, (ii) each core receives one slot per period. The approach we propose below can be extended to any configuration of TDMA.

We introduce the following notation: The TDMA period is denoted with π. The period is divided into slots of length σ. The *access time*, i.e., the time required to execute a granted request, is denoted by acc. We denote the offset of a request with regard to the start of the TDMA period as off. Given the absolute time t_{req} when the request is issued, the offset off is:

$$off = t_{\text{req}} \mod \pi \tag{4.1}$$

Expressing the timing of the bus in terms of offsets simplifies the bus model since the only possible values of the offsets are in $[0, \pi[$.

A request is granted immediately, only if the offset at the issue instant falls into the communication slot, otherwise it is delayed until the next slot of the core. Once granted, an access cannot be preempted. Therefore, (i) slots must be large enough to execute an access, i.e., $\sigma > acc$, (ii) an access request is granted only if the remaining time in the slot is sufficient, i.e., $off \leq \sigma - acc$. Requests are stalled until the next period if they cannot finish during the slot. According to this definition, the *execution time* T of a request is given by:

$$T = \begin{cases} acc & \text{if } off \leq \sigma - acc \\ \pi - off + acc & \text{otherwise} \end{cases} \tag{4.2}$$

The best case delay is when the request is issued during the slot and granted directly. In the worst-case, the request is issued when there is not enough remaining time to process it. Hence, the execution delay of a bus access varies between $[acc, \pi - (\sigma - acc)[$.

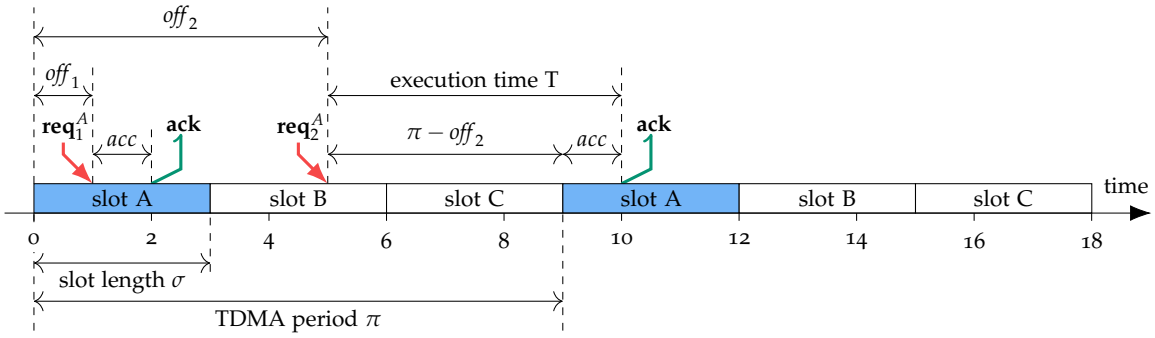


Figure 4.1: Example of a TDMA bus arbiter. Slots $\{A, B, C\}$ are assigned to cores $\{P_0, P_1, P_2\}$ respectively. This example illustrates access requests from P_0.

The following example explains the notions above.

Example 5. *Figure 4.1 illustrates an example of a TDMA bus with a period $\pi = 9$ divided into 3 equal slots $\{A, B, C\}$. Each slot has a length $\sigma = 3$. Slots $\{A, B, C\}$ are assigned to cores $\{P_0, P_1, P_2\}$ respectively. The access time of a granted request is $acc = 1$. req_1 is a bus request issued by core P_0 at instant $t = 1$. This corresponds to the offset $off_1 = 1$ which falls into its associated communication slot. This request is granted and executed with an execution time $T = acc = 1$. A second request req_2 from P_0 is issued at $off_2 = 5$ which falls outside the associated communication slot. It is, thus, stalled until the next slot of core A. The execution time of req_2 in this case is $T = 9 - 5 + 1 = 5$ (Equation 4.2).*

The execution time T of a given request varies depending on whether the request's offset falls into the associated communication slot. One of the challenges in the WCET analysis is to accurately estimate T. A program running on an architecture with a TDMA bus, can start at any offset in the TDMA period. A precise WCET analysis considers exact values of the offsets throughout the program under analysis. This is complex, costly, and sometimes not achievable. As a solution, it is possible to abstract the offsets with sets or intervals of possible values. This results in a less precise but more scalable analysis such as the work presented in [Kel+14]. A pessimistic analysis considers offsets that just miss their slots for all accesses; this corresponds to the constant worst-case delay $\pi - (\sigma - acc)$.

4.2.2 *WCET Analysis of TDMA Buses: an Example*

To illustrate the timing behavior of a TDMA bus, consider Algorithm 1. It is a simple example with two conditional statements. Reading and writing to the memory address pointed by $*x$ is done through `load` and `store` instructions that request the shared bus.

We consider each instruction to be executed in 1 processor cycle and assume that *load* and *store* instructions access the shared bus. The shared bus has a TDMA period $\pi = 6$ processor cycles, and a slot length $\sigma = 2$ processor cycles. The slot associated to the core where the analyzed program is running is $[0, 2]$. Once an access is granted, it is executed in 1 processor cycle.

Taking into account the parameters of the bus, a request emitted at offsets 0 or 1 is granted directly. Otherwise, it is suspended until the next slot. The Control-Flow Graph

Algorithm 1 Example of a C-like code fragment with bus accesses.

```
1:  function EXAMPLE(*x, y, flag)
2:      if y < 0 then
3:          *x ← *x + 1
4:      else
5:          flag ← flag + 10
6:      end if
7:      if y ≥ 0 then
8:          *x ← *x + 2
9:      end if
10:     return flag
11: end function
```

(CFG) in Figure 4.2b shows two feasible paths: the first path ($y < 0$) {b1, b2, b3, b5, b8} and the second path ($y \geq 0$) {b1, b4, b5, b6, b7, b8}. Figure 4.2a shows both feasible paths and their execution times. We suppose that the program starts at instant $t = 0$ and offset $\mathit{off} = 0$. In the case of the first execution path, block b2 emits an access request, (*load* instruction) at offset $\mathit{off} = 2$. This request is delayed until the next slot. The execution time of this path is 18 processor cycles. The second execution time has 15 processor cycles. Thus, the worst-case execution time of Algorithm 1 is $\max(15, 18) = 18$ processor cycles.

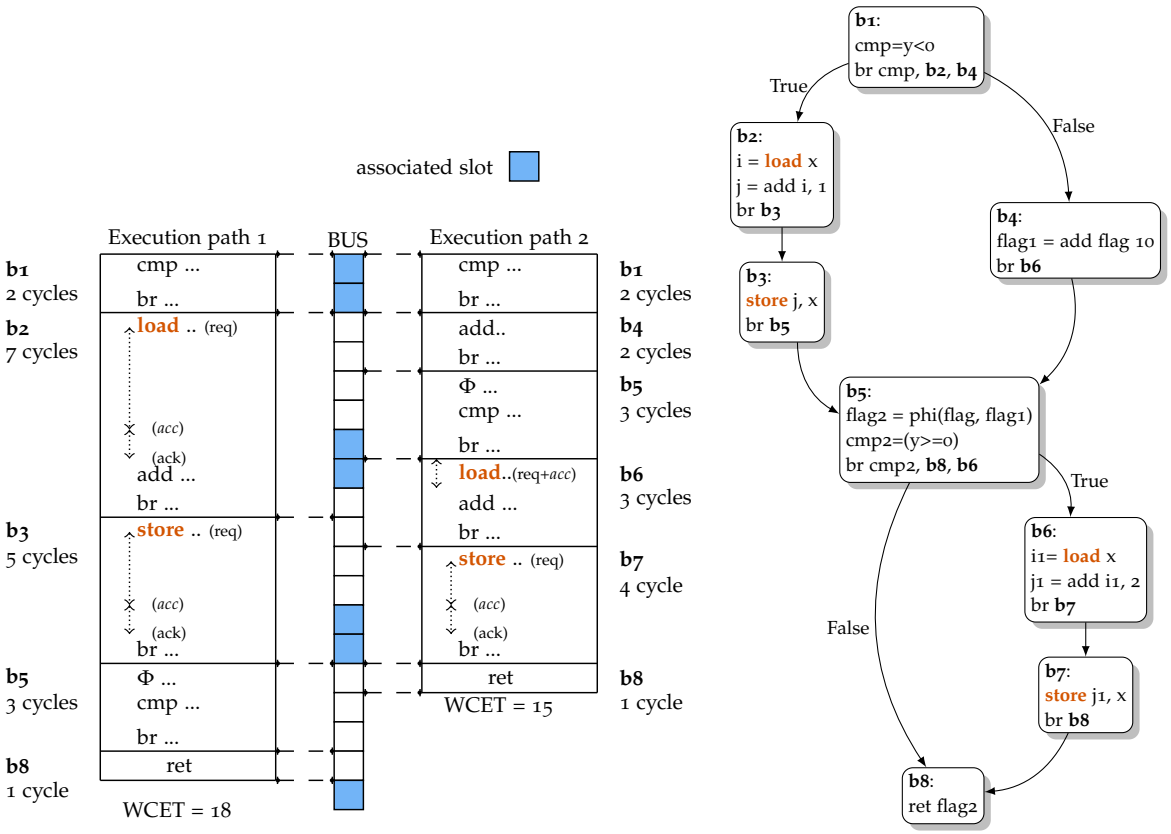
A micro-architectural analysis that does not account for the semantics of the program would have to consider the infeasible path {b1, b2, b3, b5, b6, b7, b8} when considering the *load x* instruction in block b6. In this path, the access request from the *load* instruction is issued at offset $\mathit{off} = 5$, hence not in the TDMA slot. Considering this path results in a WCET=27 cycles. As opposed to this, relying on a feasible path analysis helps to prove that the access in block b6 is in the TDMA slot, and gives a tighter WCET.

As shown above, it is important to consider an approach that combines the offset analysis and the feasible path analysis altogether to obtain a tight and correct WCET estimation. Our contribution is based on *Satisfiability Modulo Theory* to perform the feasible paths analysis combined with the TDMA offsets analysis. In the following, we give an overview of Satisfiability Modulo Theory and explain how it is applied in static timing analysis.

4.2.3 *Satisfiability Modulo Theory (SMT)*

Satisfiability Modulo Theory (SMT) is an extension to the satisfiability problem (SAT) in Bounded Model Checking [Bie+03]. In the SAT approach, a problem is satisfiable if an interpretation (a set of Boolean values), that satisfies the corresponding Boolean formula, exists. SMT extends SAT with a background theory $\mathcal{T}$. Examples are the theory of *linear integer arithmetic (LIA)* and the theory of *linear rational arithmetic (LRA)*.

SMT is expressed with *first-order logic formulas*. First Order Logic is a formalism that expresses a system (or a problem) in terms of variables (e.g. x, y), predicates (e.g., $(x < y)$), connectives ($\wedge, \vee, \neg$), and quantifiers ($\forall$, $\exists$). In our case, the SMT formula is in the Conjunctive Normal Form (or Clausal Normal Form, CNF), i.e., the formula is a conjunction



(a) WCET of Algorithm 1 with a shared TDMA bus with period $\pi = 6$ and $\sigma = 2$

(b) Control-flow graph of Algorithm 1

Figure 4.2: Example of execution paths with a shared TDMA bus

of clauses; a clause is a disjunction of atoms; an atom is an indivisible expression of a theory. Theory atoms are expressed in the form $a_0x_0 + a_1x_1 + \ldots + a_nx_n \bowtie C$, where: (i) $a_0, a_1, \ldots a_n, C$ are integers, (ii) $\bowtie$ is a relation operator in $\{=, \neq, <, >, \leq, \geq\}$, and (iii) $x_0, x_1, \ldots, x_n$ are variables. For LIA, $x_0, x_1, \ldots, x_n$ can be, for instance, in $\mathbb{Z}$.

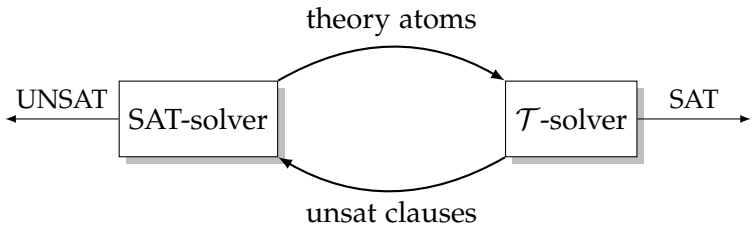


Figure 4.3: General DPLL($\mathcal{T}$) framework

An SMT-solver is a tool that verifies the satisfiability of an SMT formula. Examples of such tools are CVC4 [Bar+11], MathSAT [Bru+08], YICES [DDM06], and Z3 [DMB08]. Most

solvers are based on the DPLL($\mathcal{T}$)[1] method. Figure 4.3 illustrates the general framework used in the SMT-solvers based on DPPL($\mathcal{T}$). The framework is composed of a SAT-solver and a $\mathcal{T}$-solver. The SAT-solver operates on Boolean atoms, representing the theory atoms, to find a satisfiable solution. The $\mathcal{T}$-solver verifies the consistency of the solution with regard to the theory $\mathcal{T}$. In the case of inconsistency, complementary clauses are added to the SAT-solver as a constraint that excludes the inconsistent solution. We illustrate this execution in Example 6. Note that SMT is an NP-complete problem.

Example 6. *SMT-solving steps of a simple SMT formula in $\mathbb{Z}$*

$$\underbrace{\forall x, y \in \mathbb{Z}}_{\textit{LIA theory}} : \overbrace{\big(\underbrace{(x < -12)}_{\textit{atom}} \vee (y > x + 10)\big)}^{\textit{clause}} \wedge \overbrace{(x > 0)}^{\textit{clause}}$$

Theory: $\mathcal{T} = \mathbb{Z}$

(1) *Clauses:* $\big(\underbrace{(x < -12)}_{\textit{Boolean a}} \vee \underbrace{(y > x + 10)}_{\textit{Boolean b}}\big) \wedge \underbrace{(x > 0)}_{\textit{Boolean c}} \rightarrow (a \vee b) \wedge c$

(2) *SAT-solver:* $a = true, b = true, c = true$

(3) *$\mathcal{T}$-solver:* $(x < -12) \wedge (x > 0)$ *conflict*

(4) *New clauses:* $(a \vee b) \wedge c \wedge (\neg a \vee \neg c)$

(5) *SAT-solver:* $a = false, b = true, c = true$

(6) *$\mathcal{T}$-solver:* $x = 1, y = 12 \rightarrow$ *SAT*

(1) *Theory atoms are represented with Booleans a, b, c. This results in the first order logic formula $(a \vee b) \wedge c$.* (2) *The SAT-solver finds a solution that satisfies the Boolean formula where all atoms are* `true`. (3) *The $\mathcal{T}$-solver interprets the atoms in the $\mathbb{Z}$ theory.* (4) *The proposed solution with $(a =$ `true` $\wedge\, c =$ `true`$)$ is inconsistent in $\mathbb{Z}$ because $x < -12$ AND $x > 0$ are conflicting. Therefore, the clause$(\neg a \vee \neg c)$ is added to remove the inconsistency.* (5) *With the new clause, the SAT-solver suggests another solution.* (6) *The $\mathcal{T}$-solver finds consistent values in $\mathbb{Z}$ and returns SAT.*

The examples with SMT expressions given in this chapter are expressed in pseudo-code. In our experiments, we use SMT-LIBv2 [Dav13] which provides standard descriptions of background theories used in SMT systems.

4.2.4 *WCET by SMT*

Henry et al. [Hen+14] demonstrate how to estimate the worst-case execution time using *Bounded Model Checking*. SMT expressions are generated to encode the analyzed program

1 Davis–Putnam–Logemann–Loveland algorithm, used to decide the satisfiability of CNF propositional logic formulas [DP60]

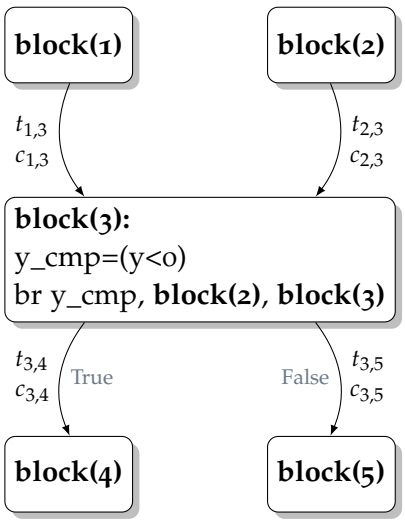


Figure 4.4: Example of a basic block (block (3)) with a join and a condition

and its execution time. This way, the feasible path analysis is achieved at the same time as the WCET analysis. The SMT expressions mean: *"Is there a path that satisfies the semantics and has an execution time greater than T?"* where T is a user defined bound. The solution of this statement represents an execution path in the program with a constraint on the execution time. An SMT-solver is then used to resolve the SMT expressions to answer the aforementioned question. In [Hen+14], the authors use a binary search method with different values of T to find an upper bound on the execution time and disprove the existence of a solution with an execution time greater than the estimated WCET.

The semantics of the program can be used in determining feasible paths. Boolean variables are assigned to each basic block and each transition. A transition between basic blocks i and j is encoded with a Boolean $t_{i,j}$ that is set to true if the transition is taken. A basic block i, encoded with a Boolean b_i, is executed if any of its entering transitions is taken, i.e. $b_i = \bigvee_k t_{k,i}$. The execution time is encoded at each transition in the CFG. Thus, $c_{i,j}$ encodes the execution time between block i and block j. This way, the WCET analysis is achieved at the same time as the feasible path analysis.

Example 7. *To illustrate the SMT encoding, we use the example of block (3) from Figure 4.4. b_3 is a Boolean assigned to block (3). This basic block is executed when any entering transition is taken. Let $t_{1,3}$ and $t_{2,3}$ be Booleans associated with the transitions from block (1) to block (3) and from block (2) to block (3) respectively. The generated SMT expression is:*

$$b_3 = (t_{1,3} \vee t_{2,3})$$

The outgoing transitions are obtained from the condition $(y < 0)$. Transition $t_{3,4}$ is taken when the condition is true, $t_{3,5}$ is taken otherwise. This gives the following expressions:

$$\begin{aligned} y\_cmp &= (y < 0) \\ t_{3,4} &= (b_3 \wedge y\_cmp) \\ t_{3,5} &= (b_3 \wedge \neg y\_cmp) \end{aligned}$$

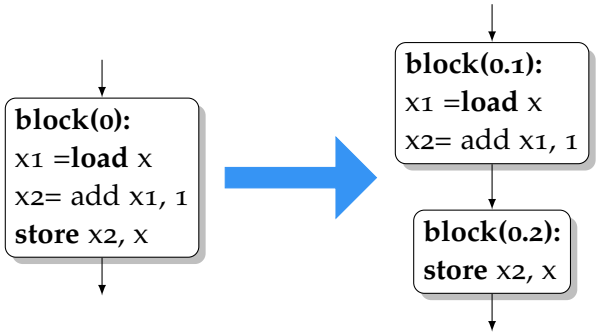


Figure 4.5: Split of basic blocks such that only a single bus access occurs at the beginning of the block

The execution time at each transition is encoded with $c_{1,3}$, $c_{2,3}$, $c_{3,4}$, and $c_{3,5}$. We explain how the total WCET is computed from these variables in Section 4.3 while introducing our contributions.

Note that loops and recursive function calls are problematic cases. We require that the compiler unrolls the loops and inlines function calls. In the remainder of this chapter, we extend this approach to include architectural information of shared resource arbitration in a multi-core processor with a TDMA bus arbiter. Our approach encodes shared TDMA bus and the program semantics within the same SMT expression, allowing the SMT-solver to prove WCET bounds that could not be deduced by analyzing both aspects independently.

4.3 SMT-BASED ANALYSIS FOR TDMA

In this section, we explain how the SMT model is extended to encode accesses to a shared bus with a TDMA arbitration policy. In this work, we consider the first slot $[0, \sigma]$ to be associated to the core on which the analyzed program is executed. To simplify the analysis, we transform the control-flow graph (CFG) so that the basic blocks access the shared bus at most once and only at their first instruction. We will refer to the *basic blocks* in the transformed CFG simply as *blocks*. Figure 4.5 illustrates this transformation: Considering that *load* and *store* instructions access the shared bus, block (0) is split into block (0.1) and block (0.2). Each block either starts with a *load* or a *store* instruction, or does not contain any memory accesses. Note that these transformations simplify the analysis but do not change the general properties of the CFG.

We extend the work of [Hen+14] to include the model of the shared bus accesses. This model is given in Section 4.3.1. Introducing the access delays implies modifications in the SMT encoding of the execution time from the previous work. We explain the timing encoding in presence of bus delays in Section 4.3.2. This work has been published in [Rih+15].

4.3.1 *Naive Timing Encoding*

Here we explain the SMT model of a program that accesses a shared bus with a TDMA arbiter. The encoding of blocks that do not access the shared bus comes straightforward from the previous work by Henry et al. [Hen+14]. A variable $c_{i,j}$ is associated to each

transition between blocks i and j and represents the time spent in block i. The worst-case execution time of each block is constant considering our assumption of a *fully timing compositional architecture*:

$$c_{i,j} = \textbf{if } t_{i,j} \textbf{ then } wcet_i \textbf{ else } 0$$

The expression means: if the transition from block i to block j is taken, $c_{i,j}$ is equal to the worst-case execution time, denoted by $wcet_i$, of block i; otherwise it is equal to 0. Note that a block in the CFG may have different WCETs depending on the execution path it takes part of. To simplify our notations, we use a single constant $wcet_i$ for block i. The values of $wcet_i$ can be obtained from an external timing analysis tool and injected directly in the SMT expression.

For blocks that access the shared resource, the SMT model takes into account the micro-architectural configuration. A TDMA arbitration policy is determined by its period π and slot length σ. The delay of a bus access at the exit of a block T_{exit} is determined according to the instant t of the request emission at the entry of the block T_{entry}. A naive implementation of the bus access model first computes the offset from T_{entry}, i.e, ($T_{entry} \mod \pi$). Then, it checks whether the offset falls into the communication slot.

Algorithm 2 gives the pseudo code of the straightforward encoding in SMT of the bus access delay. The function *tdma_access* takes as argument the time instant of a bus access request and finds its offset relative to the start of the TDMA period. We then check whether the current offset falls into the allowed communication slot. In this case, the access request is directly granted and the function returns the time of the entry plus the access delay and the execution time of the remaining instructions that do not access the bus: $T_{exit} = (T_{entry} + acc + cost)$. Otherwise, the request is delayed until the next slot and the function returns $T_{exit} = T_{entry} + (\pi - off_{entry}) + acc + cost$.

Algorithm 2 Naive version of *tdma_access*: returns the absolute time after a bus access

Require: time: T_{entry} , execution time of the block: $cost$
1: $off_{entry} \leftarrow T_{entry} \mod \pi$
2: **if** $off_{entry} < \sigma$ **then**
3: **return** $T_{entry} + acc + cost$
4: **else**
5: **return** $T_{entry} + (\pi - off_{entry}) + acc + cost$
6: **end if**

This method raises performance issues for the SMT-solver, caused by the use of the non-linear modulo operator "mod". To address this, we present in the following another encoding. In Section 4.4.1, we compare different encodings.

4.3.2 *Optimized Timing Encoding*

In this section, we explain how the timing is encoded with SMT. The assumption made previously on the form of the CFG implies that blocks either access the shared bus or not. Moreover, the blocks access the shared bus only at the first instruction. The timing encoding should take into account such configuration.

Instead of modeling only the absolute time, we also model the offsets throughout the program. We recall that the offset *off* is defined by $off = (T \mod \pi)$. These offsets are computed at each exit point of a block in the CFG. Similarly to the execution time, offsets are associated to each transition; $off_{i,j}$ encodes the offset at the transition from block i to block j.

4.3.2.1 *Blocks Without Bus Accesses*

The encoding of the execution time of blocks without bus accesses is the same as in the naive model above. The novelty here is to also encode the offsets at the entry and the exit of the block. Function *get_offset* in Algorithm 3 is used to find the offset after a block that does not access the shared bus. This function takes as parameters offset off_{entry} at the entry of the considered block and its execution time *cost*. The "mod" operator in line 1 is used to find the offset after a time *cost*. This operator does not cause performance issues because its operands *cost* and π are known constants, as opposed to the naive encoding, where the one of the operands is an unknown variable (T_{entry}). new_off is the sum of two constants smaller than π which means that new_off $< 2\pi$. The algorithm returns $(\text{new\_off} \mod \pi)$ which can be simply written with the *if..then..else* statement in lines 2 to 6.

Algorithm 3 *get_offset*: returns the offset at the exit of a block without bus accesses

Require: offset: off_{entry} , execution time of the block: *cost*
1: new_off $\leftarrow off_{entry} + (cost \mod \pi)$
2: **if** new_off$> \pi$ **then**
3: **return** new_off$-\pi$
4: **else**
5: **return** new_off
6: **end if**

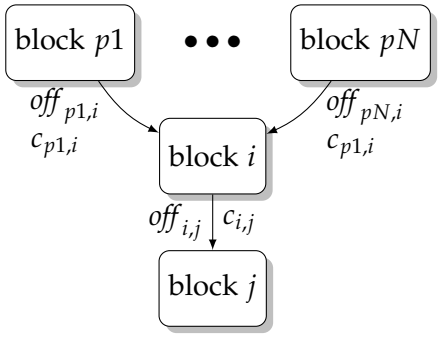


Figure 4.6: A minimal example of a CFG

Let i and j be the indices of two blocks such that block j is a direct successor of block i as illustrated in Figure 4.6. Let $\{p1, p2, \ldots, pN\}(\forall N \geq 1)$ be the direct predecessors of block i. A possible encoding of the offset between block i and block j is the SMT expression:

$$
\begin{aligned}
off_{i,j} = \text{get\_offset}\Big(& (\textbf{if } t_{p1,i} \textbf{ then } off_{p1,i} \\
& \textbf{else if } t_{p2,i} \textbf{ then } off_{p2,i} \\
& \textbf{else if } ... \\
& \textbf{else if } t_{pN,i} \textbf{ then } off_{pN,i} \\
& \textbf{else } 0), \\
& wcet_i \Big)
\end{aligned}
$$

We refer to this encoding as *"if..then..else"* encoding below. This expression means that the offset between block i and block j is computed using the offset of the corresponding entering transition to block i in the case there are many predecessors of i.

Let $off_{k,i}(k \in \{p1, p2, \ldots, pN\})$ be the offsets associated to the entering transitions from blocks k to block i. Only one entry transition is taken in a specific execution path. Let n and i be two blocks in an execution path $\mathcal{P}$ such that n is a direct predecessor of i. On path $\mathcal{P}$ we observe that:

$$
\begin{aligned}
&\text{if } \forall k \neq n : t_{k,i} = \text{false}, \ off_{k,i} = 0 \\
&\text{if } k = n : t_{k,i} = \text{true}, \ off_{k,i} \in [0, \pi[
\end{aligned}
$$

This means that at most one entering offset is not null which implies that the *sum* of all entering offsets to block i gives directly the offset at the entry of the block i in an execution path. Hence, it is possible to avoid using the nested *if..then..else* sequences in the SMT expression by using a *sum* instead. We give such encoding as follows:

$$
\begin{aligned}
off_i &= \text{get\_offset}\Big(\sum_{k \in \{p1,\ldots,pN\}} off_{k,i}, \ wcet_i \Big) \\
off_{i,j} &= \ \textbf{if } t_{i,j} \textbf{ then } off_i \textbf{ else } 0
\end{aligned}
$$

Here off_i is an intermediate variable to encode the offset at the exit of block i. We refer to this encoding as "sum" encoding.

In Example 8, we apply this encoding to block (3) in Figure 4.4.

Example 8. *Let $wcet_3$ be an upper-bound on the execution time of block (3). Let $off_{1,3}$, $off_{2,3}$, $off_{3,4}$, and $off_{3,5}$ be the offsets assigned on the transitions $1 \to 3$, $2 \to 3$, $3 \to 4$, and $3 \to 5$ respectively. The offsets $off_{3,4}$ and $off_{3,5}$ at the exit of block (3) are encoded by:*

$$
\begin{aligned}
off_3 &= get\_offset((off_{1,3} + off_{2,3}), \ wcet_3) \\
off_{3,4} &= \ \textbf{\textit{if}} \ t_{3,4} \ \textbf{\textit{then}} \ off_3 \ \textbf{\textit{else}} \ 0 \\
off_{3,5} &= \ \textbf{\textit{if}} \ t_{3,5} \ \textbf{\textit{then}} \ off_3 \ \textbf{\textit{else}} \ 0
\end{aligned}
$$

4.3.2.2 *Blocks With Bus Accesses*

Blocks that access the shared bus should take into account the delay caused by the arbitration policy. Function *tdma_access* in Algorithm 4 returns the execution time of a block taking into account the offset at its entry (off_{entry}) and the execution time of the remaining instructions (*cost*). Line 1 checks whether the current offset *off* falls into the communication slot. In this case, the request is granted and the returned time at the exit of the block is given by $T_{exit} = acc + cost$. In the other case, $T_{exit} = (\pi - off_{entry}) + acc + cost$.

Algorithm 4 *tdma_access*: returns the delay at the exit of a block with a bus access

Require: offset: off_{entry} , execution time of the block: *cost*
1: **if** $off_{entry} \in [0, \sigma - acc]$ **then**
2: **return** $cost + acc$
3: **else return** $cost + \pi - off_{entry} + acc$
4: **end if**

Function *tdma_offset*, in Algorithm 5, returns the offset at the exit of a block that accesses the bus. This function takes as inputs the offset at the entry of the block off_{entry} and the execution time *cost* of the remaining instructions that do not access the bus. It computes the new offset at the exit block which is $off_{entry} + acc + (cost \mod \pi)$, if the off_{entry} falls into the communication slot $[0, \sigma - acc]$, otherwise the new offset is $acc + (cost \mod \pi)$. Since the offset values can only be in the interval $[0, \pi[$, the modulo operation is computed using *if..then..else* instructions (see lines 6–9) to avoid the non-linear instruction mod.

Algorithm 5 *tdma_offset*: returns the offset after a bus access

Require: offset off_{entry} , execution time of the block: *cost*
1: **if** $off_{entry} \in [0, \sigma - acc]$ **then**
2: new_off $\leftarrow off_{entry} + acc + (cost \mod \pi)$
3: **else**
4: new_off $\leftarrow acc + (cost \mod \pi)$
5: **end if**
6: **if** new_off $\geq \pi$ **then**
7: **return** new_off - π
8: **else return** new_off
9: **end if**

The execution time and the offset at the exit of a block are encoded similarly to blocks without accesses to shared bus. Let i and j be two blocks such that block i is a predecessor of block j, as illustrated in Figure 4.6. Let $\{p1, p2, \ldots, pN\}(N \geq 1)$ be the predecessor

blocks of block i. The functions defined in Algorithms 4 and 5 are used in the following way:

$$
\begin{aligned}
c_i &= \text{tdma\_access}(\sum_{k\in\{p1,\ldots,pN\}} off_{k,i}, wcet_i) \\
c_{i,j} &= \textbf{if } t_{i,j} \textbf{ then } c_i \textbf{ else } 0 \\
off_i &= \text{tdma\_offset}(\sum_{k\in\{p1,\ldots,pN\}} off_{k,i}, wcet_i) \\
off_{i,j} &= \textbf{if } t_{i,j} \textbf{ then } off_i \textbf{ else } 0
\end{aligned}
$$

Here, $wcet_i$ is the worst-case execution time of the remaining instructions after the instruction that accesses the shared bus. If block i has no predecessors, the offset at its entry is set according to the considered hypotheses in the analysis.

4.3.3 *Adding Cuts to the SMT Expression*

Without further optimization, the SMT-solver shows poor performance while searching for the WCET during the experiments. The same issues were observed and addressed in [Hen+14]. The reason is that the DPLL($\mathcal{T}$) algorithm (see Section 4.2.3) works only on the provided atoms in the formula (for instance $x + y + z < 20$) without deriving new ones that may be useful (for instance $x + y < 5$). This leads to some "obvious" properties in the SMT formula being undetected. To address this issue, we manually add *cuts*, which are clauses that allow the SMT-solver to prune a very large number of partial traces from the decision tree.

Knowing that the offsets can only have the values in $[0, \pi[$ gives straightforward cuts in the case of the "sum" encoding. Let N be the number of entering transition to block i. The sum $\sum_{k=0}^{N} off_{k,i}$ is in the interval $[0, \pi[$ since there is at most one non-zero $off_{k,i}$. We add a cut for each block with at least two entering transition, i.e., with $N > 1$.

4.4 IMPLEMENTATION AND EVALUATION

Our implementation relies on PAGAI [HMM12], a tool which allows modeling programs by means of SMT expressions. It is used by Henry et al. [Hen+14] to estimate the worst-case execution time through semantic encoding with SMT expressions. PAGAI uses an intermediate representation based on the CFG obtained from LLVM[2]. Due to this constraint, our tests and proof of concept implementation use the intermediate representation instead of the executable binary. We explain in Section 4.5 how a realistic analysis can be achieved.

Figure 4.7 shows the workflow of the proof of concept. The source code is compiled with CLANG to generate LLVM bitcode. A number of optimization passes are then executed. The interesting pass in our case is the one that transforms the CFG as discussed in Section 4.3. PAGAI is then run on the transformed CFG, given in the form of an LLVM bitcode file, to generate the SMT expressions of the program.

2 LLVM is a compilation framework that relies on a strongly typed intermediate representation (http://www.llvm.org)

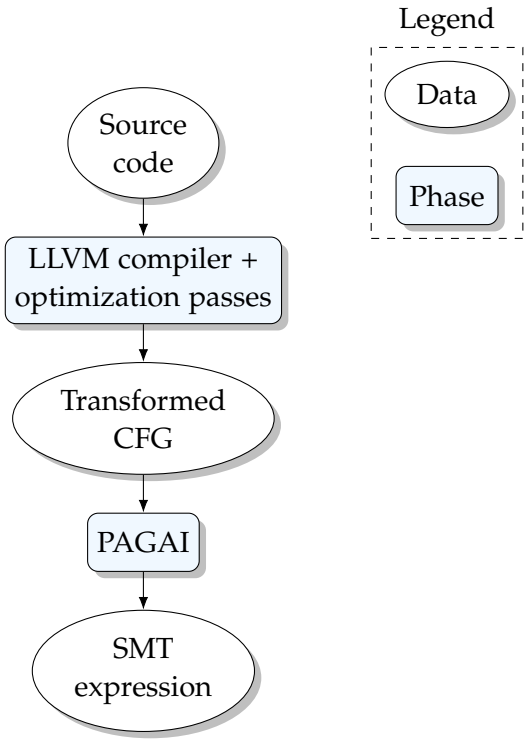


Figure 4.7: General workflow of the proof of concept to generate SMT expressions

We implement an LLVM optimization pass that transforms the CFG to fit our analysis. It splits blocks such that (i) each obtained block contains at most one instruction accessing the bus, (ii) such instruction is the first instruction of the obtained block. Figure 4.2b in Section 4.2.2 illustrates a CFG obtained after this transformation.

We use the SMT-solver Z3. It offers a C API that is used in a binary search program. The SMT-solver parses the SMT expressions and answers with SAT, UNSAT, or UNDEF. In the case of SAT, the SMT-solver gives a model of a solution that satisfies the SMT expression. We use this model to refine the binary search. For example, we look for an execution time in the interval $[X_0, Y_0]$. The binary search algorithm checks whether the execution time is greater than $\frac{X_0+Y_0}{2}$. If UNSAT is returned, the new search interval is $\left[X_1 = X_0, Y_1 = \frac{X_0+Y_0}{2}\right]$. If SAT is returned, the SMT-solver gives a model with an execution time $Z \in \left[\frac{X_0+Y_0}{2}, Y_0\right]$. The new search interval in this case is $[X_1 = Z, Y_1 = Y_0]$. The search continues until it reaches an interval $[X_n, Y_n]$ where $X_n = Y_n$.

This approach, when applied to Algorithm 1, gives the correct and optimal worst-case execution time of 18 processor cycles after 6 iterations of the binary search. The output of the binary search is:

```
Testing wcet >= 0...  SAT (value found = 18).
                              New interval = [18, 73].
Testing wcet >= 46...  UNSAT. New interval = [18, 45].
Testing wcet >= 32...  UNSAT. New interval = [18, 31].
Testing wcet >= 25...  UNSAT. New interval = [18, 24].
Testing wcet >= 21...  UNSAT. New interval = [18, 20].
Testing wcet >= 19...  UNSAT. New interval = [18, 18].
The wcet is 18 .

Computation time is 0.010000s
```

In the following, we evaluate our model of the shared TDMA bus. First, we propose a micro-benchmark to compare the results of the naive implementation of *tdma_access* and the offset-based implementation. Then, we show how the semantics encoding combined with a TDMA bus model can enhance the WCET estimation using (i) a toy example to illustrate the differences and (ii) real-world applications. For the simplicity of our proof of concept implementation, we suppose that all programs start initially at offset $off = 0$. In a real application we should consider all possible values of the offset.

4.4.1 *Performance of SMT Encodings for TDMA*

4.4.1.1 *TDMA Functions*

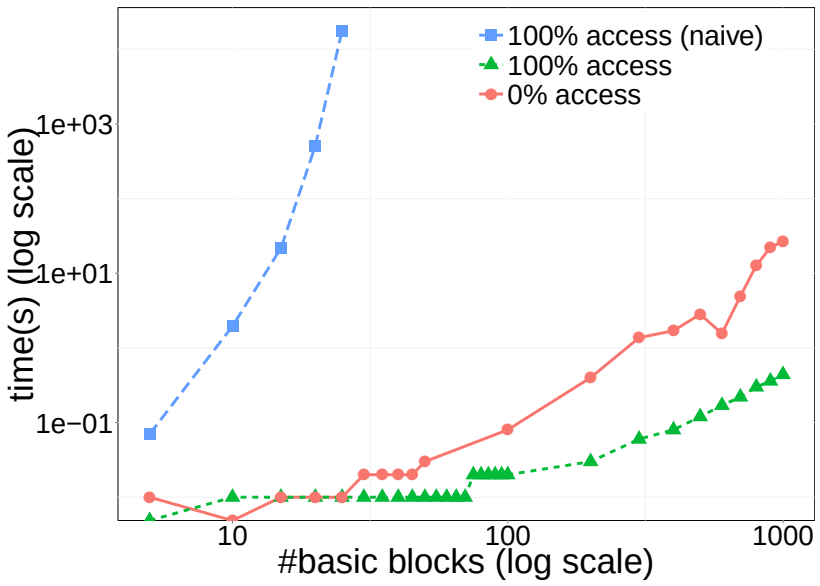
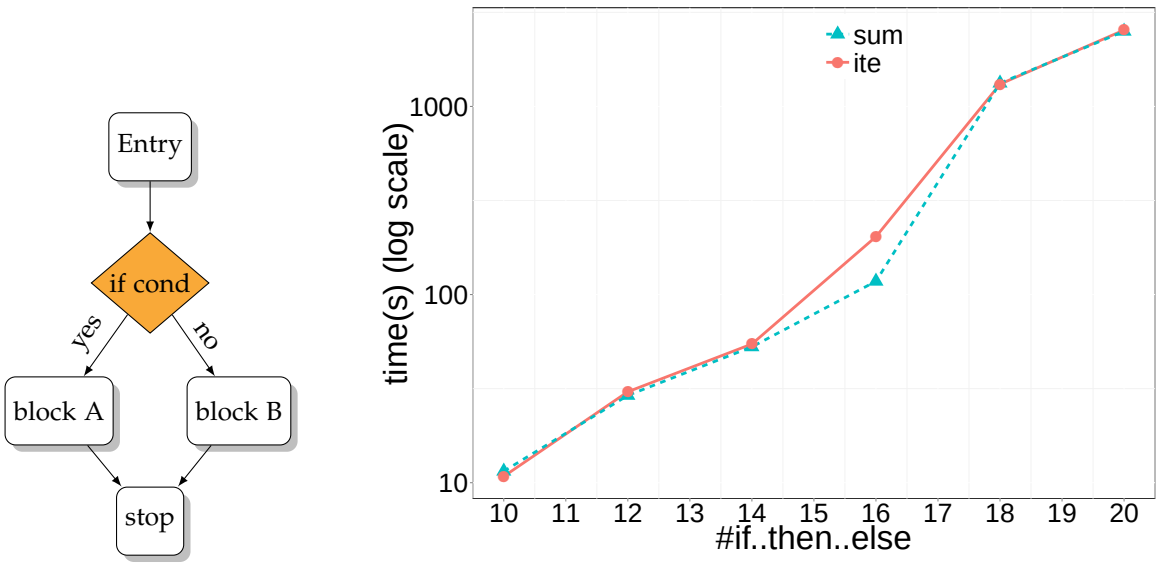


Figure 4.8: Comparison of the naive implementation of *tdma_access* ■, the offset-based implementation of *tdma_access* ▲, and *get_offset* ●

We now evaluate the analysis time of our model. A simple approach is to evaluate the analysis time on a linear path, i.e. without branches. The blocks are simple and have only one instruction each. Figure 4.8 shows a comparison of the different setups. We compare the naive implementation and the offset-based implementation of *tdma_access* on a CFG that contains only blocks with accesses to the shared bus. The naive implementation has an exponential growth of the analysis time. At only 25 blocks, it takes 17 656 s for the binary search with the SMT-solver to find the WCET, whereas, it takes only 0.44 s in the case of the offset-based implementation. This is mainly due to the non-linear mod operator used in the naive implementation. The line *"0% access"* represents a CFG composed with blocks that do not access the shared bus. This shows the performance of function *get_offset*.

4.4.1.2 *Offset Encoding*



(a) Example of a diamond formula

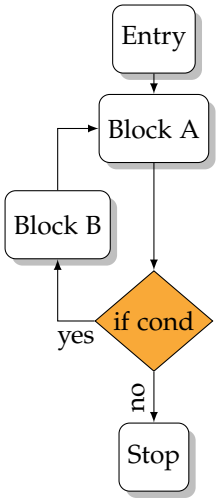
(b) Comparison of nested *if..then..else* (ite) and *sum* encoding of sequences of *if..then..else* (Figure 4.9a). TDMA bus: $\sigma = 40$, $\pi = 160$, $acc = 10$

Figure 4.9: Performance comparison of diamond formulas encodings

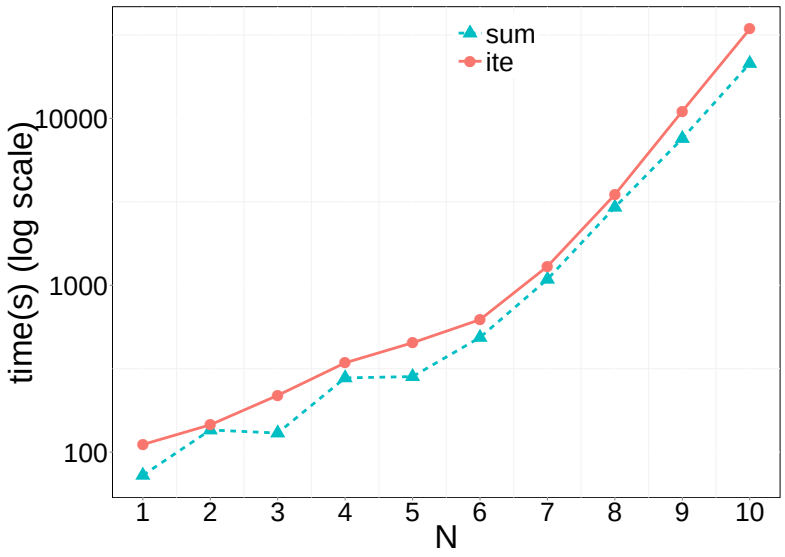
We now compare the two encodings explained in Sections 4.3.2.1 and 4.3.2.2. Figure 4.9a shows an example with one *if* condition which will generate a "diamond formula" in SMT. We compare the analysis time of the nested *if..then..else* encoding against the *sum* encoding of an increasing number of sequences of "diamond formulas" in the analyzed program. Figure 4.9b shows the results for execution time of the analysis when *Block A* and *Block B* in Figure 4.9a access the shared TDMA bus. Both encodings have almost the same analysis time with a slight advantage of the *sum* encoding.

To investigate further, we analyze the program represented in Figure 4.10a. The loop bound is 100 iterations which will generate, when the loop is unrolled, a block with 100 entering transitions. We analyze programs with N sequences of the same loop. Figure 4.10b shows the analysis time of the encodings with N in $\{1..10\}$. The *sum* encoding shows better

performance than the nested *if..then..else* encoding. For the rest of the experiments, we use the *sum* encoding.



(a) Example of a program with a loop



(b) Comparison of nested *if..then..else* (ite) and *sum* encoding of sequences of loops with 100 iterations (Figure 4.10a). TDMA bus: $\sigma = 40$, $\pi = 160$, $acc = 10$

Figure 4.10: Performance comparison of unrolled loops encodings

4.4.2 *Benchmarks*

4.4.2.1 *Experimental Setup*

We evaluate our approach with a subset of the TacleBench[3] benchmarks. The benchmarks are compiled with CLANG 3.6 to generate the LLVM bitcode. Loops are unrolled with an optimization pass of LLVM. The SMT expression is generated following the workflow in Figure 4.7. The examples are illustrated in Table 4.1. Here, "#LLVM instr." refers to the number of the instructions in the LLVM bitcode after inlining and unrolling functions and loops; "#bus access" represents the total number of `load` and `store` instructions since we consider an architecture without a cache memory. The LLVM bitcode has more instructions compared to the binary executable. Some *load* and *store* instructions in the LLVM bitcode do not exist in the executable binary. Thus, a direct comparison with other approaches, that perform directly on the binary, is not applicable. Our proof of concept, however, allows demonstrating the feasibility of the SMT-based approach.

The analysis is run under Linux Debian, on an Intel® Core® i5-3470 at 3.20 GHz with 8 GB of main memory. We consider each instruction to execute in 1 processor cycle and the platform has no cache memory (see Section 4.1 for the assumptions).

3 http://tacle.knossosnet.gr/activities/taclebench

Name	Description	#LLVM instr.	#bus access
bs	Binary search	231	12
insertsort	Insertion sort on a reversed array	493	65
jfdctint	Discrete Cosine Transformation	2334	448
fdct	Fast Discrete Cosine Transform	2502	385
compressdata	Data compression program adopted from SPEC95	674	131
fly-by-wire	UAV fly-by-wire software	2815	515

Table 4.1: Benchmarks

4.4.2.2 *Results*

The TDMA policy statically isolates programs in their respective slots which means that the analysis for each program is independent from the other programs. We therefore run the analysis for individual programs, but the results hold in a context where several programs are executed in parallel.

We compare the WCET of the offset-based analysis with the pessimistic WCET where all accesses to the shared bus are considered worst-case. This implies that each *load* and *store* instructions have an execution time of $\pi - \sigma + 2.acc - 1$. Similarly to [Lv+10], the improvement is defined as

$$S = \frac{\text{(pessimistic WCET)}}{\text{WCET}} - 1$$

We analyze different configurations of the TDMA bus. The results are illustrated in Tables 4.2, 4.3, 4.4, 4.5, and 4.6. The improvements we obtain from the offset-based analysis are proportional to the slot length and the period of the TDMA bus. The results also show that the greater the slot length is, the greater the improvement. This is expected since more accesses can be executed in the same slot. A greater TDMA period increases the pessimistic WCET. The highest improvement is 217.95% of the `bs` benchmark (231 LLVM instructions) in Table 4.5 with $\pi = 400$ and $\sigma = 200$.

Table 4.7 represents the lowest and highest observed analysis times. The offset encoding increases the analysis time of programs. The pessimistic WCET of benchmark `fly-by-wire`, from the `PapaBench` suite, is obtained in 4.02 seconds. The offset-based encoding has an analysis time of 149.01 seconds ($\pi = 400$, $\sigma = 100$, $acc = 40$). Despite the effort to linearize the SMT functions used to model the TDMA bus access, they are still very costly. The analysis time depends on the number of accesses to the shared bus as well as the number of "diamond formulas" which appears at the encoding of sequences of *if..then..else*.

Name	$WCET_{pess}$	WCET	Improvement
bs	328	261	25.67%
insertsort	1331	1313	1.37%
jfdctint	19544	17893	9.22%
fdct	17296	16012	8.01%
compressdata	2650	2275	16.48%
fly-by-wire	6201	5708	8.63%

Table 4.2: Results with the TDMA bus configuration: $\pi = 40$, $\sigma = 20$, $acc = 10$

Name	$WCET_{pess}$	WCET	Improvement
bs	448	261	71.64%
insertsort	1951	880	121.70%
jfdctint	28504	13213	115.72%
fdct	24996	11545	116.50%
compressdata	3790	1865	103.21%
fly-by-wire	9061	4312	110.13%

Table 4.3: Results with the TDMA bus configuration: $\pi = 80$, $\sigma = 40$, $acc = 10$

Name	$WCET_{pess}$	WCET	Improvement
bs	928	501	85.22%
insertsort	4431	1760	151.76%
jfdctint	64344	26413	143.60%
fdct	55796	23065	141.90%
compressdata	8350	3705	125.37%
fly-by-wire	20501	8682	136.13%

Table 4.4: Results with the TDMA bus configuration: $\pi = 160$, $\sigma = 40$, $acc = 10$

Name	$WCET_{pess}$	WCET	Improvement
bs	1768	556	217.95%
insertsort	8771	3263	168.80%
jfdctint	127064	44578	185.03%
fdct	109696	38442	185.35%
compressdata	16330	5799	181.60%
fly-by-wire	40521	14195	185.45%

Table 4.5: Results with the TDMA bus configuration: $\pi = 400$, $\sigma = 200$, $acc = 40$

Name	$WCET_{pess}$	WCET	Improvement
bs	2368	1251	89.28%
insertsort	11871	6463	83.67%
jfdctint	171864	89288	92.48%
fdct	148196	76842	92.85%
compressdata	22030	12455	76.87%
fly-by-wire	54821	29258	87.37%

Table 4.6: Results with the TDMA bus configuration: $\pi = 400$, $\sigma = 100$, $acc = 40$

Name	$(\pi = 40, \sigma = 20, acc = 10)$	$(\pi = 400, \sigma = 100, acc = 40)$
bs	0.45	0.98
insertsort	1.37	6.56
jfdctint	44.10	48.54
fdct	41.36	34.57
compressdata	4.66	3.23
fly-by-wire	28.78	149.01

Table 4.7: Analysis time, in seconds, of the benchmarks with different configurations of the TDMA bus

4.5 CONCLUSIONS AND FUTURE WORK

4.5.1 *Summary*

We introduce a new approach for WCET analysis of shared TDMA buses using Satisfiability Modulo Theory (SMT). This approach takes into account the semantics and the accesses to a shared TDMA bus to give a tighter estimation of the execution time. In our proof of concept, we consider a platform without cache memory which means that all *load* and *store*

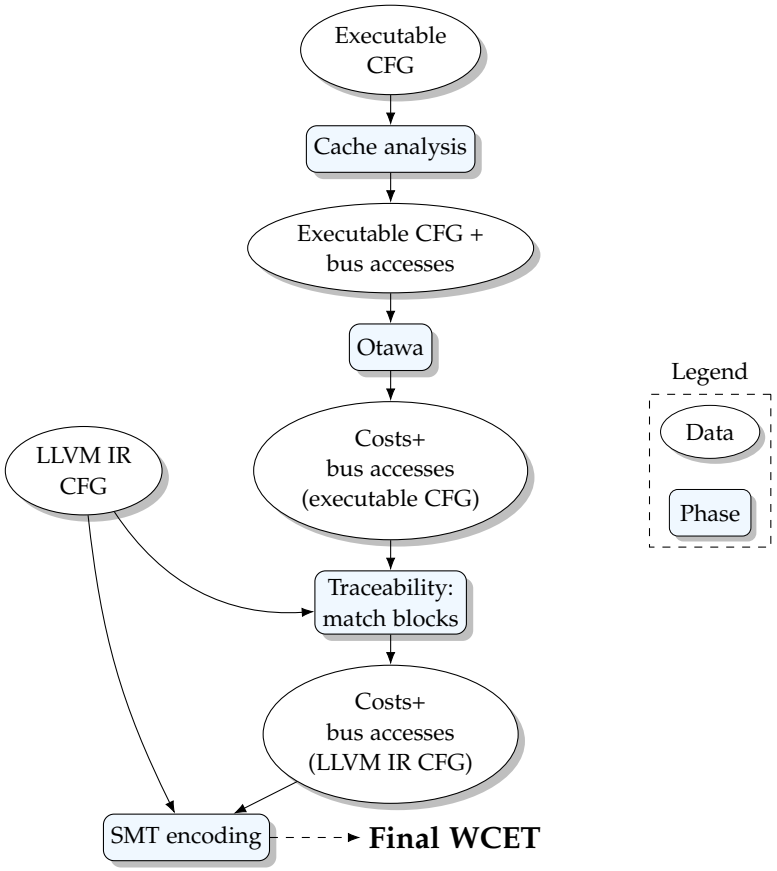


Figure 4.11: General workflow for realistic timing analysis

instructions access the shared bus. We also analyze programs in the form of LLVM bitcode due to the constraints imposed by the tool PAGAI. This is a limitation of our implementation, but not of the approach itself: the same approach can be applied to executable binaries given a generated model in SMT and with the presence of cache memory. Accesses to the bus can be obtained from an analysis of the cache's state where a cache miss is considered as an access to the shared memory through the shared bus.

The naive model of the TDMA bus shows poor performance. To overcome the issue, we propose an offset based model. The micro-benchmarks show a better scalability but remains exponential. The added cuts on the offsets improve the analysis time by indicating to the SMT-solver "obvious" properties.

Finally, we show that the micro-architectural analysis of the shared TDMA bus, and the semantic analysis can be combined in one approach using an SMT model. This approach can achieve a more precise estimation of the WCET in the presence of a shared TDMA bus. The naive encodings are very costly. We give alternative encodings that reduce considerably the solving time of the SMT expression.

4.5.2 *Future Work*

The current implementation of our approach is a proof of concept, which checks the viability and scalability of the approach. As such, taking into account a realistic model for the timing of the program is left to future work. Considering that each LLVM instruction takes exactly one cycle is clearly not realistic: the timing for each block should instead come from a micro-architectural analysis of the actual binary with a tool like OTAWA [Bal+10]. Keeping the analysis itself on LLVM bitcode allows exploiting high-level properties of the program that would be lost at the binary code level, and the SSA form of the bitcode greatly simplifies the encoding into SMT. As a consequence, a complete tool for a realistic analysis would need to work both on the binary code and the LLVM bitcode. The information obtained on the binary must be mapped to the LLVM bitcode. One solution to achieve this is through pattern matching of conditions [Bie+13] between the LLVM CFG and the executable CFG. The overall approach for such an information flow is described in Figure 4.11. It has already been applied to SMT-based WCET analysis in [Hen+14]. The idea of combining high-level semantic information with low-level binary analysis has also already been applied, for instance, in [LPR14; Ray+15].

Similarly, considering LLVM *load* and *store* operations as bus accesses is an oversimplification. Some LLVM *load* and *store* will actually be cache hits and will not access the bus, and conversely, some operations on LLVM registers will actually need to access the memory in the real program. The actual bus accesses must therefore be obtained by a prior cache analysis on the binary code [Alt+96].

Our experiments show scalability issues which is expected in NP-complete problems. We are considering optimizations and improvements in the scalability in future work. Our approach already shows substantial improvements over a naive encoding, and the results show that we do scale to reasonably-sized programs. In case of very large case-studies globally with this approach, we would probably encounter performance issues in the SMT solver to scale. We therefore need an approach that uses our analysis on reasonably-sized pieces of code extracted from a possibly larger codebase. One option is to analyze the program in portions and propagate the obtained results on a global analysis. For example, considering only a small piece of code surrounding a bus access may be sufficient to prove that this access is in the TDMA slot (or to prove a tight bound on its execution time), and this information can be injected in a global cheaper analysis. The challenge here is how one defines the analyzed portions and their sizes.

Loops with a large iteration count, which cannot be unrolled completely, could be handled using partial unrolling with an unroll factor. Loop iterations are then analyzed separately with updated information on offsets between each iteration. Kelter et al. [Kel+14] already address the loop analysis with minimum unrolling. The SMT-based approach can be complementary to include the semantics in the loop body analysis.

4.5.3 *Discussion*

We demonstrate the approach described above on an application model with independent tasks. The model of SDF applications, as presented in Chapter 2, considers tasks with

precedence/dependence relation; one execution instance of the SDF is represented with a directed acyclic graph (DAG) representing the dependent tasks. In this chapter, we see that the TDMA bus allows a time isolation of the parallel tasks; the timing interference depends only on the TDMA slots and not on the co-runner tasks. Thus, the number of co-runners does not affect the task under analysis. For an acyclic task graph in a shared resource environment with a TDMA bus, the analysis can be performed as follows: the first tasks in the DAG (the ones with no precedences) are analyzed while considering the set of all possible offsets at the entry of each task. Then, the offsets at the worst-case finish time of each task are propagated to the next tasks in the DAG. This results in a static schedule that respects the dependency relation. If the hardware or software allow it, the tasks may also be aligned on their corresponding TDMA slots which improves the analysis by reducing the set of possible offsets at the entry of each task.

The TDMA arbitration policy is *non-work conserving*. It means that shared resource accesses are stalled when they occur outside of their slots even when there is no concurrent accesses. In contrast, a *work-conserving* policy, such as *Round Robin* or *Fixed Priority*, does not stall accesses if there is no concurrency on shared resources. Although these policies are more efficient in terms of performance, the WCET is no longer independent from co-runners. A pessimistic way to account for the interference is to consider the worst-case delay at each access. Albeit this approach may scale well, it results in an unnecessarily large over-approximation. For a precise analysis, the SMT-based approach needs to model all tasks in the system as well as the bus in order to determine whether concurrent accesses are interfering. Considering platforms with tens or hundreds of cores, the number of concurrent tasks increases the complexity of the approach exponentially.

Moreover, the result of this approach is not trivial in the case of dependent tasks. Each task depends on the execution time of its precedence; it cannot start until the completion of all the precedence. The execution times of the precedence depend in turn on the co-runners. It requires a fixed-point to solve such system (see Chapter 5), which increases the complexity of an SMT-based approach. Finally, the system may not be solvable at all with the currently available tools and hardware.

In the remainder of this thesis, we approach the problem differently; instead of counting the interference at a precision of cycles, or always considering the worst-case delay, we take into account the number of shared resource accesses of each task. We consider that accesses interfere whenever they occur in the same time frame. This is a trade-off that offers a scalable analysis with a certain degree of precision.

Chapter 5

RESPONSE TIME ANALYSIS ON MULTI-CORE SYSTEMS

In this chapter, we introduce a response time analysis technique for Synchronous Data Flow (SDF) programs represented with multiple parallel dependent tasks running on a multi-core/many-core processor. The analysis computes a set of response times and release dates that respect the constraints in the task dependency graph. Our approach provides tight response times by taking into account the release dates of co-runner tasks on a multi-core with partitioned shared resources.

This chapter is organized as follows. Section 5.1 describes the target multi-core architecture and application models. Section 5.2 provides our response time analysis for synchronous data-flow programs. The proof of correctness of our analysis is discussed in Section 5.3. Section 5.4 concludes with a summary and a discussion of future work.

5.1 DATA-FLOW APPLICATIONS ON MULTI-CORE PLATFORMS

We first define the target processor on which the application runs. Then, we present the different notions used in our analysis of data-flow applications. Our approach considers an execution instance of the data-flow which is represented with a mono-rate dependent task graph.

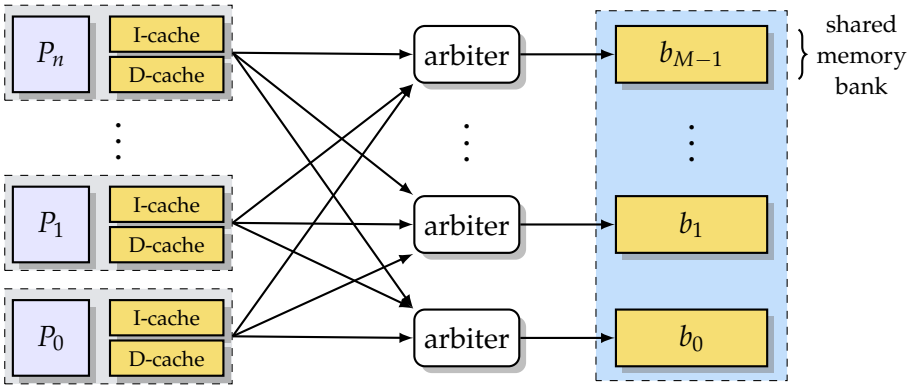


Figure 5.1: Multi-core, shared multi-bank memory architecture model

5.1.1 *Shared Multi-Bank Memory, Multi-core Architecture*

We consider a multi-core system with N identical cores. As presented in Chapter 2, we define the set of cores $\Pi = \{P_0, P_1, \ldots, P_{N-1}\}$. We also define the mapping function $Map : \Gamma \to \Pi$ that maps each task in Γ to a core in Π. We consider that cores have local caches for data and instruction, and a shared memory. Moreover, we consider a partitioned shared memory into M memory banks with a contiguous memory address space that spans through the banks. We define the set of memory banks as $\mathcal{B} = \{b_0, b_1, \ldots, b_{M-1}\}$. Figure 5.1 illustrates this architecture model. Each bank is accessed through dedicated bus arbiter from any core. When $M = 1$, cores always interfere when accessing the shared memory at the same time. When $M > 1$, cores do not interfere when accessing different memory banks. This particular architecture already exists in industrial platforms such the many-core Kalray MPPA-256 [Din+14b].

5.1.2 *Dependent Task Graph Model*

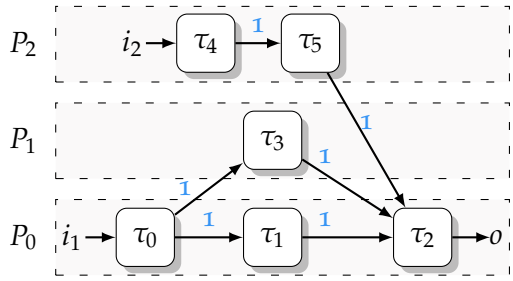
Our aim is to obtain accurate bounds on the worst-case response time for data-flow programs. An execution instance of a periodic data-flow is seen as a dependent task graph. In a mono-rate program, each task is executed once according to its dependencies. In the case of a multi-rate program, we assume an unfolded execution to the hyper-period (the least common multiple of the tasks' periods), effectively reducing the problem to a mono-rate one[1]. Also, we consider that all tasks in a cycle must complete before the end of the cycle, which is a common constraint when scheduling synchronous programs. As a consequence, scheduling can be done on one period (or hyper-period); the same schedule is then repeated indefinitely.

Our algorithm takes as input a fixed mapping of tasks to cores, and a fixed order for tasks mapped to the same core. We purposely delegate the mapping and ordering to a

1 The unfolding preserves the required minimum separation between jobs, since our scheduling scheme includes fixed release dates for each task. The unfolding process thus assigns proper release dates to multiple instances of the same task. We note the potentially large size of the hyper-period which may introduce a complexity issue.

separate tool, such the one in [NHP15; NHP17], dedicated to optimization of the schedule and mapping. Mapping and ordering of tasks can also be done manually. We produce a completely static, time-driven schedule. There cannot be two tasks active on the same core at the same time, hence we do not use preemption and a task starts immediately when it is released. The schedule specifies the exact, fixed release date for each task. This is pessimistic in the sense that each task waits for the worst-case response time of each of the tasks it is dependent on (it cannot start even if all of them have completed well before their deadline); however, our aim is to optimize the worst case, not the average case. The scheduling scheme has good properties for a hard-real time system. First, it enables the application to be executed without any operating system: we only require communication primitives, and one primitive to wait for a specified instant; they can be provided as a simple library. Also, it makes the whole execution highly predictable since the release date of a task does not depend on the execution time of previous tasks: we avoid any potential domino effects in timing.

Figure 5.2 illustrates an example of the considered application model. The task graph and mapping is given in Figure 5.2a. Each core is assigned to a memory bank. Cache misses and uncached operations go to the assigned memory bank whereas communications access the target task's memory bank. We define wcet_i the worst-case execution time of task τ_i in isolation (i.e., when executing without interference). MD_i^b ($b \in \mathcal{B}$) represents the number of accesses of task τ_i to memory bank b. A summary of the tasks' profiles is given in Table 5.2b.



(a) Data-flow application under analysis. The numbers on edges represent the accesses due to communication.

Tasks	τ_0	τ_1	τ_2	τ_3	τ_4	τ_5
wcet (cycles)	5	5	5	5	5	5
MD^{b_0} (accesses)	3	3	2	1		1
MD^{b_1} (accesses)	1			2		
MD^{b_2} (accesses)					3	2

(b) Task profiles

Figure 5.2: Example of a SDF graph and the result of the static analysis.

We introduce the following additional notation used in our analysis: each task τ_i has a release date rel_i (effectively an offset relative to the start of the period of the data-flow program) $\Theta = \{rel_1, ..., rel_n\}$ is the set of release dates and $\mathcal{R} = \{R_1, ..., R_n\}$ is the set of upper bound response times of tasks in Γ. Note that there is no order relation between rel_i and rel_{i+1} (resp. R_i and R_{i+1}) in the set Θ (resp. $\mathcal{R}$). Recall that each task is statically mapped to a core.

The approach we propose takes into account the interference on the bus as part of the response time analysis. By considering the SDF model presented in Chapter 2, we know which tasks could potentially execute at the same time and therefore be co-runners. We make use of this information to derive tight bounds on the amount of interference. Moreover, there is an implicit dependency between two successive periodic instances which

allows us to limit the analysis to only one instance of the task graph. Our analysis is based on an existing framework for multi-core response time analysis (MRTA) [Alt+15] and is detailed below in Section 5.2. In this work, we consider a static non-preemptive scheduling.

To summarize, in addition to setting a model for the shared memory architecture, our approach takes into account the task dependencies and the precise schedule including release dates and response times. In contrast, the original MRTA framework considers sporadic tasks, but does not exploit any knowledge of dependencies or sequentiality between them.

5.1.3 *Phase-based Execution Model*

We presented different execution models in Chapter 2. In this chapter, we first consider a single-phase execution model where we make no assumptions about the distribution of read and write accesses between the start and the end of a task. In some code generation schemes [Gra+18] for the SDF model, tasks execute computations, then write the result to a shared memory location where the next task can read it. Similar to [Mel+15], this execution model allows each task to be split into a first *execution* phase limited to reading the input and doing computations, and then a *write* phase where the output is sent to the next task. In the execution phase, accesses are to the local memory bank of the task whereas in the write phase, requests may access a remote memory bank[2]. We exploit this execution model in our analysis. We consider the two phases of a task as separate sub-tasks with a direct dependency relation. The sub-tasks have their own release dates. As a consequence, the considered task graphs doubles in size. In Chapter 7, we compare the single-phase model with the two-phase model using our analysis technique.

5.2 RESPONSE TIME ANALYSIS

Our approach relies on the existing framework MRTA [Alt+15]. Taking into account the application and architecture models presented above, we present in this section a refined analysis. The proposed algorithm takes into account the release dates of tasks and their dependencies, and extend the interference to accounts for the shared multi-bank memory.

5.2.1 *Multi-core Response Time Analysis*

Here, we outline the generic framework for Multi-core Response Time Analysis (MRTA) introduced by Altmeyer et al. [Alt+15], which we subsequently build upon. The MRTA framework represents a generic and compositional solution for response time analysis. It allows modeling of a wide range of different arbitration policies (and a combination of them), as well as different memory models (no cache, data and instruction cache, scratchpads, etc. and a combination of them). In this paper, we build upon the MRTA framework, instantiating it for different hardware components, bus arbitration policies, and application models.

2 A *local* memory bank, is a memory bank assigned to the core where the task of interest is executing. A *remote* memory bank is a memory bank assigned to another core.

Given a set of n sporadic tasks $\Gamma = \{\tau_0, \ldots, \tau_{n-1}\}$, where each task τ_i has a period or a minimum inter-arrival time T_i and a deadline D_i, and is statically assigned to a core, the MRTA framework computes the response time of each task taking into account the total interference at different levels of the hardware that could occur during the task's response time. By convention, we use P_x to mean the core that the task under analysis is mapped to, and P_y to indicate some other core. The subset of tasks mapped to a core P_y is denoted by Γ_y.

In the MRTA framework, tasks are represented by a set of traces, each of which consists of an ordered list of instructions, where each instruction carries information about the memory locations accessed (if any). A set of exhaustive traces (i.e. for different paths) can be used to give a sound over-approximation of the memory demand and the processor demand of a task by taking the maximum memory (processor) demand over all traces for the task. As a result, the framework decouples response time analysis from a reliance on context independent WCET values (in isolation). The analysis formulates response times directly from the demands on different hardware resources. Such a separation of concerns trades different sources of pessimism. These simplifications make the analysis tractable but are unable to take advantage of overlaps between processing and memory demands of co-runner tasks; however, this compromise is set against substantial gains in the scalability of the analysis, acquired by considering the worst-case behavior of hardware resources, such as the memory bus, over long durations equating to task response times, rather than summing the worst case over short durations such as single accesses, as is the case with the traditional approach using context-independent WCETs.

With the MRTA framework, the response time R_i of task τ_i executing on core P_x is computed using the following fixed-point relation:

$$R_i = \text{wcet}_i + I^{\text{PROC}}(i, x, R_i) + I^{\text{BUS}}(i, x, R_i) + I^{\text{DRAM}}(i, x, R_i) \tag{5.1}$$

Where wcet_i is the WCET in isolation time of task τ_i in isolation. I^{PROC} is the interference on the core due to higher priority tasks preempting or delaying task τ_i. Tasks running on different cores and accessing a shared memory have to traverse a shared bus. According to the used arbitration policy, concurrent tasks may experience different delays. I^{BUS} is the interference on the bus computed using a mathematical model of the bus arbiter. Finally, I^{DRAM} is the interference due to DRAM refreshes. Equation (5.1) is solved as part of a larger fixed-point iteration which operates over the set of tasks. The sufficient condition for the algorithm to converge toward a fixed-point is that I^{PROC}, I^{BUS}, and I^{DRAM} are *monotonic* and *bounded* functions.

In this thesis, we focus only on the shared bus interference. We assume a pre-processing step where all the code is fetched from the DRAM. We also assume a local or shared cache that is large enough to hold the application and tasks do not perform accesses to the DRAM. This means that the interference on the DRAM is $I^{\text{DRAM}} = 0$. We also consider a non-preemptive scheduling, therefore the interference on the core is $I^{\text{PROC}} = 0$.

Our analysis is tailored to the considered application model. We take advantage of the dependencies between the tasks to improve the interference analysis. Concurrent tasks mutually interfere whenever they overlap during execution. Figure 5.3 shows how the interference is considered. Thus, the set of response times $\mathcal{R} = \{R_i | \forall i \in \Gamma\}$ and the set of release

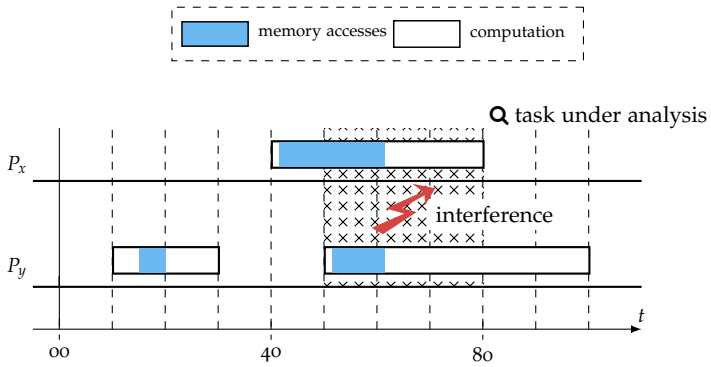


Figure 5.3: Interference from tasks on core P_y on the task on core P_x, where $P_y, P_x \in \Pi$ and $P_y \neq P_x$. Only overlapping tasks mutually interfere.

dates $\Theta = \{rel_i | \forall i \in \Gamma\}$ are required to determine tasks' overlapping. With this redefinition, the interference on the bus is obtained with the modified function $I^{\text{BUS}}(i, x, \mathcal{R}, \Theta)$. Finally, according to the assumptions above, Equation 5.1 is simplified to:

$$R_i = \text{wcet}_i + I^{\text{BUS}}(i, x, \mathcal{R}, \Theta) \tag{5.2}$$

The bus interference depends on the number of shared resource accesses from concurrent tasks that may delay the task under analysis τ_i. Considering the architecture model in Figure 5.1, tasks access memory banks through different arbiters where interference may occur on each bank. This function is defined by the formula:

$$I^{\text{BUS}}(i, x, \mathcal{R}, \Theta) = \sum_{b \in \mathcal{B}_i} \text{BUS}_b(i, x, \mathcal{R}, \Theta) \times d \tag{5.3}$$

where, $\mathcal{B}_i$ is the set of memory banks accessed by τ_i. $\text{BUS}_b(i, x, \mathcal{R}, \Theta)$ returns the number of accesses to memory bank b and which may delay the execution of task τ_i. This function models the arbitration policy used in the architecture. d is the time required to perform a memory access, also called bus latency or bus delay.

Equation 5.2 is a fixed-point formula that can be solved using *Kleene* iteration. Since $wcet_i$ is constant, a sufficient condition to converge towards a fixed-point is that $\forall b \in \mathcal{B}_i$, $\text{BUS}_b(i, x, \mathcal{R}, \Theta)$ is bounded and monotonic with regard to the arbitration policy being used; for the same release dates, the interference on shared resources cannot decrease when the response time increases. The definition of this function depends on the hardware platform. In Chapter 6, we focus on a specific many-core architecture for which we model the shared bus.

5.2.2 *Analysis of Dependent Task Graphs*

Our response time analysis algorithm (Algorithm 6) is based on the MRTA approach [Alt+15]. The original MRTA framework uses a model with sporadic independent tasks with minimum inter-arrival times and, while we analyze a single period of a single-rate application, with a static schedule and task dependencies. Using initial release dates,

Algorithm 6 Response Time Analysis Given a Set of Release Dates

```
1:  function MultiCoreRTA(Θ)
2:      l = 1
3:      ∀i : R^l[i] = wcet_i
4:      do
5:          for all i do
6:              R^{l+1}[i] = wcet_i + I^BUS(i, x, R^l, Θ)
7:                                          ▷ I^BUS is bounded and monotonic
8:          end for
9:          l = l + 1
10:     while R^l ≠ R^{l−1}
11:     return R^l
12: end function
```

Algorithm 7 Update Release Times to Start After All Dependencies

```
1:  function UpdateReleases(Θ_min,Θ,R)
2:      for all i do                    ▷ traverse tasks in a topological order
3:          Θ[i] = max(Θ_min[i], {Θ[k] + R[k] | k ∈ deps(i)})
4:      end for
5:      return Θ
6:  end function
```

Algorithm 6 computes response times that account for the interference. Since the potential interference depends on the release dates and response times, and the response times depend on the interference, this requires a fixed-point iteration (Line 10).

Algorithm 6 solves Equation 5.2 using a fixed-point iteration, and computes the response times $\mathcal{R}$ of all tasks given the release dates Θ. First, all response times are initialized with the tasks' WCETs in isolation. The response times are then computed to include the bus interference that occurs during the tasks' execution. Adding the interference increases the response time consequently. As long as the response times change (Line 10), the interference must be recomputed to account for the potential concurrent tasks. For a bounded and monotonic I^{BUS}, the process converges toward a fixed-point for the given set of release dates Θ.

After computing the response times, the schedule we get may not respect the dependencies and sequentiality constraints; it is possible that a task, suffering from interference, gets delayed and overruns the tasks that depend on it. We modify the release dates so that each task is released immediately after each of the tasks it depends on is guaranteed to have completed (Algorithm 7). Modifying the release dates may change the interference, hence we have to re-compute it using MultiCoreRTA (Algorithm 6), and so on, until a fixed point is reached (Algorithm 8).

UpdateReleases in Algorithm 7 ensures the dependency constraints between tasks are satisfied. It is parameterised by Θ_{min} which gives the earliest release date for each task: $\Theta_{\text{min}}[i] = t$ means that task τ_i cannot start before t. $deps(i)$ (Line 3) gives the set of tasks on which task τ_i depends. A task τ_i is released only when all the tasks in $deps(i)$ are guaranteed

Algorithm 8 Adapt Release Dates to Meet Real-Time Constraints

```
1:  function ComputeRT(Θ_min)
2:      l = 0
3:      Θ^l = InitRelease(), R^l = ⊥
4:      do
5:          R^{l+1} = MultiCoreRTA(Θ^l)
6:          Θ^{l+1} = UpdateReleases(Θ_min, Θ^l, R^{l+1})
7:          l = l + 1
8:      while Θ^l ≠ Θ^{l−1}
9:      if ∀_i : (Θ^l[i] + R[i]^l) ≤ D_i then
10:         return "schedulable"
11:     else return "NOT schedulable"
12:     end if
13: end function
```

to have finished. We statically schedule every release date, hence we set the release date of each task to the maximum of the *worst-case* finish time of each task it depends on.

ComputeRT in Algorithm 8 is the top level of our analysis; it uses MultiCoreRTA (Algorithm 6) to compute the response times of tasks in Γ given a set of release dates (Line 5). Then, UpdateReleases (Algorithm 7) is used to verify and update the dependency constraints. Algorithm 8 starts from initial release dates (bounded by the SDF period) (InitRelease, Line 3) and performs a fixed-point iteration.

ComputeRT terminates when UpdateReleases does not change the release dates. When the release dates are stable, the response times $\mathcal{R}^{l+1}$ computed before the call to UpdateReleases remain valid afterwards, and hence are valid at the end of the loop. Termination of Algorithm 6 is guaranteed: we limit the computation to one period of the task graph; the number of bus accesses is bounded which implies that the amount of interference seen by a task is also bounded. The response time computation of task τ_i is a monotonically increasing and bounded function, thus Algorithm 6 converges for any values in Θ. Termination of ComputeRT is non-trivial to show: the intuition is that a task cannot interfere with its past. At each iteration, release dates of tasks released before some instant of time t become fixed and remain the same for all subsequent iterations, with t advancing by at least one release date at each iteration. Note that this means the number of iterations of ComputeRT is at most $(|\Gamma| - 1)$, where $|\Gamma|$ is the task set's size. The complete proof is given in Section 5.3.2.

Since ComputeRT is parameterized by a function InitRelease, one might think that the choice of InitRelease could impact the precision of the result. However, we prove in Section 5.3.3 that the fixed point of the composition of MultiCoreRTA and UpdateReleases is unique, hence the algorithm will return the same schedule for any function InitRelease, and there is no point trying to optimize it. In our implementation, we start with $\Theta^0 = \Theta_{\text{min}}$.

5.3 TERMINATION AND CORRECTNESS OF THE RESPONSE TIME ANALYSIS

In this section we prove the convergence of ComputeRT in Algorithm 8. The algorithm uses classical Kleene iterations to find the fixed point of the composition f of MultiCoreRTA and UpdateReleases. In other words, function f is the body of the do/while loop of

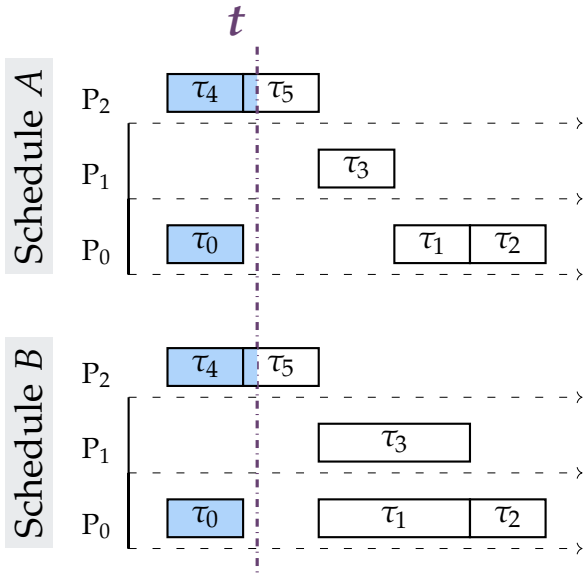


Figure 5.4: Illustration of the coincidence

ComputeRT. It takes and returns both response times and release dates, denoted by the set $\sigma^l = \{(rel_i, R_i) \mid \tau_i \in \Gamma\}$ at the l^{th} iteration (in practice, f only depends on release dates). We also use the notations $\sigma_a, \sigma_b, \ldots$ to denote any different sets of release dates and response times representing the same task set Γ. The algorithm terminates when f does not change Θ: when this happens, $\mathcal{R}$ is not modified either, so the algorithm actually finds a fixed point of f. The iteration is initialized with σ^0 which has release dates given by $\Theta_{\min}$, and computes the sequence $\sigma^n = f^n(\sigma^0)$.

The main idea of the proof is:

1. Once the prefix of a schedule does not change from one iteration to the next, it will not change later in the iteration. In other words, when all the release dates earlier than some date t are the same in σ^n and σ^{n+1}, they will remain the same in all $\sigma^k, k > n$.

2. When all the release dates earlier than some date t are the same in σ^n and σ^{n+1}, the first task(s) released after t in σ^{n+1} find their final release dates, i.e., they will not change from σ^{n+1} to σ^{n+2}.

In other words, the earliest release dates will not change after the first iteration, and the set of release dates that find their final value propagates from $t = 0$ onwards, until all release dates are final.

5.3.1 *Basic Properties of the Response Time Analysis*

To formalize the proof, we first need to define the notion of a "prefix of a schedule does not change". We define the coincidence operator $\simeq_{<t}$ to illustrate this notion given in Definition 6.

Definition 6. *Let σ_a, σ_b be two sets of pairs of release dates and response times representing the same task set $\Gamma = \{\tau_0, ..., \tau_n\}$ such that: $\sigma_a = \{(\text{rel}_0^{\sigma_a}, R_0^{\sigma_a}), ..., (\text{rel}_n^{\sigma_a}, R_n^{\sigma_a})\}$, $\sigma_b = \{(\text{rel}_0^{\sigma_b}, R_0^{\sigma_b}), ..., (\text{rel}_n^{\sigma_b}, R_n^{\sigma_b})\}$. Let $t \geq 0$ be an instant in time. σ_a coincides with σ_b before the instant t (denoted by $\sigma_a \simeq_{<t} \sigma_b$) iff:*

$$\forall i : \text{rel}_i^{\sigma_a} < t \vee \text{rel}_i^{\sigma_b} < t \Rightarrow \text{rel}_i^{\sigma_a} = \text{rel}_i^{\sigma_b}$$

Accordingly, $\sigma_a \not\simeq_{<t} \sigma_b$ iff:

$$\exists i : \text{rel}_i^{\sigma_a} < t \vee \text{rel}_i^{\sigma_b} < t \text{ and } \text{rel}_i^{\sigma_a} \neq \text{rel}_i^{\sigma_b}$$

Figure 5.4 illustrates the coincidence on two schedules of the data-flow in Figure 5.2a. Schedule A (resp. B) is a schedule obtained from n iterations of function f and represented by the set of pairs of release dates and response times σ_a (resp. σ_b). We first assume that A and B are different and later prove that they converge to the same schedule after a certain number of iterations of f. In Schedule A, tasks are executed such that the interference is avoided. In Schedule B, tasks τ_1 and τ_3 mutually interfere.

All tasks released before the instant t (τ_0, τ_4, τ_5) in both schedules have the same release dates. In this case, we say that schedules A and B coincide before t.

To clarify the properties of our algorithms, we make the following observations.

Observation 1. *Tasks that are executed in disjoint time intervals do not mutually interfere. The response time is computed in Algorithm 6 using equation (5.2).*

The term $I^{BUS}(i, x, \mathcal{R}, \Theta)$ used in equation (5.2) gives the bus interference of tasks τ_j on the task of interest τ_i. $\forall \tau_j$ such that $\text{rel}_j > \text{rel}_i + R_i$, the interference of τ_j on τ_i is 0.

Observation 2. *Let σ_a, σ_b be two sets of pairs of release dates and response times representing the same task set $\Gamma = \{\tau_0, ..., \tau_n\}$. Function* MultiCoreRTA *is deterministic and depends only on:*

1. *WCET in isolation of each task ($wcet_i$)*
2. *Memory Demand of each task (MD_i), used internally in $I^{BUS}(i, x, \mathcal{R}, \Theta)$*
3. *Release dates of each task (Θ)*

Since $wcet_i$ and MD_i are constant parameters for each task in Γ, the only variables in σ_a and σ_b, when applying MultiCoreRTA *on them, are the release dates.*

We observe that tasks that finish before the instant t in schedules A and B have the same release dates (definition of coincidence) and also the same response times. This observation is generalized below.

Observation 3. *Let σ_a, σ_b be two sets of pairs of release dates and response times associated with the same task set $\Gamma = \{\tau_0, ..., \tau_n\}$. Let t be an instant of time such that $\sigma_a \simeq_{<t} \sigma_b$.*

Tasks that are released after t in σ_a and σ_b are executed in a disjoint time interval with the tasks that finish before t. For all tasks τ_i that verify $\forall \tau_i : \text{rel}_i^{\sigma_a} + R_i^{\sigma_a} < t$ we have:

1. $\text{rel}_i^{\sigma_a} = \text{rel}_i^{\sigma_b}$ *(Definition of $\sigma_a \simeq_{<t} \sigma_b$)*

2. *All tasks τ_j released after the instant t do not interfere with tasks τ_i (consequence of Observation 1)*

According to Observation 2 on the determinism of the function MULTICORERTA *(in which the interference computation I^{BUS} is an increasing function with respect to the response time), we get $R_i^{\sigma_a} = R_i^{\sigma_b}$. Similarly, for tasks τ_i that verify $\text{rel}_i^{\sigma_b} + R_i^{\sigma_b} < t$ we also have $R_i^{\sigma_a} = R_i^{\sigma_b}$.*

Let us consider again the schedules in Figure 5.4 where A and B coincide before t. When applying the function f to both schedules, the new schedules $A' = f(A)$ and $B' = f(B)$ still coincide before t. It means that the tasks released before t in both schedules do not change their release dates when applying f. Lemma 1 below formalizes this property.

Lemma 1. *Let σ_a^0, σ_b^0 be two initial sets of release dates and response times representing the same task set $\Gamma = \{\tau_0, ..., \tau_n\}$.*
Let $n > 0$, $\sigma_a = f^n(\sigma_a^0)$, $\sigma_b = f^n(\sigma_b^0)$ and $\exists t > 0$ such that $\sigma_a \simeq_{<t} \sigma_b$. We have:

$$\sigma_a \simeq_{<t} \sigma_b \Rightarrow f(\sigma_a) \simeq_{<t} f(\sigma_b)$$

Proof. We prove that $\forall \tau_i : rel_i^{f(\sigma_a)} < t \vee rel_i^{f(\sigma_b)} < t \Rightarrow rel_i^{f(\sigma_a)} = rel_i^{f(\sigma_b)}$

σ_a and σ_b(such that $\sigma_a \simeq_{<t} \sigma_b$) are obtained after the n^{th} iteration of f which implies that we already have $\forall \tau_i \in \Gamma, \forall \tau_k \in deps(\tau_i) : \forall x \in \{\sigma_a, \sigma_b\}, rel_i^x > rel_k^x + R_k^x$.
Let σ_{I_a} be the intermediate set obtained during the application of f on σ_a such that $\forall \tau_i \in \Gamma$ we have:

1. $R_i^{\sigma_{I_a}} = \text{MULTICORERTA}(\Theta^{\sigma_a})$ and $rel_i^{\sigma_{I_a}} = rel_i^{\sigma_a}$
2. $rel_i^{f(\sigma_a)} = \text{UPDATERELEASES}(\Theta^{\sigma_{I_a}}, \mathcal{R}^{\sigma_{I_a}})$ and $R_i^{f(\sigma_a)} = R_i^{\sigma_{I_a}}$

σ_{I_b} is the intermediate set obtained during the application of f on σ_b and defined similarly to σ_{I_a}.
We apply f on σ_a and σ_b:

- MULTICORERTA computes the response times from the set of release dates. Since the release dates are not modified from σ_a to σ_b, we have: (i) $\sigma_{I_a} \simeq_{<t} \sigma_{I_b}$ (ii) for any task τ_k such that: $rel_k^{\sigma_{I_a}} + R_k^{\sigma_{I_a}} < t$ or $rel_k^{\sigma_{I_b}} + R_k^{\sigma_{I_b}} < t$ we have $R_k^{\sigma_{I_a}} = R_k^{\sigma_{I_b}}$ (Observation 3).
- UPDATERELEASES depends on release dates and response times obtained from MULTICORERTA. From (i) and (ii) we obtain $rel_i^{f(\sigma_a)} = rel_i^{f(\sigma_b)}$ for all tasks τ_i that are released before t.

Therefore, we have $f(\sigma_a) \simeq_{<t} f(\sigma_b)$. □

The previous lemma shows that the coincidence property is stable with regard to f. Moreover, the first task(s) released after t are updated with the same release date when applying f. This means that the resulting schedules will coincide before a new instant $t' > t$, where at least one task is released between t and t' in both schedules. This is expressed in the following lemma.

Lemma 2. *Let σ_a^0, σ_b^0 be two initial sets of release dates and response times representing the same task set $\Gamma = \{\tau_0, ..., \tau_n\}$. Let ε be the smallest unit of time measurement.*
Let $n > 0 : \sigma_a = f^n(\sigma_a^0)$, $\sigma_b = f^n(\sigma_b^0)$ such that $\exists t > 0 : \sigma_b \simeq_{<t} \sigma_a$.
Let $t' = \min_{\mathrm{rel}_k^x > t, x \in \{f(\sigma_a), f(\sigma_b)\}} (\mathrm{rel}_k^x) + \varepsilon$.
We have $f(\sigma_a) \simeq_{<t'} f(\sigma_b)$ with $t' > t$ and at least one task released between t and t'.

Proof. We define $\sigma'_a = f(\sigma_a)$ and $\sigma'_b = f(\sigma_b)$. We prove that for $t' = \min_{rel_k^x > t, x \in \{\sigma'_a, \sigma'_b\}} (rel_k^x) + \varepsilon$ we have $\sigma'_a \simeq_{<t'} \sigma'_b$.
According to Lemma 1, since $\sigma_a \simeq_{<t} \sigma_b$ we have $\sigma'_a \simeq_{<t} \sigma'_b$. We denote by Γ_j the set of tasks whose release date equals to $\min_{rel_j^x > t, x \in \{\sigma'_a, \sigma'_b\}} (rel_j^x)$. $\tau_j \in \Gamma_j$ denotes one of the first tasks released after t in σ'_a or σ'_b.

The following scenario occurs during the application of f to compute σ'_a and σ'_b:

- All predecessors τ_i of tasks in Γ_j are released before t in σ'_a and σ'_b, hence $rel_i^{\sigma'_a} = rel_i^{\sigma'_b}$. Moreover, since tasks in Γ_j are the first tasks released after t, no task is released in the interval $[t, rel_j[$ in σ'_a nor σ'_b. Hence, MultiCoreRTA sets $R_i^{\sigma'_a} = R_i^{\sigma'_b}$ according to Observation 3.

- Therefore, we have $\forall \tau_i \in deps(\tau_j) : rel_i^{\sigma'_a} = rel_i^{\sigma'_b}$ and $R_i^{\sigma'_a} = R_i^{\sigma'_b}$, hence UpdateReleases sets $rel_j^{\sigma'_a}$ and $rel_j^{\sigma'_b}$ to the same value (Algorithm 7, Line 3).

For all tasks τ_i such that $rel^{\sigma'_a} < t$ and $rel^{\sigma'_b} < t$ we have $rel^{\sigma'_a} = rel^{\sigma'_b}$. Furthermore, all tasks τ_j that are released at $t' = \min_{rel_j^x > t, x \in \{\sigma'_a, \sigma'_b\}} (rel_j^x) + \varepsilon$, have $rel_j^{\sigma'_a} = rel_j^{\sigma'_b}$. Therefore, we have $\sigma'_a \simeq_{<t'} \sigma'_b$. □

The above lemmas offer the necessary tools to conduct the actual proof of convergence of Algorithm 8.

5.3.2 *Convergence of the Fixed-Point*

To conduct the proof of convergence, we need to prove that (i) when all the release dates earlier than some date t are the same between two successive iterations of ComputeRT, they will remain the same in the next iterations (Lemma 1), (ii) at each iteration at least one task finds its final release date (Lemma 2). The actual proof then applies these two lemmas to build the sequence t_n such that the prefix of σ^n up to t_n does not change after iteration n (Theorem 1).

To illustrate our approach, we run Algorithm 8 on the program in Figure 5.2 (page 71) with $\Theta_{\min} = \{0, \ldots, 0\}$. Figure 5.5 shows the result of each iteration. We define the set of release dates and response times σ^l for each iteration l such that $\sigma^{l+1} = f(\sigma^l)$. In this example, tasks τ_0 and τ_4 do not have any dependencies therefore their release dates are fixed to the values from $\Theta_{\min}$ (0 in this case). We observe that $\sigma^1 \simeq_{<t_0} \sigma^2$ for $t_0 = rel_0 + \varepsilon$ as illustrated in Figure 5.5 (in cycle-based measurements, we assume that $\varepsilon = 1$ cycle).

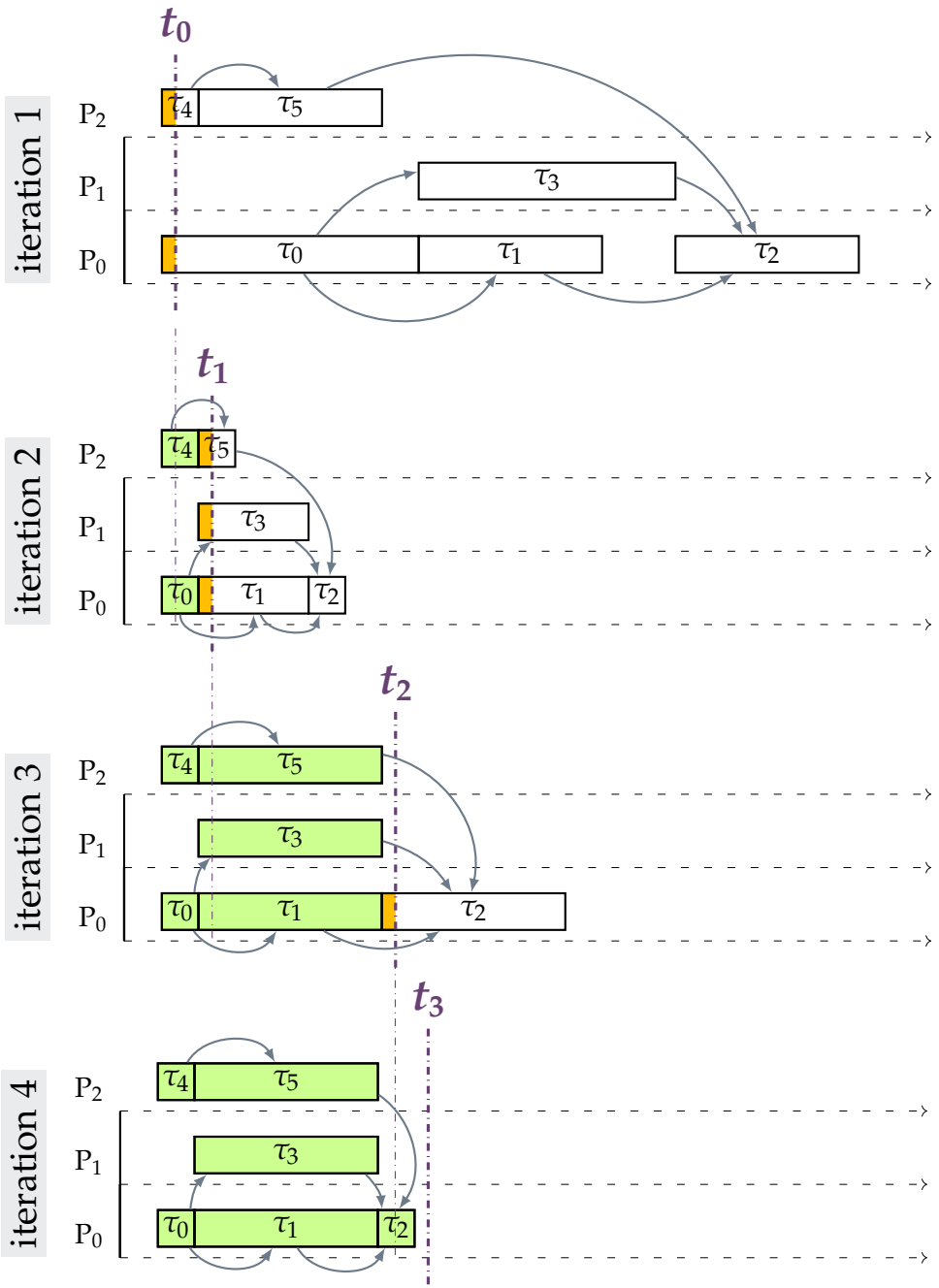


Figure 5.5: Execution of Algorithm 8 on the example in Figure 5.2 with $\Theta_{\min} = 0$. Arrows correspond to task dependencies. Tasks in green have fixed release dates and response times. Tasks in orange have only fixed release dates

According to Lemma 1, we know that tasks τ_0 and τ_4 will not change their release dates in the future iterations.

We have $\sigma^1 \simeq_{<t_0} \sigma^2$. At the next iteration of f, Lemma 2 gives $t_1 > t_0$ such that $\sigma^2 \simeq_{<t_1} \sigma^3$ and $t_1 = rel_1 + \varepsilon$. As shown in Figure 5.5, tasks τ_1, τ_3, and τ_5 are released before t_1 and

therefore fix their release dates. On the other hand, τ_0 and τ_4 finish before t_1 and according to Lemma 1, their response times remain fixed for the next iterations.

Similarly, we have $\sigma^3 \simeq_{<t_2} \sigma^4$ where τ_1, τ_3, and τ_5 finish before t_2 and hence fix their response times. Applying another iteration of f on σ^4 results in the same schedule which means that a fixed-point solution is reached.

From the example above, we derive the generalization of the convergence demonstration in Theorem 1 to any dependent task set Γ.

Theorem 1. ***(Convergence)***
The response time computation in Algorithm 8 converges and the fixed-point is reached in at most $\mid \Gamma \mid -1$ *loop iterations:*

$$\forall \sigma^0, \exists N \leq \mid \Gamma \mid -1, \forall n > N : f^{n+1}(\sigma^0) = f^n(\sigma^0)$$

Proof. Let σ^0 be any initial set of pairs of response times and release dates. To prove the convergence we prove the following: There exists a finite sequence of increasing instants $\{t_n | \exists N > 0, \forall n < N, t_{n+1} > t_n\}$ such that:

$$\forall n \geq 1, f^{n+1}(\sigma^0) \simeq_{<t_n} f^n(\sigma^0)$$

and there is at least one task released at $t_n - \varepsilon$ in $f^n(\sigma^0)$.

Base case: n=1

$\forall \sigma^0$ we have, at least for $t_0 = 0$, $f(\sigma^0) \simeq_{<t_0} \sigma^0$ that holds. By applying Lemma 2 we obtain $f^2(\sigma^0) \simeq_{<t_1} f^1(\sigma^0)$ for $t_1 = \min_{rel_i^x > 0, x \in \{\sigma^1, \sigma^2\}}(rel_i^x) + \varepsilon$ meaning at least one task is released at $t_1 - \varepsilon$.

Induction step:

Let us assume $f^{n+1}(\sigma^0) \simeq_{<t_n} f^n(\sigma^0)$ and prove

$$\exists t_{n+1} > t_n : f^{n+2}(\sigma^0) \simeq_{<t_{n+1}} f^{n+1}(\sigma^0)$$

and there is at least one task released at $t_{n+1} - \varepsilon$ in $f^{n+1}(\sigma^0)$.

By applying Lemma 2, $f^{n+2}(\sigma^0) \simeq_{<t_{n+1}} f^{n+1}(\sigma^0)$ for $t_{n+1} = \min_{rel_i^x > t_n, x \in \{\sigma^1, \sigma^2\}}(rel_i^x) + \varepsilon$ with at least one task is released at $t_{n+1} - \varepsilon$

Since the number of tasks $\mid \Gamma \mid$ is bounded, the fixed-point loop is also bounded by $\mid \Gamma \mid -1$. Therefore, given a set of tasks and a set Θ_{min}, Algorithm 8 converges in at most $\mid \Gamma \mid -1$ iterations. □

5.3.3 *Uniqueness of the Fixed-Point*

For the same dependent task set, there may exist many valid scheduling that respect the dependency relation between tasks. In fact, both schedules in Figure 5.4 are valid and both correctly accounts for the interference. By applying f on schedule A, the release date of task τ_1 is updated to the same value of τ_3. This results in a schedule $f(A) = B$, whereas $f(B) = B$ which means that schedule B is a fixed-point. Algorithm 8 converges to a unique fixed-point which does not depend on initial release dates. This is expressed in Theorem 2.

Theorem 2. ***(Uniqueness)***
For any initial values of release dates (set at Algorithm 8, line 3) Algorithm 8 results in the same release dates and response times for a given Θ_{min}. Let $\mathcal{F}_a$ and $\mathcal{F}_b$ be two fixed points representing the task set Γ:

$$\mathcal{F}_a = f(\mathcal{F}_a) \text{ and } \mathcal{F}_b = f(\mathcal{F}_b) \Rightarrow \mathcal{F}_a = \mathcal{F}_b$$

Proof. Let σ_a^0 and σ_b^0 be any initial sets of pairs of response times and release dates representing Γ. Let $\mathcal{F}_a = f^N(\sigma_a^0)$ and $\mathcal{F}_b = f^{N'}(\sigma_b^0)$ $(N, N' > 0)$ be two fixed points, i.e., $f(\mathcal{F}_a) = \mathcal{F}_a$ and $f(\mathcal{F}_b) = \mathcal{F}_b$. We prove that $\mathcal{F}_a = \mathcal{F}_b$.
By contradiction:
$\mathcal{F}_a$ and $\mathcal{F}_b$ are fixed-points. We assume: $\exists \sigma_a^0, \exists \sigma_b^0 : \mathcal{F}_a \neq \mathcal{F}_b$. This implies either:

1. $\exists i : R_i^{\mathcal{F}_a} \neq R_i^{\mathcal{F}_b} \wedge \forall k : rel_k^{\mathcal{F}_a} = rel_k^{\mathcal{F}_b}$
2. $\exists i : rel_i^{\mathcal{F}_a} \neq rel_i^{\mathcal{F}_b}$

- Case 1 is impossible according to Observation 3.
- Case 2 implies that at the instant t of the earliest release that differs in $\mathcal{F}_a$ and $\mathcal{F}_b$ $(t = \min_{\forall \tau_i \in \Gamma, x \in \{\mathcal{F}_a, \mathcal{F}_b\}} \{rel_i^x \mid rel_i^{\mathcal{F}_a} \neq rel_i^{\mathcal{F}_b}\})$ we have $\mathcal{F}_a \simeq_{<t} \mathcal{F}_b$ and $\forall t' > t : \mathcal{F}_a \not\simeq_{<t'} \mathcal{F}_b$.

  When applying f, UPDATERELEASES sets the first release date after t in $\mathcal{F}_a$ and $\mathcal{F}_b$ to the same value and $\mathcal{F}_a \simeq_{<t''} \mathcal{F}_b$ $(t'' > t)$ (according to Lemma 2) which contradicts our assumption.

□

According to Theorem 1 and Theorem 2, Algorithm 8 converges to a unique solution.

5.4 CONCLUSION

We present in this chapter our approach for response time analysis of dependent task graphs running on a multi-core, multi-resource architecture. Our approach accounts for the interference on shared buses from co-runners and computes a static schedule such that the dependencies are guaranteed. Our analysis is performed with a double fixed-point algorithm. First, a response time analysis is performed using an initial schedule and a set profiles (WCET and a worst-case number of accesses) of tasks when running in isolation, i.e., without any interference from co-runners. The profiles can be obtained from a static timing analysis tool on single-core platforms, or with measurement-based methods assuming a certain confidence degree. The response time formula is a recursive function that also requires a fixed-point algorithm. The fixed-point solution is reached if and only if the interference is monotonic and bounded. The second step is to verify the task dependencies and eventually update tasks' release dates that overrun on each other according to their dependency relation. The process of response time analysis and release date updates is repeated until a stable set of response times and release dates is found.

Our approach offers a more scalable way to analyze response times on multi-cores. The result is precise in the sense that it accounts only for the interference from the concurrent tasks running during the same time interval as the task under analysis. This method is still pessimistic; two tasks running at the same time do not necessarily interfere. The work in [DNA15] uses a notion of *sampling regions* to reduce this pessimism. The idea is to carry out the analysis using profiles of temporal regions (intervals) in the execution of tasks. The profiles indicate the distribution of shared resources accesses, and therefore establish a more precise interference analysis. Although this method may (theoretically) lead to a better analysis, it is too complex to implement, for hard real-time systems, in a real evaluation using existing static tool analysis.

Our approach takes into account the execution model that decouples the computations and communication phases. In fact, existing work [Mel+15; Bec+16] propose scheduling policies to reduce or eliminate the interference in the communication phases. Our approach is able to take into account the interference on the communications which may lead into a better overall response time of the task under analysis.

The interference on shared resources relies on the arbitration policy used in the architecture. In this chapter we considered a function that gives an upper-bound on the interference that may occur during the execution of the task under analysis. The implementation of this function depends on the considered platform. In the next chapter, we propose a model of shared resource accesses taking into account the arbitration policy of a real industrial platform.

Chapter 6

SHARED RESOURCE INTERFERENCE ANALYSIS ON A MANY-CORE PROCESSOR

In the previous chapter we presented our analysis framework assuming known upper bounds on interference. In this chapter, we detail how this upper bound is computed with regard to a specific platform. We apply our approach to the industrial many-core Kalray-MPPA 256 which corresponds to our target architecture model. This chapter is organized as follows: We present Kalray's many-core in Section 6.1. In Section 6.2, we identify sources of interference on shared resources. The upper-bound functions on interference are given in 6.3. We present a timing model of the arbitration policy in Sections 6.4 and 6.5. We discuss the timing compositionality of the platform in Section 6.6. Finally, the conclusion is in Section 6.7.

6.1 Presentation of the Kalray MPPA-256 Bostan

The Kalray MPPA-256 is a many-core processor [Din+14b]. Figure 6.1 shows an overview of this processor. It is composed of 16 tiles (called compute clusters) of 17 cores: 16 cores

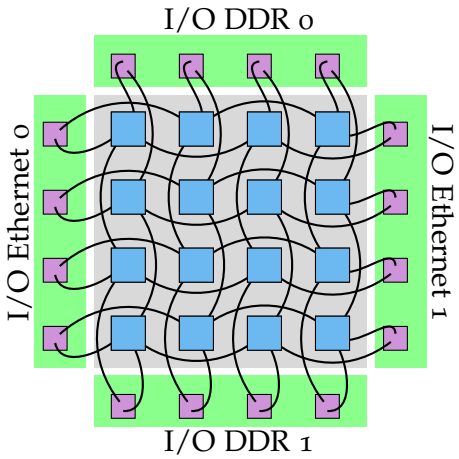


Figure 6.1: Overview of the Kalray MPPA-256

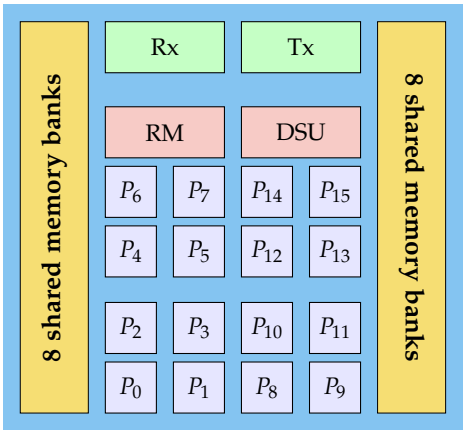


Figure 6.2: Compute cluster architecture for the Kalray MPPA-256

for processing and 1 core for resource management. The processor is connected to the external environment through two I/O quad-core clusters. Inter-cluster communication is achieved via a 2D-torus dual Network-On-Chip (NoC) for data and control. In this paper, we are interested in applications running on a compute cluster and the interference due to intra-cluster communications.

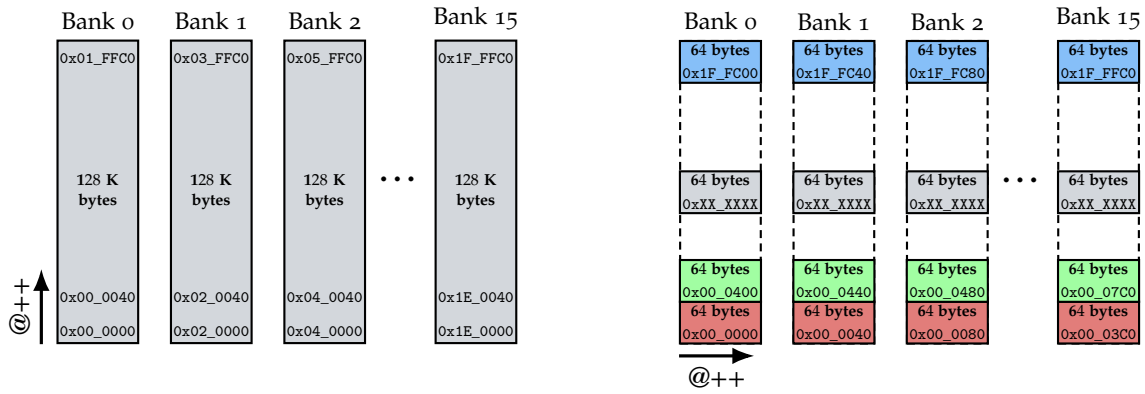
6.1.1 *Compute Cluster*

Figure 6.2 illustrates the architecture of a single compute cluster. It has 16 cores plus 1 Resource Manager (RM). The compute cluster connects to the NoC via two DMA (Direct Memory Access) interfaces; one for receiving (Rx) and one for transmitting (Tx). The cluster also has a Debug Support Unit (DSU).

Cores have an in-order Very Long Instruction Word (VLIW) pipeline and separate 8 kB 2-way set-associative private caches with 64 B lines for instructions and data. The data cache has a write buffer (WB) with 8 fully associative 64 bit entries. There is no cache coherency mechanism between the cores. Each core has its own real-time clock. Clocks in the same cluster are synchronous.

6.1.2 *Shared Memory*

In order to provide spatial isolation, the shared memory (SMEM) is partitioned into 16 banks. Each memory bank is accessed via a separate bus arbiter which significantly reduces the amount of interference compared to the alternative of a single arbiter. Figure 6.3 shows two possible configurations for the memory banks: *interleaved mode* (Figure 6.3a) where sequential memory addresses move from one bank to another, and *blocked mode* (Figure 6.3b) where each block of 128 kB consecutive memory addresses are contained in a memory bank. In this paper, we assume that blocked mode is selected, since it gives more control over the bus interference. This is because with blocked mode, cores that access different memory banks go through different arbiters hence they do not interfere with each other. We use



(a) Blocked address mode

(b) Interleaved address mode

Figure 6.3: Blocked and interleaved address configuration modes of the shared memory (SMEM)

a fixed association between cores and memory banks. More precisely, in the application model we consider, each task has a local memory buffer, and the buffers of all tasks running on the same core are mapped to the same memory bank. Tasks read and write from the local memory bank of their assigned core, while communications are achieved by writing data to other remote memory banks. Note that, when tasks are assigned to the same core, they use the same memory bank. Thus, the communication between these tasks happen on the same memory bank.

6.1.3 *Bus Arbitration*

Figure 6.4 illustrates the specific multi-level policy used to arbitrate accesses to the shared memory. We distinguish three groups which are arbitrated over three levels:

- $G1 = \{i \in [0,15] : P_i\}$: access requests from the 16 cores are first subject to round-robin arbitration.
- $G2 = \{Tx, DSU, Rm\}$: access requests from the Resource Manager (RM), Debug Support Unit (DSU) and Tx requests to the NoC are subject to round-robin arbitration.
- $G3 = \{Rx\}$: Rx requests from the NoC.

At level $L1$ (1), requests issued by data and instruction caches local to a core are processed by a local round-robin arbiter. At level $L2$ (2), there is a round-robin arbitration within each of the groups $G1$ and $G2$. This is followed by round-robin arbitration between these two groups at level $L3$ (3). Finally, $G3$ is included in the last level of the arbitration $L4$ (4), which uses a non-preemptive fixed-priority arbiter and gives the highest priority to access requests coming from $G3$. Note that $G3$ (where Rx requests arrive) has the highest priority to offload the NoC and avoid its congestion.

To summarize, an access request from a task running on a core crosses three levels of round-robin arbitration and a level of fixed-priority arbitration to reach the shared memory.

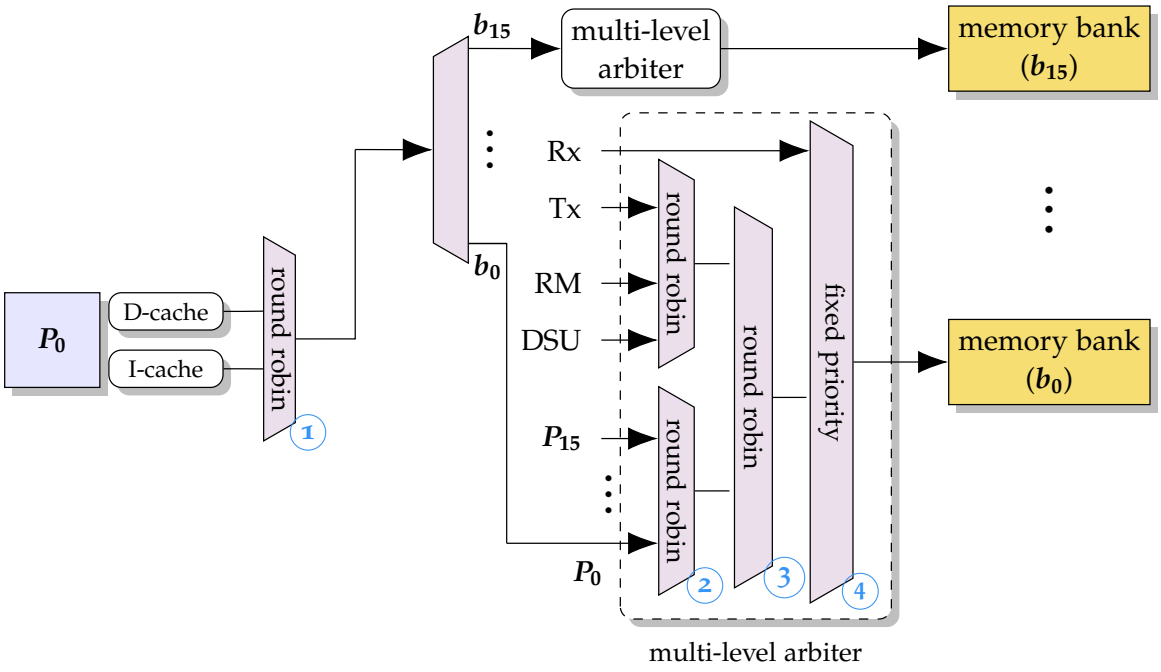


Figure 6.4: Request arbitration to a shared memory bank

6.2 TIMING ANALYSIS ON THE KALRAY MPPA-256

In this section we list the different sources of interference that need to be considered in analysing synchronous data-flow applications running on a compute cluster of the Kalray MPPA-256. This helps to understand how bus interference can be computed using the task dependency graph for a synchronous data-flow program, thus avoiding pessimism in the analysis caused by lack of information about co-runners. Sources of interference have to be identified as part of Equation 5.1 (page 73) when determining the response time R_i of task τ_i, given the hardware and application models considered.

The interference on the core $I^{\text{PROC}}(i, x, R_i)$ typically comes from delays or preemptions due to the execution of higher priority tasks on the same core. In our application model, we assume a static non-preemptive scheduler. Task release dates are set such that only one task is active per core at any given time. This effectively eliminates all interference from higher priority tasks executing on the same core. It also simplifies the analysis by removing any *cache-related preemption delays* [ADM12].

The interference due to the DRAM is mainly due to refresh cycles. The Kalray MPPA-256 supports a DDR memory accessed through the I/O clusters. An access from a core in a compute cluster has to cross the NoC and the I/O cluster and finally the DDR controller. All these layers add to the complexity of the analysis and the access delay. For a predictable operation, such accesses are generally avoided by pre-loading all the code and data into the shared memory of the compute cluster. The on-chip RAM of the MPPA is 32 MB, 2 MB per cluster, which is sufficient for many applications. We therefore assume that $\forall t > 0, I^{\text{DRAM}}(i, x, t) = 0$.

The interference on the bus depends on the specific arbitration policy used. Cache misses in the private data and instruction caches issue requests to the shared memory that are granted according to the multi-level arbiter. A detailed derivation of the $I^{\text{BUS}}(i, x, \mathcal{R}, \Theta)$ function that depends on the set of release dates of all tasks is given in the following section.

Taking the above considerations into account, the response time formula given in (5.1) simplifies to:

$$R_i = \text{PD}_i + I^{\text{BUS}}(i, x, \mathcal{R}, \Theta) \tag{6.1}$$

Note, here $\mathcal{R}$ is the set of response times and Θ is the set of release dates for all tasks.

According to the considered application model, each compute cluster executes an SDF application at a time. Nevertheless, we assume that tasks can initiate inter-cluster communications through the NoC. In this case, a DMA transfer is set up asynchronously to remote SMEMs. We assume that tasks do not wait for an acknowledgment signal from the remote cluster (as in handshaking protocols), and therefore NoC delays do not add up to the task execution (Equation 6.1).

6.3 SHARED BUS INTERFERENCE

In our application model, we consider a task dependency graph mapped to a set of cores. The hardware architecture allows the mapping of contiguous addresses to the same memory bank. Thus, concurrent accesses are independent as long as they are done to different memory banks, which reduces the bus interference. We exploit this by allocating the memory of each task running on the same core to the same bank. Tasks run on their locally reserved memory banks and access other locations only when writing data to the next successive task(s) in the task graph. We denote by MD_i^b the memory demand (number of accesses) of task τ_i on memory bank b.

6.3.1 *Understanding Memory Accesses*

Here we describe how memory accesses are issued. Using illustrative examples of different scenarios, we show how read and write accesses delays the execution. Finally, we determine the parameters needed to model the bus arbiter.

Each core has separate private caches for data and instructions. Cache misses results in accesses to the shared memory. On the other hand, other instructions bypasses the caches such as uncached `load` and `store` instructions.

6.3.1.1 *Cached Memory Accesses*

Figure 6.5 describes memory access patterns generated from the data cache. Shared memory accesses latency depends on their type (read or write accesses) and the state of the cache and the write buffer.

The data cache is 2-way set-associative and has 8 kB with 64 B lines. It implements a *least-recently-used* (LRU) replacement policy. It is complemented with a write buffer containing

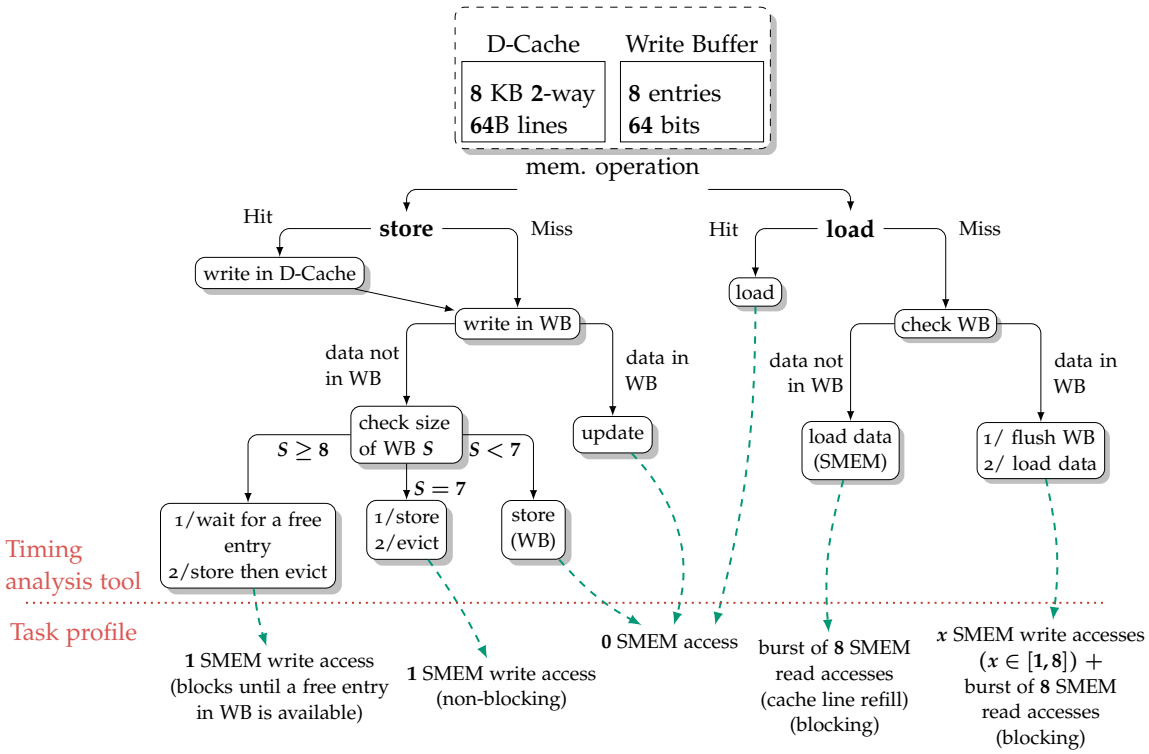


Figure 6.5: Occurrence of accesses from data cache and write butter to shared memory

8 fully associative 64 bit entries also implementing an LRU policy. The write buffer implements a *write merge*, i.e., if the written data already exists in the write buffer, it is simply updated. If the data does not exist in the write buffer, it is written in a free entry. Upon writing in the 8^{th} entry in the write buffer, the LRU entry is then evicted such as there is always an empty entry.

When a `store` instruction is executed: if it hits in the cache, the data is updated and written to the write buffer. If it is a miss, the data is written to the write buffer only. An access to the shared memory may be issued depending on the content of the write buffer. `store` instructions are normally asynchronous, however, in some cases (such as when the write buffer is already full), the core can be blocked and wait for a free entry in the write buffer.

When a `load` instruction hits in the cache, no shared memory accesses are issued. If the instruction misses in the cache, the write buffer is checked. If the requested data is not in the write buffer, it is simply requested from the SMEM. This results in a cache line re-fill which corresponds to a burst of 8 accesses. If the data is in the write buffer, it means that it was updated at some point and is not available in the SMEM yet. In this case, the write buffer needs to be flushed first then a cache line re-fill is performed. This corresponds to issuing individual accesses from the write buffer to write the data in the SMEM, followed by a burst of read accesses to refill the cache. The data cache implements a *critical-word-first* policy: a `load` access blocks the core's pipeline until the required data is present.

The core's pipeline may continue executing next instructions. The cache remains blocked until the end of the accesses burst. Any `load/store` is blocking all along this duration.

6.3.1.2 *Instruction Cache*

The instruction cache is 2-way set-associative with 64 B lines and a size of 8 kB. The instruction cache receives fetches from the core to a buffer called *pre-fetch buffer* (PFB). The cache returns a group of up to eight 64 bit words. When the requested instructions miss in the instruction cache, a burst of accesses is issued to the shared memory. The core is stalled during this time.

6.3.2 *Illustrative Examples on Cached Load and Store Instructions*

An instruction that hits in the cache takes 1 cycle to execute. On a cache miss, the amount of time the core stalls depends on the instruction that is executed after. A miss provokes a fetch of a full cache line from the shared memory issuing 8 memory accesses. The following example shows a load miss followed by a memory instruction that uses the same data.

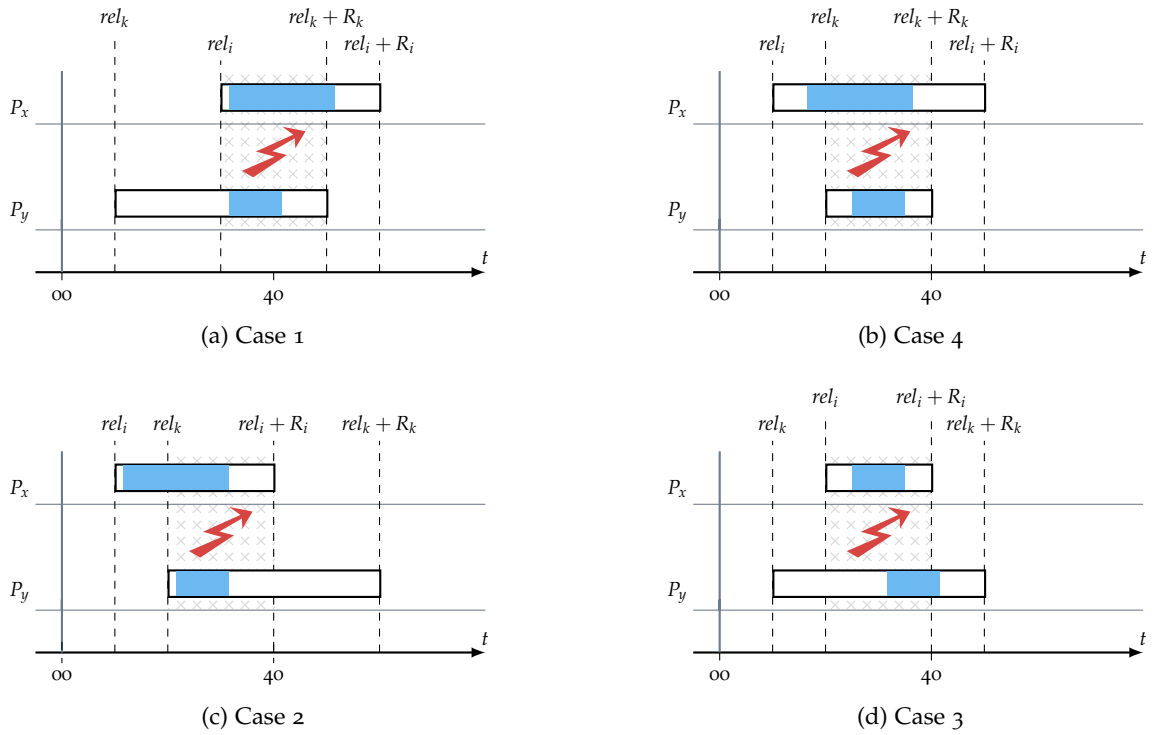
```
1 lw $r0 = 0[$r20] ## misses
2 sd 10[$r5] = $r6r7 ## stalls 17 cycles as L1D cache is busy
```

The accesses are pipelined which results in 10 cycles for the first access and the rest accesses arriving at the rate of 1 access per cycle. During this time, the *Load Store Unit* (LSU) is not available to execute another memory operation. The cache remains busy until the last data is received. This results in a total stall time of 17 cycles of the following `sd` (store double) operation.

Fetching cache lines operates in a *critical-word-first* way, i.e., the requested data will arrive first to the cache followed by the remaining elements in the cache line. If the requested data is used just after the load (which is the case in the example below), the stall time is 10 cycle for the first critical. The LSU remains unavailable for 7 more cycles but any other instruction that does not access the cache can be executed meanwhile. Example:

```
1  lw $r0 = 0[$r20] ## misses
2  add $r5 = $r0, $r2 ## stalls 11 cycles
3  xor $r5 = $r5, $r30 ## any instructions but loads/stores
4  sub $r50 = $r40, 35 ## any instructions but loads/stores
5  mul $r24 = $r51, $r8 ## any instructions but loads/stores
6  add $r50 = $r50, 3 ## any instructions but loads/stores
7  nop ## any instructions but loads/stores
8  nop ## any instructions but loads/stores
9  nop ## any instructions but loads/stores
10 lw $r28 = 0[$r31] ## granted by the D-Cache with no stall
```

The instruction `add` at Line 2 stalls for 11 cycles since it depends on the data in register `$r0` loaded at Line 1. Any instruction that does not request the cache can be executed without stalling. The second load instruction at Line 10 is granted without stall since it arrives after 17 cycles when the cache is available.



(a) Case 1

(b) Case 4

(c) Case 2

(d) Case 3

Figure 6.6: Cases of overlapping tasks

6.3.3 *Variables in Bus Interference Model*

We define the bus interference function introduced in Chapter 5 as the following:

$$I^{\text{BUS}}(i, x, \mathcal{R}, \Theta) = \sum_{b \in B_i} \text{BUS}_b(i, x, \mathcal{R}, \Theta) \times d \tag{6.2}$$

where d is the latency of a bus access without interference, B_i is the set of memory banks accessed by task τ_i, and $\text{BUS}_b(i, x, \mathcal{R}, \Theta)$ is a function that, accounting for the arbitration policy, gives an upper bound on the number of accesses to bank b that can delay completion of task τ_i (running on core P_x) during a given time interval.

In order to derive $\text{BUS}_b(i, x, \mathcal{R}, \Theta)$, we need to compute an upper bound on all bus accesses during the response time of task τ_i. We define $S_i^{x,b}(\mathcal{R})$ as an upper bound on the number of accesses to bank b by the task of interest τ_i running on core P_x within its response time. Note that since the scheduler is non-preemptive the bus accesses from core P_x come only from the memory demand of task τ_i on the memory bank b (MD_i^b). Since we analyze one instance of task τ_i, we have:

$$S_i^{x,b}(\mathcal{R}) = MD_i^b \tag{6.3}$$

We define $\underset{i,k}{\Delta}(\mathcal{R},\Theta)$ the overlap duration between tasks τ_i and τ_k. The computation of the overlap is trivial given the release dates and response times of tasks τ_i and τ_k as shown in Figure 6.6. Note that $\underset{i,k}{\Delta}(\mathcal{R},\Theta)$ is 0 when the tasks do not overlap.

$$\underset{i,k}{\Delta}(\mathcal{R},\Theta) = \begin{cases} \min(R_k, rel_k + R_k - rel_i) & \text{if } (rel_i \leq rel_k + R_k \leq rel_i + R_i) \text{ (Figures 6.6a,6.6b)} \\ \min(R_i, rel_i + R_i - rel_k) & \text{if } (rel_k \leq rel_i + R_i \leq rel_k + R_k) \text{ (Figures 6.6c,6.6d)} \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

We use $\underset{i,k}{W}^b(\mathcal{R},\Theta)$ to denote an upper bound on the number of accesses by task τ_k that may interfere with task τ_i at the memory bank b during its response time. In the absence of detailed information on the pattern of access requests within a task, we consider that any two tasks that overlap in time can interfere on each of their accesses. Moreover, we consider that the maximum number of accesses occur during this overlap. Let d_{min} be the minimum duration between successive non-blocking accesses. $\underset{i,k}{W}^b(\mathcal{R},\Theta)$ is given by:

$$\underset{i,k}{W}^b(\mathcal{R},\Theta) = \min(MD_k^b, \left\lceil \frac{\underset{i,k}{\Delta}(\mathcal{R},\Theta)}{d_{min}} \right\rceil) \tag{6.5}$$

We use $A_i^{y,b}(\mathcal{R},\Theta)$ to denote an upper bound on the number of accesses by all tasks running on core $P_y \neq P_x$ during the response time of task τ_i. The number of accesses is bounded by the memory demand of each task on memory bank b. $A_i^{y,b}(\mathcal{R},\Theta)$ is therefore given by:

$$A_i^{y,b}(\mathcal{R},\Theta) = \sum_{k \in \Gamma_y} \underset{i,k}{W}^b(\mathcal{R},\Theta) \tag{6.6}$$

The terms $S_i^{x,b}(\mathcal{R})$ and $A_i^{y,b}(\mathcal{R},\Theta)$ are used to derive an upper bound on the number of accesses that contribute to the interference during the response time of task τ_i, which also depends on the bus arbitration policy. The multi-level arbiter of the Kalray MPPA-256 requires a combination of several policies (see Figure 6.4). In the following, we give a mathematical model of the multi-level bus using the terms $S_i^{x,b}(\mathcal{R})$ and $A_i^{y,b}(\mathcal{R},\Theta)$.

6.4 SIMPLIFIED MODEL OF THE MULTI-LEVEL BUS ARBITER

In this section, we study the multi-level arbitration policy used in the Kalray MPPA-256 architecture. For simplicity, the model constructed here focuses on the multi-level arbitration policy without detailing other aspects of the shared bus (such as memory pipelining). This results in a safe but over-approximated upper bound on the shared bus interference. More details regarding hardware aspects are added later to complete this model.

We consider a bus arbiter to a memory bank b as shown in Figure 6.4 (page 88). The policy operates over 4 levels which we label $L1$ to $L4$ where $L1$ is the first (left-most) level,

and $L4$ the final level which is based on fixed-priority arbitration. Our analysis is built up following the hierarchy from level $L1$ to level $L4$.

Level $L1$: As input to the first level, we assume that the maximum number of accesses generated by each source in the response time of a task can be determined. These values are as follows:

- First group (G1): this is a core and may be treated in the same way as the analysis given for a round-robin arbiter in [Alt+15]. Note that we do not need to distinguish between accesses that come via the Instruction Cache (IC) and those that come via the Data Cache (DC), since all must be processed before the task of interest τ_i can complete. Hence, we may represent the output from this group as either $S_i^{x,b}(\mathcal{R})$ or $A_i^{y,b}(\mathcal{R},\Theta)$ depending on whether we are computing the accesses from the core that τ_i executes on, or from another core.
- Second group (G2): here we only need to compute the overall output from the group: $A_i^{G2,b}(\mathcal{R},\Theta) = A_i^{Tx,b}(\mathcal{R},\Theta) + A_i^{DSU,b}(\mathcal{R},\Theta) + A_i^{RM,b}(\mathcal{R},\Theta)$, since we are only interested in the interference it generates.
- Third group (G3): there is only one item, hence the output is the same as the input: $A_i^{G3,b}(\mathcal{R},\Theta) = A_i^{Rx,b}(\mathcal{R},\Theta)$.

Level $L2$: At level $L2$ the outputs (accesses) from all 16 processors are combined via a 16 to 1 round-robin arbiter. Note that each core has only one slot in the round-robin cycle. The number of accesses to bank b that can delay the execution of a task on core P_x at the output of $L2$ is given by:

$$\text{BUS}_b^{L2}(i,x,\mathcal{R},\Theta) = \sum_{y\in G1 \wedge y\neq x} \min\left(A_i^{y,b}(\mathcal{R},\Theta), S_i^{x,b}(\mathcal{R})\right) \tag{6.7}$$

where x is the index of the core P_x that task τ_i executes on, and similarly y ranges over the other 15 cores.

The worst-case situation occurs when each access in $S_i^{x,b}$ is delayed by each core $P_y \neq P_x$ for 1 slot. Given the round-robin arbiter, interference by core P_y is limited to the minimum of the number of accesses from P_y and from P_x, i.e. $\min\left(A_i^{y,b}(\mathcal{R},\Theta), S_i^{x,b}(\mathcal{R})\right)$.

Level $L3$: At level $L3$, the output from the level $L2$ arbiter, i.e. Equation 6.7, is combined with that from the second group, i.e. $A^{G2,b}(\mathcal{R},\Theta)$, again via a round-robin arbiter. Here, interfering accesses from co-runners with the task of interest can also be interfered by accesses from $G2$. We illustrate this with the following example:

Example 9. *We consider a task running on core P_0 and executing 5 accesses to the shared memory. We also consider that co-runner tasks on P_1 and P_2 execute simultaneously 7 accesses each. Finally, we consider that the Tx executes 30 accesses. The worst case happens when the accesses from cores P_1 and P_2 are granted before P_0. The number of interfering accesses is $\min(7,5) = 5$ accesses from each core according to Equation 6.7. Each of these accesses can also be delayed at level L3 by one access from Tx (10 accesses from Tx). The remaining accesses from Tx can delay P_0, in this case, $\min(30,5) = 5$ accesses. Thus, the number of interfering accesses with the task of interest at L3 is: $\min\left(30, (5 + \min(5,7) + \min(5,7))\right) = 15$ accesses in addition to the 10 accesses from co-runners at L2.*

To formalize the scenario in Example 9, we define $\lambda(i, x, \mathcal{R}, \Theta)$ as the number of accesses of task τ_i and all accesses that may interfere with it at $L2$. This is given by:

$$\lambda(i, x, \mathcal{R}, \Theta) = S_i^{x,b}(\mathcal{R}) + \sum_{y \in G1 \wedge y \neq x} \min\left(A_i^{y,b}(\mathcal{R}, \Theta), S_i^{x,b}(\mathcal{R})\right) \tag{6.8}$$

The worst-case situation occurs when each access in $\lambda(i, x, \mathcal{R}, \Theta)$ is delayed by the output of $G2$ for 1 slot. Interference by the output of $G2$ is limited to $A_i^{G2,b}(\mathcal{R}, \Theta)$. Hence, the interference at $L3$ is:

$$\text{BUS}_b^{L3}(i, x, \mathcal{R}, \Theta) = \text{BUS}_b^{L2}(i, x, \mathcal{R}, \Theta) + \min\left(A_i^{G2,b}(\mathcal{R}, \Theta), \lambda(i, x, \mathcal{R}, \Theta)\right) \tag{6.9}$$

Level *L*4: Finally, at level $L4$, the output from the level $L3$ arbiter, i.e. (6.9), is combined with the output from $G3$, i.e. $A_i^{Rx,b}(\mathcal{R}, \Theta)$. As this is done via a fixed-priority arbiter with higher priority given to $A_i^{Rx}(\mathcal{R}, \Theta)$, we have:

$$\text{BUS}_b^{L4}(i, x, \mathcal{R}, \Theta) = \text{BUS}_b^{L3}(i, x, \mathcal{R}, \Theta) + A_i^{G3,b}(\mathcal{R}, \Theta) \tag{6.10}$$

Finally, considering d as a worst-case delay from one interfering access, the bus interference is given by:

$$I^{\text{BUS}}(i, x, \mathcal{R}, \Theta) = \sum_{b \in B_i} \text{BUS}_b^{L4}(i, x, \mathcal{R}, \Theta) \times d \tag{6.11}$$

Equation 6.11 is injected in Equation 5.2 (page 74). The latter is solved with fixed-point iterations. We observe that the final solution to Equation 5.2 always accounts for the worst-case interference for overlapping tasks.

6.5 FULL MODEL OF THE INTERFERENCE ON SHARED RESOURCES

Equation 6.11 gives an upper bound on the shared bus interference. This model is simple and abstracts many complicated features in the hardware. Although the obtained upper bound is more precise than simply adding a fixed pessimistic delay to all tasks, it remains very pessimistic with regard to the real task execution. In this section, we give the complete model of the shared resource accesses by exploiting the architecture's properties such as accesses' pipelining and bursts.

6.5.1 *Bursts of Memory Accesses*

As shown in Figure 6.5, some instructions, such as load misses, generate bursts of accesses. The bus arbiter grants accesses at the granularity of a burst. This means that once the burst is granted, all the accesses within the burst are executed without interruption. So far, we counted accesses individually which results in bursts executing in a time $n \times \alpha$, where n is the number of accesses in the burst and α is the time to execute one access. In reality, accesses in a burst are issued at each cycle thanks to the pipelining of shared memory accesses. A burst of n accesses is executed in a time $\alpha + (n - 1)$ cycles.

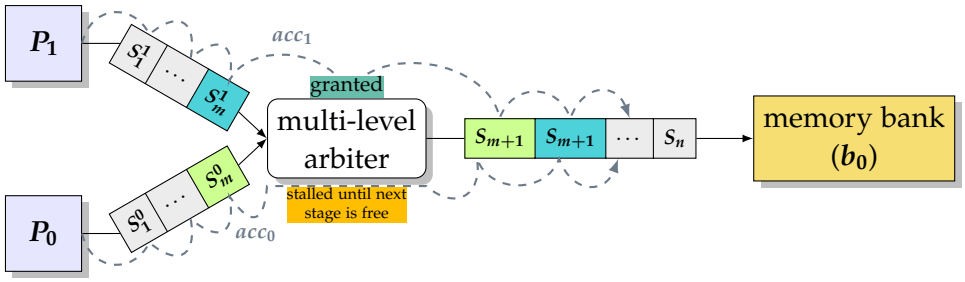


Figure 6.7: Shared bus pipeline

	t	$t+1$	$t+2$	...
S_1^0				
...				
S_m^0		stall		
S_{m+1}				
S_{m+2}				
...				

(a) Viewpoint of core P_0: progress of access acc_0 in the bus pipeline

	t	$t+1$	$t+2$	...
S_1^1				
...				
S_m^1				
S_{m+1}				
S_{m+2}				
...				

(b) Viewpoint of core P_1: progress of access acc_1 in the bus pipeline

Figure 6.8: Interference delay considering the shared bus pipeline

If a core requests the bus for a burst of accesses, the arbiter grants the request for all the burst at once. As a consequence, the interference depend on whether the task under analysis is delayed by an access or a burst. Bursts have different sizes (number of accesses) leading to a varying penalties on the task of interest.

The model presented in Section 6.4 abstracts the aspects described above and leads to pessimistic but conservative upper bound on concurrent accesses, providing that the bus latency used in the model considers the delay from bursts of access. To improve this model, we need to consider: (i) a notion of a burst of accesses (ii) blocking and non-blocking accesses.

6.5.2 *Memory Access Pipeline*

Existing work [Alt+15; Gia+16; SS16] assumes that bus arbiters stall concurrent accesses during the whole execution time of a granted access. This assumption is conservative but introduces more pessimism. In fact, accesses to the shared memory are pipelined through FIFOs and different stages on the bus. The pipeline reduces the critical path[1] between the core and the shared memory. At each clock cycle, an access moves by one stage across the pipeline.

Figure 6.7 illustrates the stages in the bus pipeline with the arbiter. Part of the pipeline is private to the core ($S_0^x, S_1^x, \ldots, S_m^x$ for core P_x), i.e., only the core's accesses traverse it. The

1 The longest path between two gates on the chip

other part $(S_{m+1}, \ldots, S_n)$ is shared and subject to the bus arbiter. We aim at demonstrating how much delay a concurrent access adds up to the execution time. Let req_0 and req_1 be two access requests issued concurrently from P_0 and P_1 respectively. In Figure 6.8, req_1 is granted access first and progresses to the next stage in the pipeline whereas req_0 is stalled. At the next cycle req_1 progresses to the following stage and req_0 is granted by the arbiter to progress in the empty stage left by req_1. If we consider single accesses, this results in a stall time of 1 cycle.

As a conclusion, a concurrent access may stall the task of interest for at most 1 cycle. A burst of n accesses granted at once may stall the task of interest for at most n cycles. In the general case, we define d_s as the *delay suffered from a single access* and d_t as *the delay suffered from a burst of accesses*.

6.5.3 *Blocking and Non-blocking Memory Accesses*

There are two types of accesses: blocking memory accesses and non-blocking memory accesses. For instance, `load` instructions block the core until the requested data is ready. Some `store` instructions on the other hand can be non-blocking, thanks to the write buffer, and the fact that the core does not block while waiting for the data to reach the SMEM. In some cases, when the write buffer is full for example, a `store` instruction blocks the core until a free entry is available.

A conservative approach is to consider all accesses blocking; however, if the analysis can prove that a certain number of accesses do not block the core, the estimated upper bound on the interference can be improved. Non-blocking accesses do not delay the task of interest, though they still interfere on concurrent tasks. Therefore, the analysis needs to know the total number of accesses and (if possible) the number blocking accesses of each task in the system. Otherwise, all accesses are considered blocking.

6.5.4 *Arbitration Policy*

From the above we determine that shared memory accesses can be granted individually or as bursts of accesses. The delay on shared resource accesses then varies depending on the types of interfering accesses. We introduce the notion of *transaction* to abstract this behavior.

Definition 7. *A transaction is a non-preemptive set of memory accesses granted at once by the shared bus arbiter. A transaction may have a single access or a burst of x accesses.*

Consequently to the properties of shared memory accesses, a transaction can be blocking or non-blocking.

Given the types of memory accesses above, we complete our model of the multi-level arbiter. In addition to the memory demand MD_i^b, this new model assumes a known upper bound on the number of blocking transactions $WCBT_i^b$ of task τ_i to memory bank b. MD_i^b counts all blocking and non-blocking accesses individually not taking into account the bursts. By definition, we have $WCBT_i^b \leq MD_i^b$.

Let $S_i'^{x,b}$ be an upper bound on the number of blocking transactions issued by task τ_i running on core P_x to memory bank b.

$$S_i'^{x,b} = WCBT_i^b \tag{6.12}$$

$A_i^{y,b}(\mathcal{R}, \Theta)$ is (as defined in Section 6.4) an upper bound on the number of accesses issued from co-runner tasks on core P_y ($y \neq x$) that may interfere with the task of interest τ_i.

We construct the model of the multi-level arbiter in the same way as in Section 6.4. As previously, we consider all accesses at the output of level $L1$ coming from the data cache, the write buffer, and the instruction cache.

A task can at most be delayed by concurrent accesses at each issued blocking transaction. Further, we consider the worst case of interference where a burst of accesses delays each blocking transaction issued by τ_i for d_t cycles. The interference suffered by the task of interest from a core (or bus requester) is upper bounded by $S_i'^{x,b} \times d_t$.

On the other hand, all accesses (blocking and non-blocking) from concurrent tasks may interfere with the task of interest with a delay d_s each. Therefore, the interference suffered by the task of interest from a core (or bus requester) is upper bounded by $A_i^{y,b}(\mathcal{R}, \Theta) \times d_s$.

Let $itf_b^X(i, x, \mathcal{R}, \Theta)$ ($X \in \{L2, L3, L4\}$) be an upper bound on the delay suffered by task τ_i when accessing memory bank b at bus arbiter level X. The upper-bound function of the interference from all cores at level $L2$ is:

$$itf_b^{L2}(i, x, \mathcal{R}, \Theta) = \sum_{y \in G1 \wedge y \neq x} \min\left(S_i'^{x,b} \times d_t \ , \ A_i^{y,b}(\mathcal{R}, \Theta) \times d_s\right) \tag{6.13}$$

The number of times task τ_i can be delayed at $L3$ is defined similarly to Equation 6.8 by:

$$\lambda'(i, x, \mathcal{R}, \Theta) = S_i'^{x,b} + \sum_{y \in G1 \wedge y \neq x} \min\left(S_i'^{x,b}, A_i^{y,b}(\mathcal{R}, \Theta)\right) \tag{6.14}$$

Note that $\lambda'(i, x, \mathcal{R}, \Theta)$ depends on $S_i'^{x,b}$ instead of $S_i^{x,b}$. In the worst case, each access in $\lambda'(i, x, \mathcal{R}, \Theta)$ is delayed by a burst of accesses. Similarly to Equation 6.9, the upper bound on the delay suffered by task τ_i at $L3$ is:

$$itf_b^{L3}(i, x, \mathcal{R}, \Theta) = itf_b^{L2}(i, x, \mathcal{R}, \Theta) + \min\left(\lambda'(i, x, \mathcal{R}, \Theta) \times d_t \ , \ A_i^{G2,b}(\mathcal{R}, \Theta) \times d_s\right) \tag{6.15}$$

Finally at level $L4$ and similarly to Equation 6.10, we add a penalty from the fixed-priority arbiter:

$$itf_b^{L4}(i, x, \mathcal{R}, \Theta) = itf_b^{L3}(i, x, \mathcal{R}, \Theta) + A_i^{G3,b}(\mathcal{R}, \Theta) \times d_s \tag{6.16}$$

The upper-bound function on bus interference is:

$$I^{\text{BUS}}(i, x, \mathcal{R}, \Theta) = \sum_{b \in \mathcal{B}} itf_b^{L4}(i, x, \mathcal{R}, \Theta) \tag{6.17}$$

Equations 6.13, 6.15, 6.16, and 6.17 give a more precise upper bound on the interference by taking into account different delays corresponding to whether the task is delayed by bursts or individual accesses. These equations also accounts for memory accesses pipelining by

setting d_s and d_t accordingly which greatly reduces the over-approximation. There is an assumption that the number of blocking and non blocking accesses as well as the number of transactions is known or can be obtained from an external tool. This might not be always possible. In this case, all accesses are assumed blocking (until proven otherwise) and individual. In this case we have $S_i'^{x,b} = MD_i^b$ and $d_s = d_t$. Equation 6.17 becomes equivalent to 6.11.

6.6 TIMING COMPOSITIONALITY OF SHARED RESOURCE ACCESSES

The model we propose remains conservative assuming a timing composable and compositional architecture. As introduced in Chapter 2, compositionality means that the local worst-case scenario lead to a global worst-case scenario or when there is no timing anomalies. Composability means that different components (for example, cores, bus, shared memory) can be analyzed separately. The global analysis is then composed of local analyses.

The Kalray cores are simple enough to prove the absence of timing anomalies, thanks to the VLIW, in-order pipeline and the LRU replacement policies. Kalray MPPA-256 has been studied in the project *CERTAINTY* FP7[2] and preliminary work shows that it has fully timing compositional cores. This study however does not consider the behavior of shared resource accesses with interference. We discuss here potential issues and hypothetical scenarios that may exhibit anomalies.

6.6.1 *Left Side and Right Side Bus Masters*

The SMEM banks in the compute clusters are partitioned into 8 banks on the left side and 8 banks on the right side of cores (see Figure 6.2, page 86). Each core has one bus master per side. Cores issuing requests to both sides must read the accesses in the same issuing order.

Misaligned accesses are accesses to data which address is not a multiple of its memory block size. This results in an extra latency penalty. Also, data may be requested from different memory banks. One must investigate that an access from a core to the right side of the shared memory does not affect the accesses to the left side. Moreover, the interference on a bank on one side should not affect the access delays to the other side of the SMEM that could result in a domino effect. The following is an illustration of this scenario:

Example 10. *Consider a core performing misaligned accesses which result in two bursts.*

1. *Two bursts are issued to two banks on different sides b_0 and b_8 respectively.*
2. *The first burst encounter interference on b_0 and takes longer to come back.*
3. *The second burst has no interference on b_8.*
4. *Since ordering is enforced, the second burst is blocked until the first burst comes back.*

In the scenario above bank b_8 is blocked due to the interference on b_0. If b_8 remains blocked, potential concurrent accesses from other cores to this bank might be blocked. This

2 http:///www.certainty-project.eu/

in turn might block other banks, potentially resulting in unbounded delays (domino effect). We consider an execution model where one bank is mapped to one core. A task accesses (either remotely or locally) only one bank at a time. Thus, access misalignment on two banks does not occur in our case.

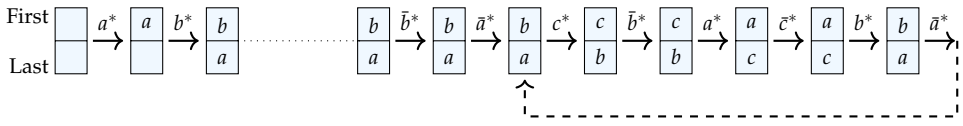
6.6.2 *Write Buffer*

6.6.2.1 *Anomalies Due to Used Policies*

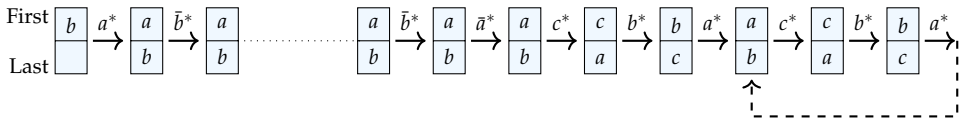
Timing compositionality in write buffers has been well studied in [DAR16] (Appendix C). The authors show the effects of different policies, such as retirement policies (determining when entries are retired from the write buffer) and write policies (determining for example whether data should be merged upon writes).

A potential source of timing anomalies is the replacement policy. In fact, some policies such as FIFO and pseudo-LRU have been proven to exhibit timing anomalies [Alt13; Rei08]. Anomalies may also occur at the combination of some implementations such as data replacement and data retirement policies. *Davis et al.* discuss timing anomalies of write buffers in [DAR16].

For example, there is a potential domino effect with the write merge and lazy retirement policies combined with FIFO replacement policy. The following example illustrates this situation.



(a) Empty write buffer



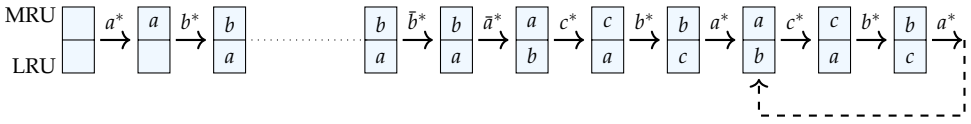
(b) Non-empty write buffer

Figure 6.9: Example of a domino effect with a FIFO write buffer: empty write buffer does not lead to the worst-case execution time [DAR16]

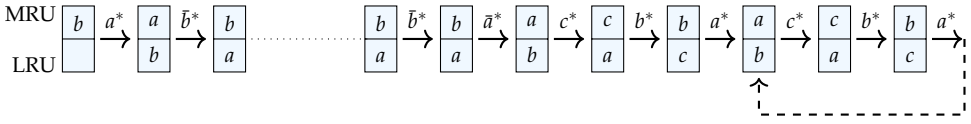
Example 11. *For the sake of simplicity, let us consider a write buffer of two entries with FIFO replacement policy. Figure 6.9 illustrates this effect as described in [DAR16]. Let a,b,c be data addresses. A write to a is noted by a^*, and a write merge is denoted by $\bar{a}^*$ (similarly for b and c). We consider the following write sequence: $a^*, b^*, b^*, a^*, c^*, b^*, a^*, c^*, b^*, a^*, \ldots$, with the sub-sequence c^*, b^*, a^* repeating. The first scenario in Figure 6.9a describes a situation with an initially empty buffer. When writing the sequence c^*, b^*, a^*, it results in a write merge at each second write. In Figure 6.9b, however, the write buffer is considered initially non-empty. This affects its content by switching a and b at the third step and making each write in the repeating sub-sequence stalling (no write merge is possible).*

The example above shows that considering an empty buffer initially as a worst case does not necessarily lead to the worst-case execution time for a FIFO write buffer.

Kalray MPPA-256's write buffers are write-only and use the LRU policy for data replacement combined with a *lazy retirement* policy. In the lazy retirement policy, data writes to the shared memory occur only when the content of the write buffer reaches a certain threshold. In this case, the oldest entry is evicted from the write buffer and written to the SMEM.



(a) Empty write buffer



(b) Non-empty write buffer

Figure 6.10: There is no domino effect with an LRU write buffer

The result of Example 11 with the LRU replacement policy is shown in Figure 6.10. The scenarios with empty (Figures 6.10a) and non-empty (Figure 6.10b) write buffers exhibit the same behavior. As a conclusion, it is safe to consider an empty buffer as a worst case to achieve the worst case execution time.

6.6.2.2 *Anomalies Due to Bus Interference*

The bus model we propose relies on known upper bounds on the number of shared resource accesses. These upper bounds can be obtained through different methods; using a dedicated static analysis tool, or through measurement-based approaches of tasks in isolation. There is an implicit assumption that these upper bounds in isolation are independent from concurrent accesses and therefore can be used safely when estimating shared resource interference. We show here that this is not always the case.

Example 12. *Figure 6.11 describes scenario that leads to different numbers of accesses depending on the bus interference. Assuming an initial state of the data cache and the write buffer as shown in figure. The first instruction results in writing h to the* 8^{th} *entry of the write buffer, therefore triggering the eviction of the LRU entry containing a. In isolation, the eviction issues an access to the shared memory. The total number of issued accesses to the shared memory is* 16 *in isolation. In the scenario with interference, evicting a may take longer due to the unavailability of the bus. By trying to read a while it is still in the buffer, the policy is to flush the entire buffer and reload a cache line from the shared memory. This produces* 8 *more accesses than the execution in isolation. The accesses to* $\{x1, x2, \ldots, x7\}$ *results in the same final state of the write buffer in both scenarios. It shows that the scenario with interference will not recover from the extra number of accesses.*

This example shows that simple single-core analyses, when used to obtain task information in isolation, are not compositional with regard to multi-core execution. This is because the number of SMEM accesses is dependent on concurrent accesses. Any analysis, though

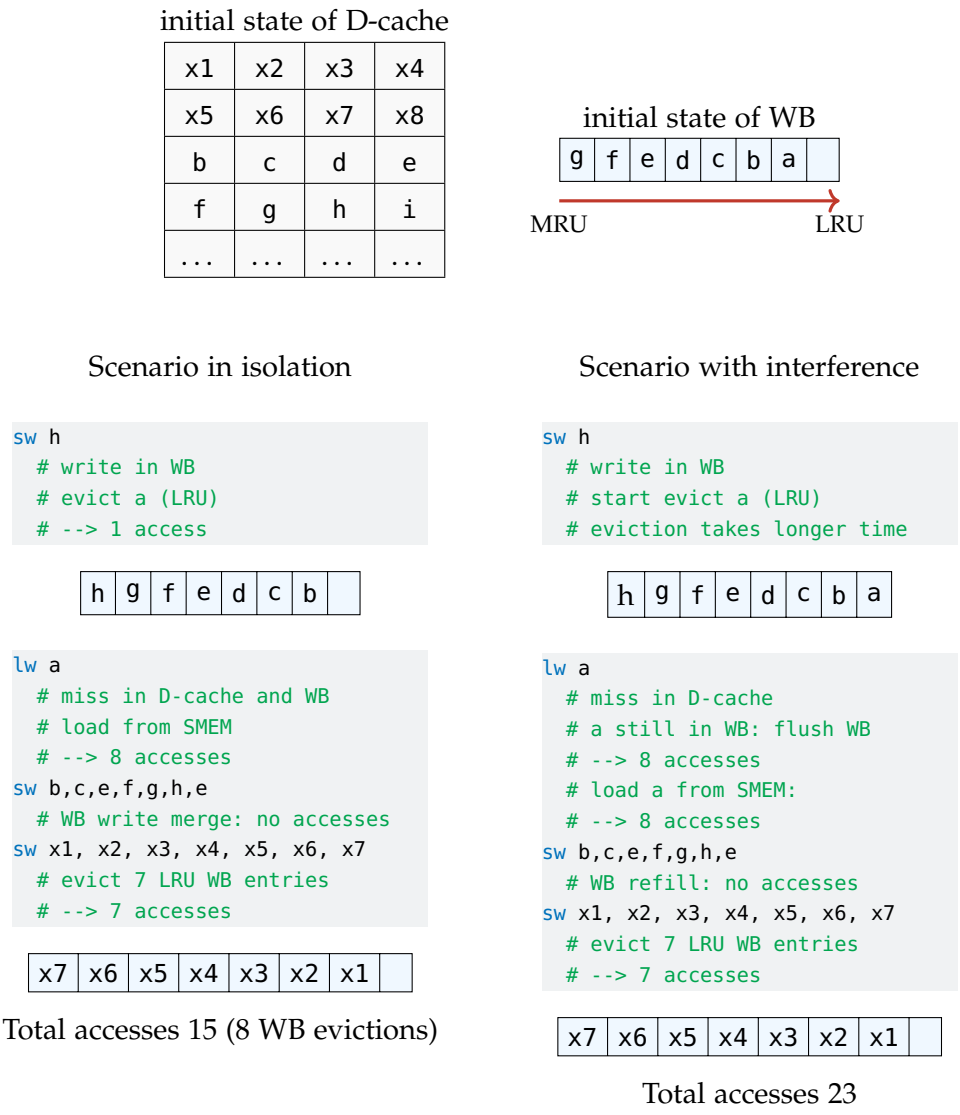


Figure 6.11: The same execution leads to different numbers of accesses in isolation and in interference

performed on tasks in isolation, must take into account potential interference from concurrent cores.

As a conclusion, the considered upper bound on shared resource accesses must take into account the aspects discussed above. Measured number of accesses, for example using hardware performance monitoring counters, obtained from measurement-based techniques might not be safe in this case, since it is harder to trigger the worst-case behavior under contention than in isolation. Static analysis tools, such as OTAWA, can conservatively compute this upper bound by modeling the write buffer pessimistically.

6.7 CONCLUSION

In this section we describe our approach that computes upper bounds on the shared bus interference. We target the Kalray MPPA-256 many-core processor and provide an accurate model of its shared memory accesses and the arbitration policy. We first present the multi-level arbiter and present a simple model that abstracts away low level details but gives conservative bounds. This model is then amended with more aspects of memory accesses. Here we take advantage of access bursts, access pipelining, and blocking/non-blocking accesses.

We focused on local interference between cores in a compute cluster when accessing the shared memory. We will later work on a model of the NoC traffic, i.e. $A_i^{Rx,b}$ and $A_i^{Tx,b}$. Any formula returning a number of accesses for a given time interval can be plugged into their computation: we can use the hardware configuration of the packet shaper (at the exit of each cluster), which limits the allowed bandwidth for each compute cluster on the NoC. Computations based on Network Calculus can also provide the worst-case number of accesses for a time window [Din+14a]. We can also use knowledge of the application's architecture (e.g., compute the amount of data that is written to and from the cluster during each clock cycle of the application).

The resource manager will also require consideration, as in general it can access the shared memory and is itself a shared resource to consider in response time computations. The RM is responsible of task deployments and interrupt handling. Therefore, context switches are performed by the RM. It is necessary to account for any delay affecting the execution of interrupts and context switches especially in hard real time systems. In our model we assume a conservative and constant penalty added to all tasks to account for such delays. We can also combine the results from existing work [Pha+13] to account for overheads when scheduling the analyzed application.

A full study of timing compositionality is required in order to provide a sound and correct timing analysis. Timing anomalies when discovered can be avoided by (i) hardware design (for example, adding stalls) which can be costly, in silicon and performance, and not always possible. (ii) Adding penalties in the analysis to account for any potential anomaly. This may result in very imprecise bounds. A work proposed in [HJR16] allows accounting for timing anomalies by adding corresponding bounded and sound timing penalties without greatly decreasing the analysis precision.

Although timing compositionality of the Kalray MPPA-256 may not be proved (or disproved), this platform is still a good processor designed with predictability in mind as it eliminates many sources of anomalies as shown in Section 6.6. This results in a theoretically smaller penalty to make up for any anomaly. A full study of these aspects must be conducted to achieve a correct and conservative analysis. This remains as a future work. In this work, we assume that timing anomalies do not occur or their analysis is taken into account by combining the results with those in [HJR16].

Part III

EVALUATION

Chapter 7

EXPERIMENTAL EVALUATION

In this chapter we evaluate our approach using different configurations. We show how the application model as well as the architecture configuration may affect the estimation of the WCRT. First, we compare the effect of multi-memory bank configuration on the schedulability of the application. Then, we investigate the effect of the number of blocking and non-blocking accesses/transactions, as well as the traffic of the NoC on the execution. Finally, we analyse a case study of a flight management system controller to validate our approach.

7.1 EXPERIMENTAL SETUP

Static analysis tools such as, OTAWA [BAL+10] and AIT [WIL+08][1], do not yet support (or only partially support) the Kalray MPPA-256 Bostan. For this reason, we establish the task profiles from measurement-based techniques such that: (i) the profiles are realistic although the resulted estimation is not guaranteed, (ii) the framework is compatible with inputs from formal methods or measurement-based methods. Each task is executed in isolation while profiling processor cycles and the number of cache misses. Several measurements are performed for each task and the results show a variance approaching zero. This reflects the efforts made in the design of the Kalray MPPA-256 targeting real-time applications. Note that the measurement-based approach may suffer from the situation where the number of accesses depends on the interference (see Example 11 on page 100).

1 To the best of our knowledge, AIT supports the first generation Kalray MPPA-256 Andey only.

In our experiments, we consider a bus delay from an interfering single access $d_s = 1$ and from an interfering burst of accesses $d_t = 8$ cycles obtained from internal specifications. We also consider that the context-switch delay is included in the task execution. In our analysis, we focus only on the steady state of the application, i.e., the first iterations are excluded and all data and code is already loaded. Therefore, we assume that the Resource Manager (RM), which loads the application onto the cores before operation starts, does not interfere with running tasks ($A_i^{RM,b} = 0, \forall i, b$). Finally, the Debug Support Unit (DSU) is disabled during operation ($A_i^{DSU,b} = 0, \forall i, b$).

7.1.1 *Bus Model*

We introduced our bus model in Chapter 6, more specifically in Equations 6.13 to 6.17 on page 98. It assumes known upper-bounds on shared resource access demand from cores as well as the access demand from the NoC. The shared resource access demand can be obtained by analyzing tasks individually and in isolation. Static analysis can be used to obtain precise upper-bounds on number of shared resource accesses and prove whether accesses are blocking or non-blocking. In the case where an access cannot be proven non-blocking, we simply assume it is blocking. Measurement-based techniques can also be used to generate the task timing information using the provided hardware counters. It is however challenging to know which accesses are non-blocking. We study the effect of the types of shared resource accesses in Section 7.3.2.

The NoC traffic affects the interference on shared resources. Our bus model can be used to provide an upper-bound on the interference considering the upper-bound on the number of accesses from the DMA during a time frame. This can be modeled as "tasks" that do not perform any computation and only access the shared memory. Our model is flexible and can be adapted to considering arrival curves instead. We present the effect of the NoC on the timing analysis in Section 7.3.3.

Applications that run in isolation on a compute cluster to do not suffer from NoC interference. Therefore, accesses from the NoC do not occur during the execution of the application of interest. As a consequence, by setting $A^{Rx,b} = A^{Tx,b} = 0$ in Equations 6.15 and 6.16 on page 98, the interference is simplified to one level of round-robin arbitration.

7.1.2 *Execution Model*

We first consider a single-phase execution model where we make no assumptions about the distribution of read and write accesses between the start and the end of a task. In our code generation scheme for the SDF model, tasks execute computations, then write the result to a shared memory location where the next task can read it. Similar to [Mel+15], this execution model allows each task to be split into a first *execution* phase, limited to reading the input and doing computations, and a *write* phase where the output is sent to the next task. In the execution phase, the accesses are to the local memory bank of the task, whereas in the write phase requests may access a remote memory bank. We exploit this execution model in our analysis. We consider the two phases of a task as separate subtasks with a direct

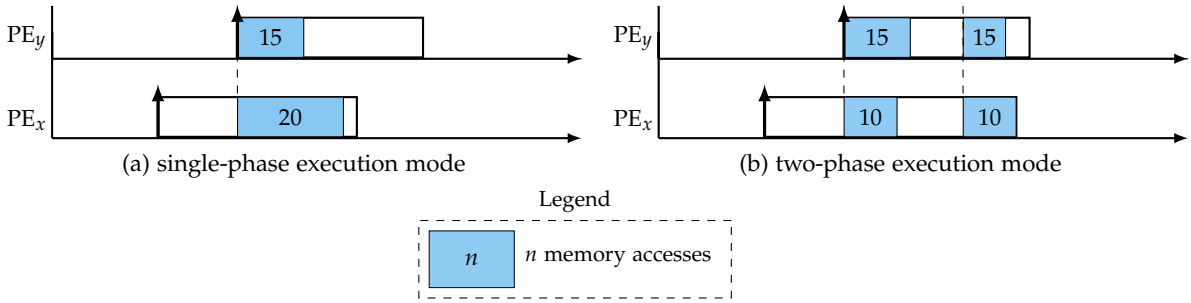


Figure 7.1: Pessimism in single-phase and two-phase execution models

dependency relation. Using our analysis technique we compare the single-phase model with the two-phase model.

Note that the effect of the execution model depends on the application itself. There exist cases where it is better to consider a single-phase execution model. Figure 7.1 illustrates this situation in a simplified way, where all accesses are blocking and $d_t = 8, d_s = 1$. In Figure 7.1a (single-phase model), the number of accesses that may delay the task running on P_x is $\min(20 \times 8, 15 \times 1) = 15$ cycles. In Figure 7.1b, the two phases are represented with two sub-tasks, each of them seeing $\min(10 \times 8, 15 \times 1) = 15$ cycles interfering accesses (as a worst-case interference). This results in a total of 30 cycles counting both phases. Due to this effect, one would perform the analysis on both execution models and would consider the one with the smallest estimation.

7.1.3 *Experiments*

We explore and compare a number of setups for the experimental evaluation to determine the effectiveness of various techniques that form part of the schedulability analysis. In the first experiment **E1**, we use our approach taking into account a two-phase execution model. Experiment **E2** also applies our approach, but using a single-phase execution model. In experiment **E3**, we use a simplified approach that discards the release dates of tasks, meaning that all tasks potentially overlap, and considers the tasks using the two-phase execution model. The same approach as **E3** is used in **E4**, but using the single-phase execution model. Finally, we consider in experiment **E5** that co-runners continuously interfere with the task of interest by issuing bursts of accesses to the shared memory. This is a pessimistic analysis that assumes the worst-case interference on each memory access. Note that this may result in unbounded interference due to the fixed-priority level of the MPPA bus. In this case, we consider the upper bound on the number of accesses by all higher priority components during the analysed execution instance. Then, we assume that each task access is delayed by all the higher priority accesses. In the following, we compare the different analyses with different arbitration policies for each benchmark.

In summary:

- **E1** analysis of two-phase task model
- **E2** analysis of single-phase task model

Task	WCET (cycles)	MD (accesses)	Dependencies	Accesses to the bank of					
				τ_1	τ_2	τ_3	τ_4	τ_5	τ_6
τ_1	5	2	$\emptyset$		30		10		
τ_2	8	10	$\{\tau_1\}$			20			
τ_3	20	18	$\{\tau_2, \tau_4, \tau_6\}$						
τ_4	5	2	$\{\tau_1\}$			50			
τ_5	8	10	$\emptyset$						20
τ_6	20	8	$\{\tau_5\}$			50			

Table 7.1: Task profiles of the SDF example in Figure 5.2a

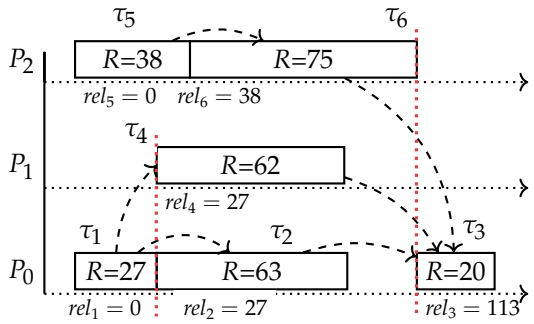


Figure 7.2: Static scheduling of the example in Figure 5.2a considering 3 memory banks

- **E3** analysis of two-phase task model without accounting for the release dates
- **E4** analysis of single-phase task model without accounting for the release dates
- **E5** pessimistic analysis without consideration of co-runner tasks

7.2 DIDACTIC EXAMPLE

We first present the result of analyzing a simple didactic example similar to Figure 5.2a (on page 71). For this example, we use the profiles in Table 7.1. Figure 7.2 gives a static schedule computed by our approach which accounts for the bus interference and the dependencies between tasks. Figure 7.3 compares the end-to-end estimated response time obtained with different analyses.

We note that taking into account the memory banks always yields a better estimation of the overall response time. Further, taking into account the two-phase execution model ($E1$), the estimation is 10.27 times smaller than the pessimistic approach ($E5$) while the analysis with the single-phase execution model ($E2$) is 8.49 times smaller. The approache $E3$ (resp. $E4$) that discards the release dates is 5.62 (resp. 6.24) times smaller than $E5$ when taking into account the memory banks.

The micro-benchmark discussed above shows the functioning of our framework on a simple program. It does not fully expose how the estimated WCRT behaves in different

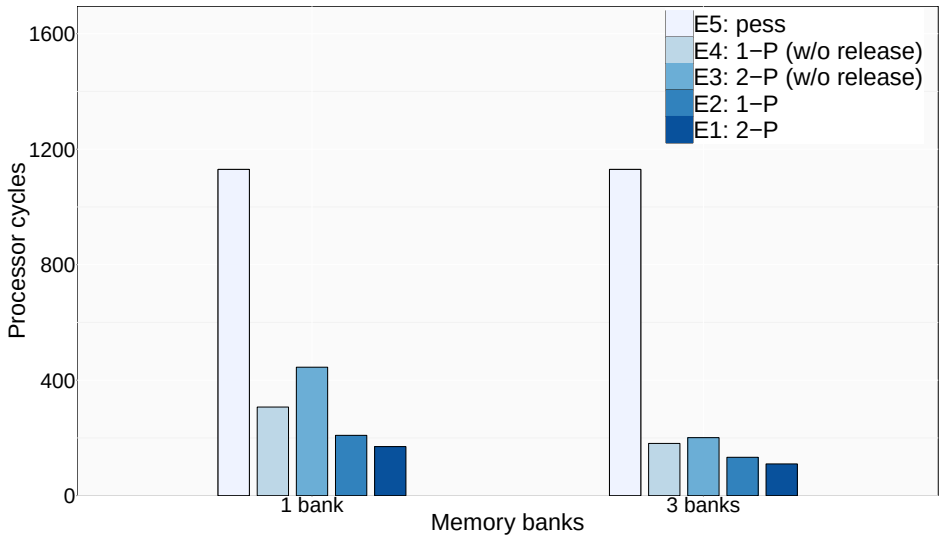


Figure 7.3: Comparison of the end-to-end response time obtained with different analyses of the SDF example in Figure 5.2a

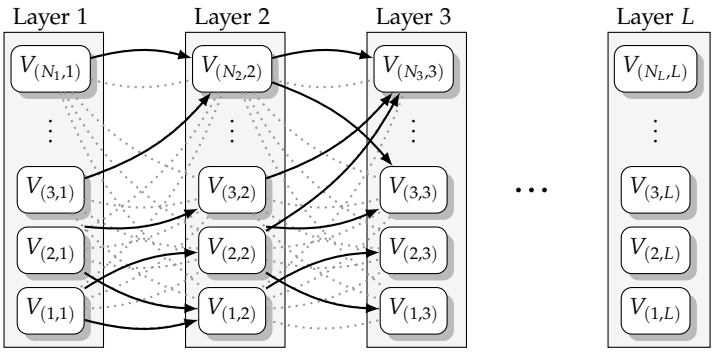
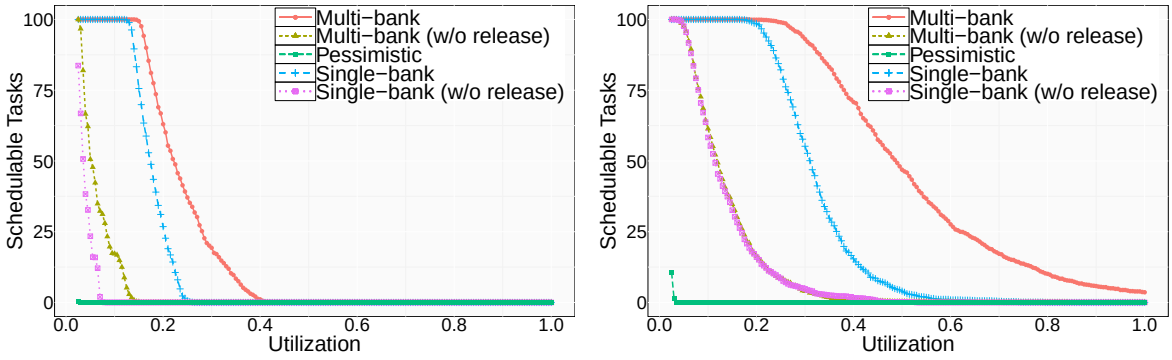


Figure 7.4: The *layer-by-layer* method in DAG generation. An example with L layers and N_k vertices per layer $(1 \leq k \leq L)$. Edges are generated according to a given probability.

situations. The following section goes into further analyses of our framework with varying workload parameters.

7.3 RANDOMLY GENERATED DAGS

To better understand several aspects of our analysis, we simulate randomly generated graphs with varying parameters including utilization, blocking and non-blocking accesses/transactions, and traffic of the NoC. The graphs are generated using the *layer-by-layer* method proposed by *Tobita and Kasahara* [TK02] as shown in Figure 7.4. Each layer represents a set of independent vertices. Edges are generated between layers with a certain probability. To avoid cycles, edges are directed from the lower index vertex to the higher index vertex (both vertices are in different layers). We denote the probability of creating an edge by ρ.



(a) fat graphs $25 \leq N \leq 50$ (b) long graphs $15 \leq N \leq 35$

Figure 7.5: Number of schedulable DAGs vs. utilization with: $M = 8$ cores, $b = 1$, $\rho = 0.5$

Each vertex represents a task with the following parameters: wcet_i, MD_i, $WCBT_i$, initial release dates $rel_i = 0$, and an assigned core. wcet_i is the worst-case execution time in isolation. MD_i is an upper-bound on the number of accesses due to cache misses and other accesses related to the execution of the task (uncached accesses, atomic operations,... etc). In our experiments, wcet_i and MD_i are generated randomly and uniformly in $[550, 650]$ and $[250, 550]$ respectively. $WCBT_i$ is determined by $WCBT_i = b \times MD_i$, where b denotes the ratio of blocking transactions. Edges in the DAG represent dependencies due to communications. Each edge has a parameter that gives the number of accesses required in the communication, also generated randomly in $[0, 100]$.

Cores are assigned to tasks sequentially. The first generated task is mapped to the first core, the second task to the second the core, and so on. We go back to the first core when all cores are assigned.

The schedulability of the generated DAGs depends also on their shape. In the layer-by-layer generation method, DAGs can be *long* (more layers and fewer vertices per layers) or *fat* (fewer layers and more vertices per layer). Fat DAGs increase the parallelism and therefore the interference between co-runner tasks. In the following, we generate random long and fat DAGs. Our experiments show how the interference affects the analysis differently depending on the shape of DAGs.

7.3.1 *Effect of CPU Utilization*

We study the effect of shared resource interference on the schedulability of the application considering different utilization. Utilization here is defined as the ratio of a base execution time with regard to the period. We then run the analysis that takes into account the interference and determines how it affects the application.

Our target application model is periodic DAGs with implicit deadlines, where all tasks in the DAG have the same period and deadline. We define the following characteristics for each DAG: (i) the end-to-end response time with no interference, R_{min}, and (ii) the end-to-end response time on a single-core, R_{single}. R_{min} is obtained from our framework (given a mapping and an execution order) considering a perfect bus, i.e, $I^{\text{BUS}}(i, x, \mathcal{R}, \Theta) = 0, \forall i, x$.

R_{single} is obtained by summing all wcet_i, which is equivalent to a single-core execution also with no interference (we do not consider interference from the NoC).

To compute the deadlines, we use a formula that is a function of R_{min}, R_{single}, and the number of cores M assigned to the DAG:

$$D = (R_{\text{min}} + 2 \times \frac{R_{single}}{M}) \times \frac{1}{u} \tag{7.1}$$

where u is a utilization factor in $[0, 1]$. This formula is similar to the one used by *Saifullah et al.* [Sai+14] with the difference, in their case, that several DAGs are scheduled on the same processor and their utilization follow a Gamma distribution. In our case, we want to show the effect of intra-DAG interference on shared resources and how it affects its schedulability considering the formula in Equation 7.1. Note that for the same task set, long DAGs result in larger R_{min} whereas fat DAGs result in a smaller R_{min}. Equation 7.1 produces a larger deadline in the former case.

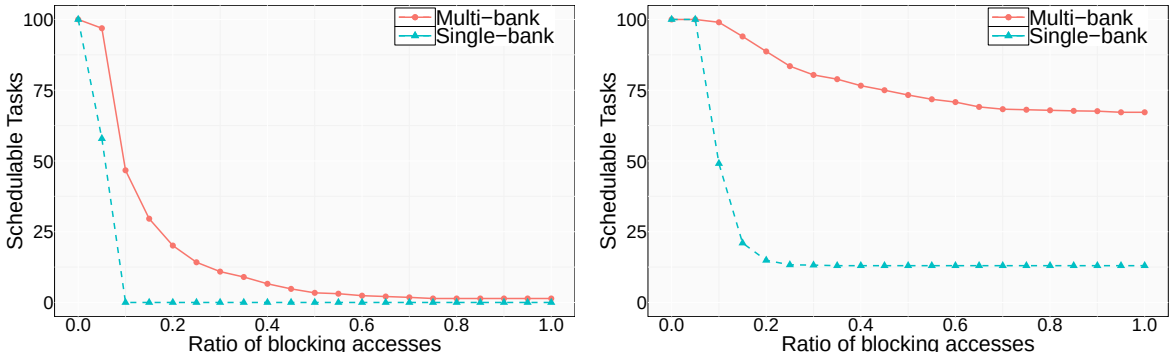
Figure 7.5 shows the number of schedulable tasks against the utilization parameter u. For each value of u, 1000 tasks are generated and analyzed. u is varies between 0.025 to 1 with a step of 0.005. Here, we compare the effect of considering the shared memory bus and/or the release dates. We consider tasks where all accesses are blocking (b=1), edges (and therefore communications) are generated with the probability $\rho = 0.5$. The generated DAGs contain 25 to 50 tasks mapped to 8 cores for fat DAGs and 15 to 35 tasks for long DAGs.

The results show that, as expected, the memory banks reduces the interference considerably: at $u = 0.265$ (resp. $u = 0.85$) the single-bank analysis has 0% of schedulable tasks where the multi-bank analysis schedules 31.8% (resp. 6.4%) for the case of fat DAGs (resp. long DAGs). Moreover, when the release dates of tasks are not taken into account, all tasks are unschedulable at $u = 0.155$ (resp. $u = 0.08$) for the multi-bank analysis (resp. the single-bank analysis) in the case of fat DAGs. The analysis that does not take into account memory banks and release dates schedules only 83.7% of tasks at $u = 0.025$ for fat DAGs. The pessimistic analysis schedules 0.5% of tasks at $u = 0.025$ for fat DAGs and 0.01% of tasks at $u = 0.035$ for long DAGs. Then, it falls to 0% afterward.

7.3.2 *Effect of Blocking Transactions*

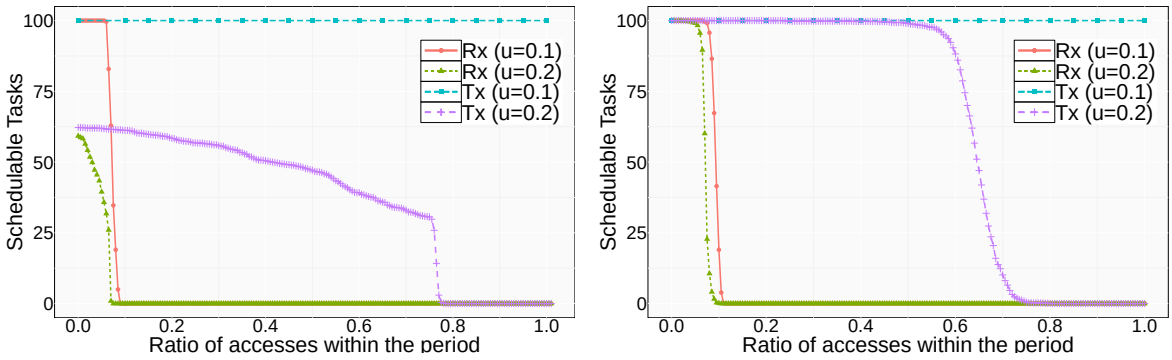
To evaluate the accuracy of our bus mode, we propose the experiment with a varying number of blocking accesses. Here we choose $WCBT_i$ values such as $WCBT_i = MD_i \times b$, where b is a ratio in $[0, 1]$. $b = 0$ means that all accesses are asynchronously executed without stalling the task. $b = 1$ means that the task blocks at each access and suffers from potential interference.

Figure 7.6 shows how the number of blocking accesses affects the end-to-end response time DAG and therefore its schedulability. Here again, we compare the single-bank analysis and the multi-bank analysis. We fix the utilization at $u = 0.4$. For fat graphs (Figure 7.6a), when $b = 1$, 14% and 0% of tasks are schedulable in the multi-bank and the single-bank analyses respectively. This is also shown in Figure 7.5a. When $b = 0$, 100% of the tasks become schedulable in both analyses. The number of blocking accesses has more effect in



(a) fat graphs $25 \leq N \leq 50$ (b) long graphs $15 \leq N \leq 35$

Figure 7.6: Number of schedulable DAGs vs. blocking access ratio with 8 cores, $u = 0.4$, $\rho = 0.5$



(a) fat graphs $25 \leq N \leq 50$ (b) long graphs $15 \leq N \leq 35$

Figure 7.7: Number of schedulable DAGs vs. NoC traffic with: 8 cores, $\rho = 0.5$, $b = 1$

the single-bank analysis. 100% of the tasks become non schedulable when $u = 1$ against 86% in the case of the multi-bank analysis.

In the case of long graphs (Figure 7.6b), when $b = 1$, the multi-bank analysis schedules 67.2% of tasks against 13% for the single-bank analysis. It shows how the blocking accesses' interference affect the schedulability of fat graphs more than long graphs.

7.3.3 *Effect of the Network-on-Chip*

The Network-on-Chip is a source of interference that we include in our model of the bus arbiter. The NoC is implemented with Real-Time Calculus and therefore its traffic follows arrival curves. We can use these curves to extract upper-bounds on the number of accesses during a time window. In this thesis, we delegate the analysis and extraction of arrival curves to an external tool. Nevertheless, we show how a varying workload of the NoC affects the schedulability of the application.

In this experiment, the number of accesses from Rx (resp. Tx) varies with a factor of the DAGs period. We assume that the accesses start at release date 0. Figure 7.7 represents the number of schedulable tasks in presence of different NoC workloads, according to our

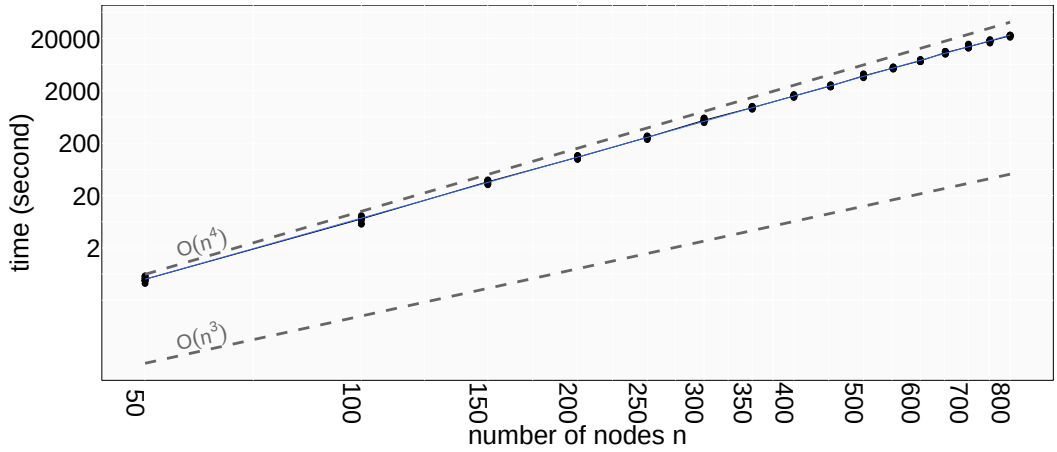


Figure 7.8: Run-time analysis in log-log scale with: 8 cores, $\rho = 0.5$. Graph lines for $\mathcal{O}(n^3)$ and $\mathcal{O}(n^4)$ are shown as an indication.

analysis that takes into account the memory banks. Note that Tx is arbitrated with a round-robin at L3 of the multi-level arbiter while Rx has a fixed high priority at level L4. Although Tx can execute more times than the cores, its effect on the schedulability is less harmful than Rx. On the other hand, the Rx directly affects the schedulability if it executes more than 0.09 (resp. 0.065) of the period when $u = 0.1$ (resp. $u = 0.2$) in the case of fat graphs (Figure 7.7a). In the case of long graphs (Figure 7.7b), the tasks become unschedulable if Rx executes more than 0.115 (resp. 0.105) of the period when $u = 0.1$ (resp. $u = 0.2$).

The above experiments are artificial scenarios that do not impose any restriction on the amount of accesses from the NoC. Note that on the real platform, the number of packets sent on the NoC (through Tx) follows a packet shaper at the routers. The packet shaper ensures that the NoC is not congested so that each packet sent is guaranteed to reach destination. For this reason, the Rx has the highest priority in the multi-level arbiter. The number of accesses from the Rx is also subject to the packet shaper at the Tx of the distant cluster emitting the accesses.

7.3.4 *Performance Analysis*

Here we study the time complexity of Algorithm 8 (ComputeRT). Let n and e be the number of tasks and edges (dependencies) respectively in the analyzed application. MultiCoreRTA (Algorithm 6) takes up to n iteration to converge. Each iteration the response time of all tasks, which takes n iterations. The response time formula depends on the interference function $I^{\text{BUS}}(i, x, \mathcal{R}, \Theta)$ which takes up to $n-1$ iterations to compute the interference from concurrent tasks. Thus, MultiCoreRTA has a time complexity of $\mathcal{O}(n^3)$. UpdateReleases (Algorithm 7) iterate over all tasks. For each task, the algorithm checks iterate over its dependencies (edges) to check and potentially update the release date. Thus, the time complexity of this algorithm is $\mathcal{O}(ne)$.

Finally, ComputeRT takes $n-1$ to converge, therefore, the total time complexity is:

$$\mathcal{O}(n^4 + n^2 e)$$

Since, $e < n^2$, case where there are edges between all tasks in the graph, the time complexity is simply:

$$\mathcal{O}(n^4)$$

To get an idea on our analysis' scalability, we run an experiment with an increasing number of tasks while measuring the time it takes our algorithm to terminate. For each number of tasks, 10 benchmarks are generated and measured. This experiment run on an Intel® Core™ i5-4590 at 3.30 GHz with 16 GB of RAM. The results are shown in a log-log scale in Figure 7.8. At $n = 800$, the analysis' run-time is around 6 hours. At $n = 500$, the analysis takes around 1 hour. The run-time also depends on the number of edges. With the probability to generate an edge $\rho = 0.5$, the number of edges of the generated benchmarks with $n = 500$ is around 62000 edges. The slop of the line in Figure 7.8 is 3.87, therefore, the asymptotic running time obtained by the experiment is $\mathcal{O}(n^{3.87})$.

7.4 ROSACE (FLIGHT MANAGEMENT SYSTEM)

Pagetti et al. [Pag+14] provide a case study of a Flight Management System (FMS) called *ROSACE*[2]. The case study consists of a multi-rate controller and an environment simulator. In this kind of application, the input (sampled from physical sensors) is transmitted to a controller which, after computation, sends commands to the actuators. Figure 7.9 illustrates the set of tasks in the SDF application, their inputs, outputs, dependencies, and their rates. We assume that it takes 8 accesses to write a token. We established task profiles by executing each task in isolation and measuring a trace of its execution. Since we cannot establish which access or transaction is blocking, we consider all the measured SMEM accesses to be blocking. The profiles are given in Table 7.2.

The inputs from sensors and the commands to the actuators are sent through the NoC via the Rx and Tx components. Since there are multiple rates, we unfold the SDF program over a hyper-period in order to make the model compatible with our approach. In this case, tasks with a frequency of 100 Hz execute twice within the hyper-period, while tasks with a frequency of 50 Hz execute only once. Further, the Rx component writes the inputs (h, az, vz, q, va) to the shared memory four times (200 Hz) within the hyper-period, while the Tx component reads and transmits the outputs $(\delta_{ec}, \delta_{the})$ once (50 Hz). Our experiments consider a time window with the length of the hyper-period that starts with the Tx accesses from the previous execution of the program.

There are several possible mappings for the multi-rate application. We choose the mapping described in Figure 7.10 and evaluate its schedulability with the previously defined analyses. We also consider a single-level round-robin bus (RR) as well as the multi-level arbiter (MPPA). This allows us to compare the performance of the MPPA against the conventional RR arbitration policy using our approach. Figure 7.11 gives the smallest period, in processor cycles, for which the mapping in Figure 7.10 is schedulable. This is equivalent to finding the slowest processor clock frequency that satisfies the scheduling requirements.

The results in Figure 7.11 show that accounting for the memory banks improves the estimation with a factor of 1.28 to 1.92 in $E1, E2, E3, E4$ (5 banks vs. 1 bank). Our refined

2 Open source implementation available on the svn repository https://svn.onera.fr/schedmcore/branches/schedmcore-RTAS2014/Case_Study_RTAS

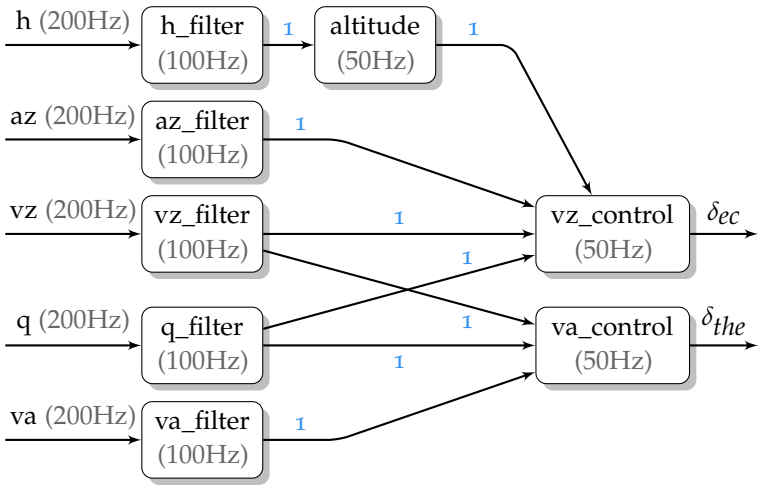


Figure 7.9: Flight Management System controller

Function	WCET (cycles)	MD(accesses)
altitude	275	22
az_filter	274	22
h_filter	326	24
q_filter	338	24
va_control	303	24
va_filter	301	23
vz_control	320	25
vz_filter	334	25

Table 7.2: Task profiles of the FMS controller

approach that takes into account the number of memory banks and the release dates can verify schedulability with a hyper-period of 1376 cycles ($E1$) and 1388 cycles ($E2$) assuming the MPPA bus. This represents an improvement by a factor of 7.34 ($E1$) and 7.27 ($E2$) compared to the pessimistic approach in $E5$ with 10104 cycles. The gain achieved by considering release dates is a factor of 1.40 in $E1$ (respectively 1.14 in $E2$) when compared against $E3$ (resp. $E4$) which ignores release dates. Our analysis with the RR bus gives an estimation of 1352 cycles in $E1$ (resp. 1376 cycles in $E2$) which corresponds to a gain of a factor 5.28 (resp. 5.19) when compared to the pessimistic approach in $E5$ that has 7152 cycles. Note that the two-phase model is more pessimistic than the single-phase model when comparing $E3$ and $E4$. This is due to accumulated pessimistic considerations on the write phase and the execution phase which may lead in some cases to a higher estimation than when the execution is considered as a single phase, as described in Section 7.1.2. The analysis of the RR arbiter provides slightly better performance than that for the multi-level arbiter. Any pessimistic assumption in the analysis have a higher effect on the multi-level arbiter than

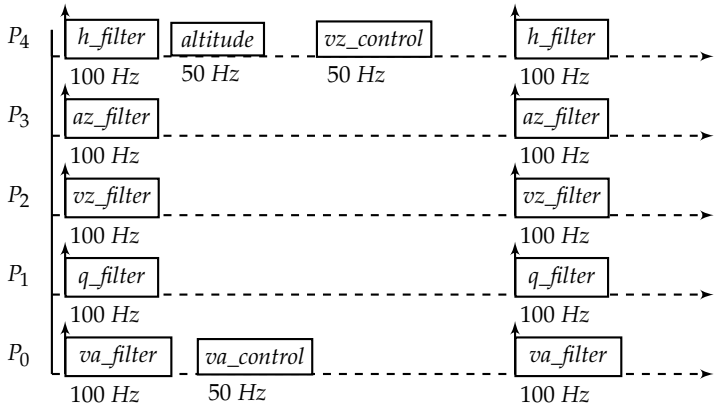


Figure 7.10: Task-to-core mapping and unfolding of tasks in the FMS controller

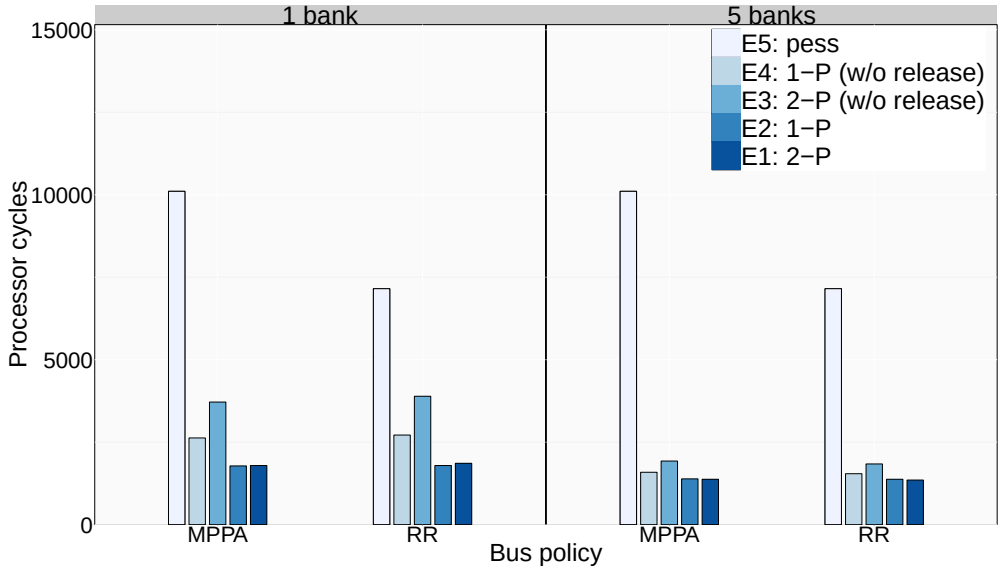


Figure 7.11: The smallest schedulable period obtained with different analyses

the RR arbiter. This is due to the fixed-priority level that pessimistically counts all highest priority accesses at each bus access.

Finally, we comment on the run-time of our approach. The analysis of the FMS controller takes 0.15 seconds (Intel 2.4 GHz CPU). The analysed hyper-period has 18 tasks. The analysis in $E2$ (single-phase execution model) takes 3 iterations in Algorithm 8 and at most 20 iterations at each execution of Algorithm 6. In $E1$, the analysed hyper-period has 31 subtasks/tasks and takes 4 iterations in Algorithm 8 and at most 26 iterations at each execution of Algorithm 6.

7.5 CONCLUSION

We evaluate our analysis that computes a valid static schedule of a synchronous data-flow application on the Kalray MPPA-256 multi-core architecture with shared memory and a

multi-level arbiter. We start the analysis with a given mapping, set of dependencies between tasks and precedence constraints: the choice of the mapping and the order of tasks on a given core can either be defined manually or delegated to a separate allocation algorithm.

The analysis we derive is based on the Multi-core Response Time Analysis (MRTA) framework [Alt+15]. We extend this framework by deriving a mathematical model of the multi-level bus arbitration policy used by the Kalray MPPA-256. Further, we refine the analysis to account for the release dates and response times of co-runners, and the use of memory banks. Improvements to the precision of the analysis may be achieved by splitting each task into two sequential phases, with the majority of the memory accesses in the first phase, and few writes in the second phase. Our experimental evaluation addresses the ROSACE avionics case study. Using measurements from the Kalray MPPA-256 as a basis, we show that the new analysis introduced in this paper leads to response times that are a factor of 7.35 smaller compared to the default approach of assuming that each access is subject to the worst-case interference.

We also study the effect of different parameters on the estimated WCRT. Here we focus on (i) the effect of memory banks and release dates, (ii) the effect of blocking transactions, and (iii) the effect of accesses from NoC. We rely on randomly generated DAGs using the *layer-by-layer* method. With some tweaks, the method can generate *Fast Fourier Transform* graphs, *Laplace* graphs, or *Stencil* graphs. In future work, other DAG generation methods [DRW98; ER59] can also be investigated with regard to the above parameters.

Through the randomly generated benchmarks, our analysis shows that it can take advantage of precise information when available on the application model. For example, our analysis with non-blocking accesses yields a better estimation compared with pessimistic analyses with the same ratio of non-blocking accesses.

Chapter 8

FROM TIMING ANALYSIS TO REAL-TIME IMPLEMENTATION

In this chapter we discuss the result presented in this thesis and how it can be used in state-of-the-art work. Scheduling techniques must rely on tight estimations of the WCRT which in turn depends on co-runner tasks. However, in order to obtain a tight upper-bound on the response time, a mapping and scheduling should be known in advance. Indeed, the response time is highly influenced by the co-runner tasks. Concurrent accesses to the same shared resource may introduce interference that should be accounted for in the response time analysis. The search for an optimal scheduling with a tight WCRT analysis that includes the shared resource interference is a challenging open problem.

8.1 DESIGN CHOICES AND IMPLEMENTATION

SDF languages such as Lustre [Hal+91] offer an efficient programming paradigm that, using a certified compiler, can produce deterministic sequential code. An execution instance of an SDF application is represented with a task dependency graph, where the amount of exchanged data among the tasks is deterministic and known in advance. We consider the application to be running on a multi/many core architecture with a partitioned shared memory. Among the challenges while parallelizing such applications, we need to (i) specify a task mapping that optimizes a certain cost function; (ii) specify a scheduling per processing element that respects dependencies; (iii) find a tight estimate of the response time that takes into account the interference from co-runners. There exists several solutions for the mentioned points when taken individually. However, the connection and the interaction among them remains an open problem.

8.1.1 *Code Generation and Impact on WCRT*

A code generator transforms a high-level language into a low-level language (such as the C language) which in turn is compiled to run on a target architecture. Here synchronization and communication routines are added to ensure functional correctness of the program. Execution models can be applied: (i) A single-phase execution model where the output is "sent" to the next node as soon as it is ready. (ii) A two-phase execution model with an *execution phase* and a *replication phase*. The *execution phase* uses only the local memory bank to compute and store the output data. Then, a *replication phase* is added to copy the output data into their memory destinations. The number of shared resource accesses may be significantly increased, but with a good scheduling of the execution and replication phases the interference may be reduced. The choice of the execution model affects the results in the next steps.

The WCRT analysis must be able to derive tight upper-bounds on the interference due to concurrent accesses to shared resources, and therefore to estimate the response times and release dates for all tasks. This requires a task mapping and scheduling as well as the constraints on the task dependencies. The execution model given by the code generation step can highly influence the analysis. As seen in Chapter 7, in the case of a two-phase execution model, the upper bounds on the shared resource interference in the execution and replication phases might be too pessimistic compared with the single execution model. One may run the analysis on both execution models and retain the one with the best overall.

8.1.2 *The WCRT–Mapping–Scheduling Relation*

The work in this thesis sheds the light on the current state of timing analysis on multi-core systems. The traditional analysis (as inherited from single-core systems) is done in two steps: (i) get the timing information of each task, (ii) perform the scheduling and mapping according to the provided information. Due to the potential large interference on shared resources, it is hard to reach a satisfying result with such approach.

The response times must be re-estimated to precisely account for the interference on shared resources and potentially reduce idle times. Since the timing information changes, a new mapping and schedule might need to be recomputed. A possible solution relies on constraint programming languages. A solver finds a satisfiable task mapping that optimizes a cost function (for example, the end-to-end response time). The WCRT Analysis can then compute a tighter estimation of the response times by taking into account potential interference. New timing constraints are added to the model and the search stops when no better solution can be found.

Since the mapping problem is NP-hard, it is hard to prove that the iterations between WCRT analysis and scheduling/mapping analysis converges toward an absolute optimal solution (assuming its existence). In the absence of such guarantee, one can iterate for a certain number of times and choose the best solution. This interaction remains as an open problem.

Holistic approaches that perform both steps at the same time seem a good starting point toward efficient execution on multi-core systems. So far, our timing analysis is used in

state-of-the-art work [Gra+18; MHP17] only to tightly estimate the interference on shared resources in precomputed schedule and task mapping.

8.2 INTEGRATION WITHIN THE CAPACITES PROJECT

This work takes place within the *CAPACITES* project [Cap]. This project aims at investigating and efficiently exploiting integrated many-core architectures in safety-critical hard real-time systems. The contributions target Kalray MPPA-256 many-core processor with a strong focus on industrial applications. Among other goals, the project aims at: providing timing analysis tools, investigating the predictability of the hardware, adapting safety-critical software, etc.

Our contribution is integrated in the workflow of the tool-chain shown in Figure 8.1. It aims at producing an executable binary and static schedule and mapping from an SDF program written in a high-level language such as SCADE or Lustre. The process goes through source-to-source compilation from a high-level language to a low-level language (typically C). The steps taken are:

1. A code generator performs source-to-source compilation from a Lustre code to a C code. Execution models can be implemented here. Note that, we analyze here the tasks and not the wrapping code that call them. In this phase, any initial simple mapping and scheduling can be used to create a binary.

2. By analyzing the binary, this phase establishes tasks' WCETs in isolation, either by measurement-based techniques or using an adapted version of OTAWA. Since the binary does not have the final mapping and scheduling of tasks, the WCETs obtained here are an approximation to the final WCETs of tasks with their final mapping and schedule.

3. The WCETs are used to generate a task mapping and a schedule. Any mapping and scheduling technique (for instance, [NHP17]) can be used as long as it takes into account the task dependencies.

4. The schedule and task mapping are added to the C code. This includes synchronization routines to ensure functional properties and dependencies between tasks. This step generates a new binary.

5. New profiles (WCETs and shared resource access requests) are generated from the new binary. The introduced synchronization routines must be included in the interference analysis. The task mapping and the schedule change the code and data memory addresses, thus the number of accesses per memory banks.

6. The contribution of this thesis comes in this phase. Here, a new refined schedule is generated to accurately account for the interference on shared resources.

7. In this last step, the new release dates are injected in the synchronization routines. The final binary is produced here. Note that adding/updating release dates must not

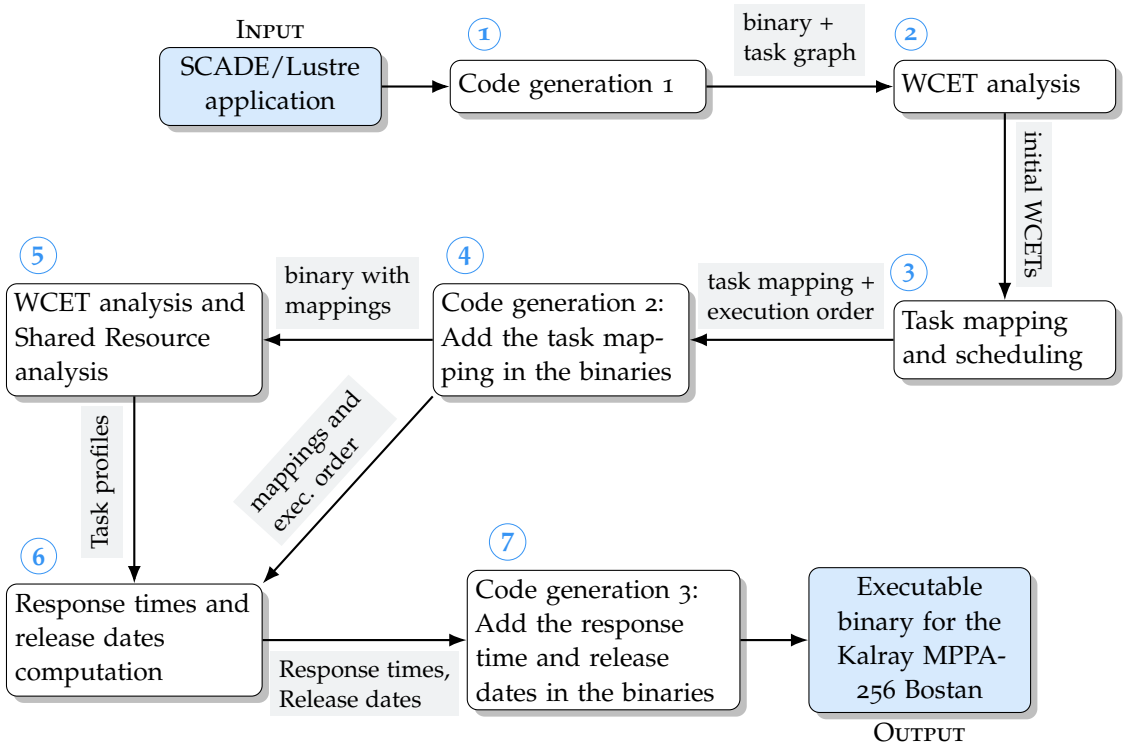


Figure 8.1: The proposed tool-chain within the CAPACITES project

change the memory layout [1] of the binary. Changing the memory layout may result in a different number of accesses which may invalidate the analysis.

The above tool-chain is used by *Graillat et al.* [Gra+18] that focuses mainly on code generation. The experimental evaluation is performed on a Lustre version of the ROSACE case study. Primary results show that tasks execute 26.1% to 49.6% of the budget time allocated to them – this budget is obtained using our approach. Our approach is also used by Martinez et al. [MHP17] who focus on the scheduling and mapping part of the tool-chain.

8.3 CONCLUSION

Multi-core and many-core systems are a reality in embedded systems. This is a game changer in terms of design, analysis, and validation tools. We highlight in this thesis the importance of the interference on shared resources. We do not address the scheduling and mapping problems. Our proposed approach can be integrated with other tools to refine and improve the execution on multi-core and many-core systems.

The preliminary results of comparison with the real execution show that our approach gives safe and conservative estimation on the execution time. The ROSACE case study (used in [Gra+18]) is a small application with little memory accesses. It would be more interesting to compare the estimated execution times with real execution times of a memory-intensive industrial case study.

1 Location of data and code in the memory

A static mapping/scheduling can be defined with: a task-to-core mapping, an execution order per core, and release dates. In our approach, we consider a list scheduling where only the release dates are updated to account for the interference. A future work is to investigate how changing the execution order and task mappings affects the end-to-end response time. This becomes more interesting in the two-phase execution model, when co-scheduling communications and computations such that the interference is reduced [Mel+15; Bec+16]. The tool-chain in Figure 8.1 can be extended by adding an iteration from (6) to (3). The task mapping and/or scheduling are updated with new timing information on the interference.

Chapter 9

CONCLUSIONS AND PROSPECTS

9.1 SUMMARY

Multi-core and many-core architectures are emerging in embedded real time systems. This thesis addresses the issue of taking into account the interference between cores, with shared buses, in the Worst-Case Execution Time (WCET) analysis.

Our analysis computes hard bounds on the execution time: any approximation done in the analysis should be conservative; the worst-case may be overestimated, but not underestimated. Previous works showed that formal methods like model-checking, abstract interpretation, SMT-solving (logical formula resolution) and real-time calculus could be used for worst-case timing analysis. Still, existing timing analyses of shared-memory systems with these techniques reach a general issue of complexity and precision.

Analyzing an arbitrary system would not be feasible. Our proposed analysis takes into account the application and the hardware architecture to deduce the possible interference; if one can prove that one piece of code will be executed at a time where no other parts of the application use the memory, then the analysis can safely assume that the memory accesses it performs are not penalized by other concurrent accesses.

9.1.1 *Context of the Thesis*

In this thesis, we focus on the timing aspects of an application (such as the WCET and the WCRT) in the presence of timing interference on shared resources. We propose techniques

to find tight upper-bounds on delays of shared resources accesses in multi-core and many-core systems.

The contributions of this thesis apply on two levels: at the source code level, the accesses to shared resources are analyzed to obtain accurate upper-bounds on access delays. The second contribution is on the level of binary. Using profiles obtained from external tools, an overall upper-bounds is added to the WCET to account for potential delays.

We focus on the Synchronous Data-Flow programming paradigm. This paradigm is used in avionics and automotive systems. It offers flexibility on the execution model that can easily enforce the predictability of the system. In general, our method can be applied to any dependent task graphs.

To provide accurate upper-bounds on shared resources, it is necessary that the underlying architecture is simple and predictable. Complex architectures tend to have many states which increases the over-approximation and/or harm the scalability of the analysis. In this context, we focus on the industrial platform Kalray MPPA-256. This processor is designed to offer more performance with a predictable behavior which makes it a good candidate for hard-real time systems.

To address the problem of shared resource interference, we start with a TDMA-based arbitration policy of shared resources. This work relies on the modeling of shared resources and the program semantics with Satisfiability Modulo Theory expressions. Therefore, without running the program, our method gives a tight upper-bound on the WCET. The analysis itself yields good results but have an exponential complexity. This can have some limitations on large systems. Our second contribution focuses on a more scalable approach to tightly account for the interference on shared resources. Detailed summary of our contributions is presented in the following section.

9.1.2 *Contributions*

This thesis presents two major contributions. The first one proposes an approach with Satisfiability Modulo Theory (SMT) to analyze programs running in a real-time environment with shared resources under a TDMA arbitration policy. The second contribution proposes an algorithm (with the proof of its correctness) of response time analysis of dependent tasks (particularly synchronous data-flow programs) taking into account the interference on shared resources.

9.1.2.1 *WCET Analysis of TDMA Buses*

Time Division Multiple Access arbitration policy enforces a bounded delay on shared resource accesses. This non-work conserving policy has the advantage of predictable behavior compared with other work-conserving policies, such as fixed-priority or round-robin. It offers a timing isolation which implies that cores can be analyzed separately.

We propose an alternative approach to the widely used ILP-based modeling to encode the program semantics and the arbitration policy with SMT expression. The SMT expression means: "Is there a feasible path with an execution time longer than X?", where X is a candidate upper-bound on WCET. This way, the feasible path analysis can be performed at the same time as the WCET analysis with shared resource accesses. We evaluate our

approach with the TACLeBench benchmark suite where the results show a considerable improvement compared with simple pessimistic approaches.

9.1.2.2 *Timing Analysis of Dependent Task Graphs*

Our main contribution is an algorithm to compute a static, time-driven, periodic schedule, as commonly used in hard real time systems for maximum predictability. We assume that the mapping of tasks to cores and the execution order is given (either manually or provided by a separate tool), and compute a set of release dates (offsets) and response times for each task. This is an iterative process, with release dates dependent on the response times of preceding tasks, and response times dependent on the set of co-runners, which are in turn dependent on task release dates. The process either converges on a valid, all dependence relations respected, and schedulable configuration or deems the system unschedulable with that task mapping. In the latter case a different mapping could be tried. We also give a proof of convergence of the algorithm. It is non-trivial since the usual monotonicity argument does not apply; the sequence of release dates computed at each iteration may not be monotonic.

This work aims at providing tools dedicated to the use of many-core systems in hard real-time systems. Our methods were used in the framework of the CAPACITES project. Timing analysis is required to partition and schedule tasks such that the resources are optimally used. Our contributions are directly applied on the industrial many-core Kalray MPPA-256. The results of our experimental evaluation provide a safe and predictable execution of tasks with a tight over-approximation.

9.1.2.3 *Model of an Industrial Shared Bus Arbiter*

We target applications running on the Kalray MPPA-256 many-core processor. We identify the sources of interference for an application running on a compute cluster, and provide a mathematical model for them. The model builds upon the Multi-core Response Time Analysis (MRTA) framework [Alt+15]; a generic approach to response time analysis for multi-core and many-core systems. Unlike MRTA, we consider a static, time-driven schedule, and hence cannot use the same fixed-point algorithm. Instead, we provide a novel algorithm that uses not only the mapping but also the information about *when* each task is executed; this allows us to model the interference precisely. Finally, we evaluate our approach with different benchmarks and apply it to the case study ROSACE, a realistic avionics application. The experimental results show that the new analysis leads to tighter response times than the default approach of assuming worst-case interference on each memory access.

9.2 FUTURE WORK

9.2.1 *SMT-based approaches for WCET analysis*

Our initial work on SMT encodings of shared TDMA buses opens many doors for research and improvements. SMT shows good potential as an alternative to the classic ILP-based approaches. There remain future work to be done on this topic which we explained in

Section 4.5.2. In summary: (i) Our proof-of-concept implementation needs to be adapted for realistic timing model in order to provide realistic WCET estimates that can be compared with measured execution times. (ii) To counter the scalability issue, the analysis can be made modular, operating on portions of the code to compute a global WCET estimate. (iii) Another source of non-scalability is the analysis of loops as we only considered unrolled loops. It is interesting to investigate how loops can be analyzed in SMT with minimum enrolling.

Finally, this contribution of the thesis can serve as basis to other research works in order to address the points mentioned above.

9.2.2 *Modeling the Shared Resource Accesses*

We focus in this work on the interference on the shared SRAM in compute clusters. We assume that the code and data can fit in the memory without accessing the external DRAM. MRTA [Dav+17], on which our framework is built upon, already supports an extension for interference from DRAM. As a future work, a more accurate model of Kalray's DRAM controller will be added for a more accurate and tight timing delays. Accesses to the external DRAM go through the Network-on-Chip which also needs to be accurately analyzed.

The Network-on-Chip needs to be modeled and analyzed in the context of a realistic behavior. This takes into account the constraints by the packet shaper at the NoC routers. The NoC is implemented following arrival curves from the Real-Time Calculus framework. Existing tools [DG17; BMF11] are used to compute upper-bounds on communication delays and/or arrival curves of packets on the NoC.

On the one hand, our framework can be extended to get inputs from arrival curves. The interference from the NoC can be computed using upper-bounds on the memory accesses that may occur within a time window. On the other hand, our framework can determine when the data is ready to be sent through the NoC. This information is injected back to the external NoC analysis tool.

9.2.3 *Timing Compositionality and Composability*

We assume throughout this thesis predictable architectures without timing anomalies. In our experiments and evaluation, the Kalray MPPA-256 shows deterministic and predictable behavior. Still, a formal proof of the timing compositionality of the processor is required. The inputs to our framework must be composable. For instance, the upper-bound on the number of memory accesses must be conservative and independent from co-runners.

We discuss an example with the write buffer in the data cache 6.6.2 (see Section 6.6.2). We do not address the instruction cache in this work. The instruction pre-fetch buffer requires a closer look at its behavior. A simulation-based model of the instruction cache is studied in [MP17]. This model can be adapted to interact with our tool by providing upper-bounds on accesses from the instruction cache. The challenge here is to ensure that the obtained upper-bound is safe and covers all the timing aspects regarding the caches.

Hahn et al. [HJR16] propose a method to enable compositionality for shared resource interference in multi-core systems. This work offers a different approach to timing analysis

than the traditional ones that assume timing compositionality. Our framework can be used to complement this approach and provide tight and guaranteed upper-bounds on shared resource access delays.

9.2.4 *Comparison with Real Execution*

In the short term, it is important to evaluate the precision of the analysis by comparing its results with real measurements on the platform. This requires the implementation of the full tool-chain that generates an executable binary from the high-level source code of the SDF application, including the mapping and scheduling of tasks. Synchronization routines must be lightweight and predictable. Any delay induced from such routines must also be added to the timing interference analysis. This is an on-going work by *Amaury Graillat* at *Verimag*. Preliminary results seem promising on the ROSACE case study.

9.2.5 *Application Models*

We applied our approach on Synchronous Data-Flow programs, mainly used in avionics (for example, SCADE used by Airbus). Our analysis can be extended to include other programming models used in industrial applications. It would be interesting to analyze real-time programs written in the OpenMP [DM98] and OpenCL [SGS10] standards. Model-based software design, such as MATLAB/SIMULINK, can also benefit from our approach. This design approach is used in automotive, for example in the AUTOSAR standard [Hei+04], which is also a potential market for the Kalray MPPA-256.

We focused mainly on applications running on bare machines. Our approaches can be extended to include a *Real-Time Operating System (RTOS)*. The MRTA framework, on which we built upon, has been extended to include interference from RTOS and interrupt handlers [Dav+17]. In our case, Kalray uses its own hypervisor, acting as an RTOS, to orchestrate the cores on a compute cluster. The RTOS is run on the Resource Manager which we already model from the perspective of the multi-level bus arbiter. The challenge is: (i) to derive tight bounds on the number of accesses from the RTOS that contribute in the interference on the shared memory, (ii) model the interference on the Resource Manager when tasks run system calls and other interrupt handlers.

9.2.6 *Future of Timing Analysis of Multi-Core Real-Time Systems*

This thesis focuses on multi-core and many-core systems. Such architectures promise more performance but come with more challenges than single-cores. Existing approaches from single-core processors reach their limits resulting in sub-optimal exploitation of the processor. The traditional separation between WCET community and scheduling/mapping community is not well established anymore. New approaches must operate over the timing analysis and the scheduling and mapping of tasks in a holistic way.

The work presented in this thesis provides a first step toward an efficient execution on multi-core and many-core systems. It can be extended, for instance, by adding a third fixed-

point iteration, to operate through several task mappings. The result should be an optimal mapping and scheduling that accounts for the interference on shared resources.

BIBLIOGRAPHY

[Ait] *aiT*. URL: https://www.absint.com/ait/ (cit. on p. 38).

[AGR07] Benny Akesson, Kees Goossens, and Markus Ringhofer. "Predator: A Predictable SDRAM Memory Controller." In: *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '07. Salzburg, Austria, 2007, pp. 251–256 (cit. on p. 32).

[Alt+96] Martin Alt, Christian Ferdin, Florian Martin, and Reinhard Wilhelm. "Cache Behavior Prediction by Abstract Interpretation." In: *Science of Computer Programming*. Springer, 1996, pp. 52–66 (cit. on p. 67).

[ADM12] S. Altmeyer, R. I. Davis, and C. Maiza. "Improved cache related pre-emption delay aware response time analysis for fixed priority pre-emptive systems." In: *Real-Time Systems* 48.5 (2012), pp. 499–526 (cit. on p. 88).

[Alt13] Sebastian Altmeyer. "Analysis of preemptively scheduled hard real-time systems." PhD thesis. Saarland University, 2013 (cit. on pp. 12, 15, 100).

[Alt+15] Sebastian Altmeyer, Robert I. Davis, Leandro Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. "A Generic and Compositional Framework for Multicore Response Time Analysis." In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS)*. 2015, pp. 129–138 (cit. on pp. 38, 72, 74, 94, 96, 119, 129).

[ARM04] ARM Limited, ed. *ARM7TDMI r4p1 Technical Reference Manual*. ARM Limited. 2004 (cit. on p. 19).

[ARM11] ARM Limited, ed. *Cortex-R4 and Cortex-R4F Technical Reference Manual*. ARM Limited. 2011 (cit. on p. 19).

[Axe+14] Philip Axer et al. "Building Timing Predictable Embedded Systems." In: *ACM Trans. Embed. Comput. Syst.* 13.4 (Mar. 2014), 82:1–82:37 (cit. on p. 19).

[Bal+10] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. "OTAWA: An Open Toolbox for Adaptive WCET Analysis." English. In: *SEUS 2010*. 2010, pp. 35–46 (cit. on pp. 25, 38, 47, 67, 107).

[Bal+17] Thomas Ballenthin, Boris Dreyer, Christian Hochberger, and Simon Wegener. "Hardware Support for Histogram-Based Performance Analysis of Embedded Systems." In: *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*. May 2017, pp. 1–10 (cit. on p. 40).

[Bar+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "Cvc4." In: *International Conference on Computer Aided Verification*. Springer. 2011, pp. 171–177 (cit. on p. 50).

[Bec+16] Mathias Becker, Dakshina Dasari, Borislav Nicolic, Benny Åkesson, Vincent Nélis, and Thomas Nolte. "Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform." In: *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*. 2016, pp. 14–24 (cit. on pp. 22, 33, 46, 84, 125).

[BDL04] Gerd Behrmann, Alexandre David, and Kim G. Larsen. "A Tutorial on Uppaal." In: *Formal Methods for the Design of Real-Time Systems: 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM-RT 2004*. LNCS 3185. 2004, pp. 200–236 (cit. on p. 35).

[BGJ91] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. "Synchronous programming with events and relations: the SIGNAL language and its semantics." In: *Science of Computer Programming* 16.2 (1991), pp. 103 –149 (cit. on p. 14).

[BCP02] G. Bernat, A. Colin, and S. M. Petters. "WCET analysis of probabilistic hard real-time systems." In: *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.* 2002, pp. 279–288 (cit. on p. 10).

[Ber07] Gérard Berry. "Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems: Proceedings of the GM R&D Workshop." In: 2007. Chap. SCADE: Synchronous Design and Validation of Embedded Control Software, pp. 19–33 (cit. on p. 14).

[Bie+03] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. "Bounded model checking." In: *Advances in computers* 58 (2003), pp. 117–148 (cit. on p. 49).

[Bie+13] Armin Biere, Jens Knoop, Laura Kovács, and Jakob Zwirchmayr. "The Auspicious Couple: Symbolic Execution and WCET Analysis." In: *WCET* 30 (2013), pp. 53–63 (cit. on p. 67).

[BRS11] Roman Bourgade, Christine Rochange, and Pascal Sainrat. "Predictable bus arbitration schemes for heterogeneous time-critical workloads running on multicore processors." In: *ETFA2011*. Sept. 2011, pp. 1–4 (cit. on p. 32).

[BRS13] Roman Bourgade, Christine Rochange, and Pascal Sainrat. "Predictable Two-level Bus Arbitration for Heterogeneous Task Sets." In: *Proceedings of the 26th International Conference on Architecture of Computing Systems*. ARCS'13. Prague, Czech Republic, 2013, pp. 341–351 (cit. on p. 32).

[BMF11] Marc Boyer, Jorn Migge, and Marc Fumey. "PEGASE - A Robust and Efficient Tool for Worst-Case Network Traversal Time Evaluation on AFDX." In: *SAE Technical Paper*. SAE International, Oct. 2011 (cit. on p. 130).

[Bru+08] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, and Roberto Sebastiani. "The mathsat 4 smt solver." In: *International Conference on Computer Aided Verification*. Springer. 2008, pp. 299–303 (cit. on p. 50).

[BW16] Alan Burns and Andy Wellings. *Analysable Real-time Systems: Programmed in Ada*. CreateSpace Independent Publishing Platform, 2016 (cit. on p. 13).

[But04] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Santa Clara, CA, USA: Springer-Verlag TELOS, 2004 (cit. on p. 12).

[Cap] *CAPACITES*. URL: http://capacites.minalogic.net/en/ (cit. on pp. 3, 123).

[Car+15] Thomas Carle, Dumitru Potop-Butucaru, Yves Sorel, and David Lesens. "From Dataflow Specification to Multiprocessor Partitioned Time-triggered Real-time Implementation." In: *LITES* 2.2 (2015), 01:1–01:30 (cit. on pp. 36, 37, 41).

[CRM10] Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. "Modeling Shared Cache and Bus in Multi-cores for Timing Analysis." In: *Proceedings of the 13th International Workshop on Software and Compilers for Embedded Systems*. SCOPES '10. St. Goar, Germany: ACM, 2010, 6:1–6:10 (cit. on pp. 35, 40).

[CKH16] J. Choi, D. Kang, and S. Ha. "Conservative modeling of shared resource contention for dependent tasks in partitioned multi-core systems." In: *DATE*. 2016, pp. 181–186 (cit. on pp. 38, 41).

[Chr] *Chronos*. URL: http://www.comp.nus.edu.sg/~rpembed/chronos/ (cit. on p. 39).

[Chr12] George Chrysos. "Intel® xeon phi coprocessor (codename knights corner)." In: *Hot Chips 24 Symposium (HCS), 2012 IEEE*. IEEE. 2012, pp. 1–31 (cit. on p. 17).

[CC77] Patrick Cousot and Radhia Cousot. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints." In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. 1977, pp. 238–252 (cit. on p. 10).

[CG+12] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quiñones, and F. J. Cazorla. "Measurement-Based Probabilistic Timing Analysis for Multi-path Programs." In: *2012 24th Euromicro Conference on Real-Time Systems*. July 2012, pp. 91–101 (cit. on p. 39).

[Cul+10] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza (Burguière), Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. "Predictability Considerations in the Design of Multi-Core Embedded Systems." In: *Embedded Real Time Software and Systems (ERTSS)*. 2010 (cit. on pp. 19, 23).

[DM98] Leonardo Dagum and Ramesh Menon. "OpenMP: an industry standard API for shared-memory programming." In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55 (cit. on p. 131).

[Dah+05] Anat Dahan, Daniel Geist, Leonid Gluhovsky, Dmitry Pidan, Gil Shapir, Yaron Wolfsthal, Lyes Benalycherif, Romain Kamdem, and Younes Lahbib. "Combining System Level Modeling with Assertion Based Verification." In: *6th International Symposium on Quality of Electronic Design (ISQED 2005), 21-23 March 2005, San Jose, CA, USA*. 2005, pp. 310–315 (cit. on p. 10).

[DNA15] Dakshina Dasari, Vincent Nelis, and Benny Akesson. "A framework for memory contention analysis in multi-core platforms." In: *Real-Time Systems* (2015), pp. 1–51. ISSN: 1573-1383 (cit. on pp. 35, 38–40, 46, 84).

[Dav13] R. Cok David. *The SMT-LIBv2 Language and Tools: A Tutorial*. Mar. 2013 (cit. on p. 51).

[DP60] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory." In: *J. ACM* 7.3 (June 1960), pp. 201–215 (cit. on p. 51).

[DAR16] Robert I. Davis, Sebastian Altmeyer, and Jan Reineke. *Analysis of Write-back Caches under Fixed-priority Preemptive and Non-preemptive Scheduling. Technical Report*. Tech. rep. https://www.cs.york.ac.uk/ftpdir/reports/2016/YCS/502/YCS-2016-502.pdf. University of York, 2016 (cit. on p. 100).

[Dav+17] Robert I. Davis, Sebastian Altmeyer, Leandro S. Indrusiak, Claire Maiza, Vincent Nelis, and Jan Reineke. "An extensible framework for multicore response time analysis." In: *Real-Time Systems* (July 2017) (cit. on pp. 130, 131).

[DMB08] Leonardo De Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver." In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. 2008, pp. 337–340 (cit. on p. 50).

[DRW98] Roberet P. Dick, David L. Rhodes, and Wayne Wolf. "TGFF: task graphs for free." In: *Hardware/Software Codesign, 1998. (CODES/CASHE '98) Proceedings of the Sixth International Workshop on*. Mar. 1998, pp. 97–101 (cit. on p. 119).

[DG17] Benoît Dupont de Dinechin and Amaury Graillat. "Network-on-chip service guarantees on the kalray MPPA-256 bostan processor." In: *Proceedings of the 2nd International Workshop on Advanced Interconnect Solutions and Technologies for Emerging Computing Systems*. ACM. 2017, pp. 35–40 (cit. on p. 130).

[Din+14a] Benoît Dupont de Dinechin, Yves Durand, Duco van Amstel, and Alexandre Ghiti. "Guaranteed Services of the NoC of a Manycore Processor." In: NoCArc 2014. Cambridge, United Kingdom, 2014, pp. 11–16 (cit. on p. 103).

[Din+14b] Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. "Time-critical Computing on a Single-chip Massively Parallel Processor." In: *DATE 2014*. Dresden, Germany, 2014, 97:1–97:6 (cit. on pp. 17, 20, 30, 38, 70, 85).

[DDM06] Bruno Dutertre and Leonardo De Moura. "The yices smt solver." In: *Tool paper at http://yices. csl. sri. com/tool-paper. pdf* 2.2 (2006), pp. 1–2 (cit. on p. 50).

[EC80] E. Allen Emerson and Edmund M. Clarke. "Characterizing correctness properties of parallel programs using fixpoints." In: *Automata, Languages and Programming: Seventh Colloquium Noordwijkerhout, the Netherlands July 14–18, 1980*. 1980, pp. 169–181 (cit. on p. 10).

[ER59] P. Erdös and A. Rényi. "On random graphs, I." In: *Publicationes Mathematicae (Debrecen)* 6 (1959), pp. 290–297 (cit. on p. 119).

[Fer+99] Christian Ferdinand, Florian Martin, Reinhard Wilhelm, and Martin Alt. "Cache behavior prediction by abstract interpretation." In: *Science of Computer Programming* 35.2 (1999), pp. 163 –189 (cit. on p. 23).

[FF98] Eric Fleury and Pierre Fraigniaud. "A general theory for deadlock avoidance in wormhole-routed networks." In: *IEEE Transactions on Parallel and Distributed Systems* 9.7 (July 1998), pp. 626–638 (cit. on p. 31).

[Fos09] Harry Foster. "Applied Assertion-Based Verification: An Industry Perspective." In: *Found. Trends Electron. Des. Autom.* 3.1 (2009), pp. 1–95 (cit. on p. 10).

[Gar+76] M.R Garey, R.L Graham, D.S Johnson, and Andrew Chi-Chih Yao. "Resource constrained scheduling as generalized bin packing." In: *Journal of Combinatorial Theory, Series A* 21.3 (1976), pp. 257 –298 (cit. on p. 33).

[Gia+14] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. "Mapping Mixed-criticality Applications on Multi-core Architectures." In: *Proceedings of the Conference on Design, Automation & Test in Europe*. DATE '14. 2014, 98:1–98:6 (cit. on pp. 17, 46).

[Gia+16] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, Lothar Thiele, and Benoît Dupont de Dinechin. "Mixed-criticality scheduling on cluster-based manycores with shared communication and storage resources." In: *Real-Time Systems* 52.4 (2016), pp. 399–449 (cit. on pp. 36–38, 41, 46, 96).

[Gia+17] Georgia Giannopoulou, Pengcheng Huang, Rehan Ahmed, Davide B. Bartolini, and Lothar Thiele. "Isolation Scheduling on Multicores: Model and Scheduling Approaches." In: *Real-Time Systems* 53.4 (July 2017), 614–667 (cit. on p. 33).

[Gra+18] Amaury Graillat, Matthieu Moy, Pascal Raymond, and Benoît Dupont de Dinechin. "Parallel Code Generation of Synchronous Programs for a Many-core Architecture." In: *DATE*. 2018, to appear (cit. on pp. 72, 123, 124).

[Gus+06] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Bjorn Lisper. "Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution." In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. Dec. 2006, pp. 57–66 (cit. on p. 46).

[Gus+10] Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. "Towards WCET Analysis of Multicore Architectures Using UPPAAL." In: *WCET 2010*. Vol. 15. OpenAccess Series in Informatics (OASIcs). 2010, pp. 101–112 (cit. on p. 35).

[HJR16] Sebastian Hahn, Michael Jacobs, and Jan Reineke. "Enabling Compositionality for Multicore Timing Analysis." In: *Proceedings of the 24th International Conference on Real Time and Networks Systems (RTNS)*. 2016, pp. 299–308 (cit. on pp. 103, 130).

[Hal+91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. "The synchronous data flow programming language LUSTRE." In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320 (cit. on pp. 14, 121).

[HRP17] Damien Hardy, Benjamin Rouxel, and Isabelle Puaut. "The Heptane Static Worst-Case Execution Time Estimation Tool." In: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Vol. 57. OpenAccess Series in Informatics (OASIcs). 2017, pp. 1–12 (cit. on pp. 25, 38).

[Hei+04] Harald Heinecke, Klaus-Peter Schnelle, Helmut Fennel, Jürgen Bortolazzi, Lennart Lundh, Jean Leflour, Jean-Luc Maté, Kenji Nishikawa, and Thomas Scharnhorst. "Automotive open system architecture-an industry-wide initiative to manage the complexity of emerging automotive e/e-architectures." In: *Convergence* (2004), pp. 325–332 (cit. on p. 131).

[HMM12] Julien Henry, David Monniaux, and Matthieu Moy. "PAGAI: A Path Sensitive Static Analyser." In: *Electron. Notes Theor. Comput. Sci.* 289 (Dec. 2012), pp. 15–25 (cit. on p. 58).

[Hen+14] Julien Henry, Mihail Asavoae, David Monniaux, and Claire Maïza. "How to Compute Worst-case Execution Time by Optimization Modulo Theory and a Clever Encoding of Program Semantics." In: *Proceedings of the 2014 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*. 2014, pp. 43–52 (cit. on pp. 51–53, 58, 67).

[JM97] Jean Marc Jézéquel and Bertrand Meyer. "Design by contract: the lessons of Ariane." In: *Computer* 30.1 (1997), pp. 129–130 (cit. on p. 9).

[KM17] Timon Kelter and Peter Marwedel. "Parallelism analysis: Precise WCET values for complex multi-core systems." In: *Science of Computer Programming* 133 (2017). Formal Techniques for Safety-Critical Systems (FTSCS 2014), pp. 175 –193 (cit. on p. 37).

[Kel+13] Timon Kelter, Tim Harde, Peter Marwedel, and Heiko Falk. "Evaluation of resource arbitration methods for multi-core real-time systems." In: *13th International Workshop on Worst-Case Execution Time Analysis*. Vol. 30. OpenAccess Series in Informatics (OASIcs). 2013, pp. 1–10 (cit. on p. 46).

[Kel+14] Timon Kelter, Heiko Falk, Peter Marwedel, Sudipta Chattopadhyay, and Abhik Roychoudhury. "Static Analysis of Multi-core TDMA Resource Arbitration Delays." In: *Real-Time Syst.* 50.2 (Mar. 2014), pp. 185–229 (cit. on pp. 35, 40, 48, 67).

[Kim+14] Hyoseung Kim, Dionisio De Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Rajkumar. "Bounding memory interference delay in COTS-based multi-core systems." In: *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2014 IEEE 20th*. IEEE. 2014, pp. 145–154 (cit. on p. 46).

[Kim+13] Jinwoo Kim, Hyunok Oh, Junchul Choi, Hyojin Ha, and Soonhoi Ha. "A novel analytical method for worst case response time estimation of distributed embedded systems." In: *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*. May 2013, pp. 1–10 (cit. on p. 38).

[Kop11] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer, 2011 (cit. on p. 8).

[LPT09] Kai Lampka, Simon Perathoner, and Lothar Thiele. "Analytic Real-time Analysis and Timed Automata: A Hybrid Method for Analyzing Embedded Real-time Systems." In: *Proceedings of the Seventh ACM International Conference on Embedded Software*. EMSOFT '09. Grenoble, France, 2009, pp. 107–116 (cit. on p. 37).

[Lam+14] Kai Lampka, Georgia Giannopoulou, Rodolfo Pellizzoni, Zheng Wu, and Nikolay Stoimenov. "A formal approach to the WCRT analysis of multicore systems with memory contention under phase-structured task sets." In: *Real-Time Systems* 50.5-6 (2014), pp. 736–773 (cit. on p. 37).

[LPR14] Hanbing Li, Isabelle Puaut, and Erven Rohou. "Traceability of Flow Information: Reconciling Compiler Optimizations and WCET Estimation." In: *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*. 2014, p. 97 (cit. on p. 67).

[LM95] Yau-Tsun Steven Li and Sharad Malik. "Performance Analysis of Embedded Software Using Implicit Path Enumeration." In: *Proceedings of the 32nd Annual ACM/IEEE Design Automation Conference*. DAC '95. San Francisco, California, USA, 1995, pp. 456–461 (cit. on p. 24).

[LMS15] Zhenmin Li, Avinash Malik, and Zoran Salcic. "Reducing Worst Case Reaction Time of Synchronous Programs on Chip-multiprocessors with Application-Specific TDMA Scheduling." In: *Proceedings of the 13th International Workshop on Java Technologies for Real-time and Embedded Systems*. JTRES '15. Paris, France: ACM, 2015, 11:1–11:9 (cit. on p. 36).

[LS99] Thomas Lundqvist and Per Stenström. "Timing Anomalies in Dynamically Scheduled Microprocessors." In: *Proceedings of the 20th IEEE Real-Time Systems Symposium*. RTSS '99. 1999, pp. 12– (cit. on pp. 18, 19).

[Lv+10] Mingsong Lv, Wang Yi, Nan Guan, and Ge Yu. "Combining Abstract Interpretation with Model Checking for Timing Analysis of Multicore Software." In: *Proceedings of the 2010 31st IEEE Real-Time Systems Symposium*. RTSS '10. IEEE Computer Society, 2010, pp. 339–349 (cit. on pp. 35, 63).

[Man+17] Renato Mancuso, Rodolfo Pellizzoni, Neriman Tokcan, and Marco Caccamo. "WCET Derivation under Single Core Equivalence with Explicit Memory Budget Assignment." In: *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Vol. 76. Leibniz International Proceedings in Informatics (LIPIcs). 2017, 3:1–3:23 (cit. on p. 37).

[MHP17] Sébastien Martinez, Damien Hardy, and Isabelle Puaut. "Quantifying WCET reduction of parallel applications by introducing slack time to limit resource contention." In: *Proceedings of the 25rd International Conference on Real Time and Networks Systems (RTNS)*. 2017, to appear (cit. on pp. 123, 124).

[MP17] Omayma Matoussi and Frédéric Pétrot. "Modeling instruction cache and instruction buffer for performance estimation of VLIW architectures using native simulation." In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017*. Mar. 2017, pp. 266–269 (cit. on p. 130).

[Mel+15] Alessandra Melani, Marko Bertogna, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, and Giorgio Buttazzo. "Memory-Processor Co-Scheduling in Fixed Priority Systems." In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems (RTNS)*. 2015, pp. 87–96 (cit. on pp. 22, 33, 72, 84, 108, 125).

[Mel+12] Diego Melpignano, Luca Benini, Eric Flamand, Bruno Jego, Thierry Lepley, Germain Haugou, Fabien Clermidy, and Denis Dutoit. "Platform 2012, a Many-core Computing Accelerator for Embedded SoCs: Performance Evaluation of Visual Analytics Applications." In: *Proceedings of the 49th Annual Design Automation Conference*. DAC. 2012, pp. 1137–1142 (cit. on p. 17).

[Mou15] Jad Mouawad. "F.A.A Orders Fix for Possible Power Loss in Boeing 787." In: *The New York Times* (May 1, 2015). Available: https://www.nytimes.com/2015/05/01/business/faa-orders-fix-for-possible-power-loss-in-boeing-787.html [Last accessed: September 8, 2017] (cit. on p. 9).

[NHP15] Viet Anh Nguyen, Damien Hardy, and Isabelle Puaut. "Scheduling of parallel applications on many-core architectures with caches: bridging the gap between WCET analysis and schedulability analysis." In: *9th Junior Researcher Workshop on Real-Time Computing (JRWRTC 2015)*. 2015 (cit. on p. 71).

[NHP17] Viet Anh Nguyen, Damien Hardy, and Isabelle Puaut. "Cache-Conscious Offline Real-Time Task Scheduling for Multi-Core Processors." In: *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Vol. 76. Leibniz International Proceedings in Informatics (LIPIcs). 2017, 14:1–14:22 (cit. on pp. 33, 71, 123).

[Now+14] Jan Nowotsch, Michael Paulitsch, Daniel Buhler, Henrik Theiling, Simon Wegener, and Michael Schmidt. "Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement." In: *ECRTS*. 2014, pp. 109–118 (cit. on p. 11).

[NYP17] Vincent Nélis, Patrick Meumeu Yomsi, and Luís Miguel Pinho. "The P-SOCRATES Timing Analysis Methodology for Parallel Real-Time Applications Deployed on Many-Core Platforms." In: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Vol. 57. OpenAccess Series in Informatics (OASIcs). 2017, pp. 1–9 (cit. on p. 40).

[OLF17] Dominic Oehlert, Arno Luppold, and Heiko Falk. "Bus-Aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems." In: *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*. Vol. 76. Leibniz International Proceedings in Informatics (LIPIcs). 2017, 1:1–1:22 (cit. on p. 36).

[Pso] *P-SOCRATES*. URL: http://www.p-socrates.eu (cit. on p. 40).

[Pag+14] C. Pagetti, D. Saussie, R. Gratia, E. Noulard, and P. Siron. "The ROSACE case study: From Simulink specification to multi/many-core execution." In: RTAS 2014. 2014, pp. 309–318 (cit. on pp. 20, 21, 116).

[Pan+15] M. Panic, J. Abella, C. Hernandez, E. Quiñones, T. Ungerer, and F. J. Cazorla. "Enabling TDMA Arbitration in the Context of MBPTA." In: *2015 Euromicro Conference on Digital System Design*. Aug. 2015, pp. 462–469 (cit. on p. 39).

[Pao+09] Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla, and Mateo Valero. "An Analyzable Memory Controller for Hard Real-Time CMPs." In: *IEEE Embedded Systems Letters* 1.4 (Dec. 2009), pp. 86–90 (cit. on p. 32).

[Pao+13] Marco Paolieri, Jörg Mische, Stefan Metzlaff, Mike Gerdes, Eduardo Quiñones, Sascha Uhrig, Theo Ungerer, and Francisco J. Cazorla. "A Hard Real-time Capable Multi-core SMT Processor." In: *ACM Trans. Embed. Comput. Syst.* 12.3 (Apr. 2013), 79:1–79:26 (cit. on p. 32).

[Pel+11] R. Pellizzoni, E. Betti, S. Bak, G. Yao, J. Criswell, M. Caccamo, and R. Kegley. "A Predictable Execution Model for COTS-Based Embedded Systems." In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. Apr. 2011, pp. 269–279 (cit. on pp. 21, 33).

[Pel+08] Rodolfo Pellizzoni, Bach D. Bui, Marco Caccamo, and Lui Sha. "Coscheduling of CPU and I/O Transactions in COTS-Based Embedded Systems." In: *Proceedings of the 2008 Real-Time Systems Symposium*. 2008, pp. 221–231 (cit. on p. 22).

[Pel+10] Rodolfo Pellizzoni, Andreas Schranzhofer, Jian-Jia Chen, Marco Caccamo, and Lothar Thiele. "Worst Case Delay Analysis for Memory Interference in Multicore Systems." In: *DATE*. Dresden, Germany, 2010, pp. 741–746 (cit. on p. 38).

[Per+16a] Quentin Perret, Pascal Maurère, Éric Noulard, Claire Pagetti, Pascal Sainrat, and Benoît Triquet. "Mapping Hard Real-time Applications on Many-core Processors." In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. 2016, pp. 235–244 (cit. on pp. 17, 41).

[Per+16b] Quentin Perret, Pascal Maurere, Eric Noulard, Claire Pagetti, Pascal Sainrat, and Benoît Triquet. "Predictable composition of memory accesses on many-core processors." In: *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*. 2016 (cit. on pp. 32, 36).

[Per+16c] Quentin. Perret, Pascal Maurere, Eric Noulard, Claire Pagetti, Pascal Sainrat, and Benoît Triquet. "Temporal Isolation of Hard Real-Time Applications on Many-Core Processors." In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2016, pp. 1–11 (cit. on p. 32).

[Pha+13] Linh T. X. Phan, Meng Xu, Jaewoo Lee, Insup Lee, and Oleg Sokolsky. "Overhead-aware compositional analysis of real-time systems." In: *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2013, pp. 237–246 (cit. on p. 103).

[Pro] *PROXIMA*. URL: http://www.proxima-project.eu (cit. on p. 39).

[PNP13] W. Puffitsch, E. Noulard, and C. Pagetti. "Mapping a multi-rate synchronous language to a many-core processor." In: *RTAS 2013*. 2013, pp. 293–302 (cit. on pp. 17, 33).

[Rad+12] 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." In: *ACM Trans. Archit. Code Optim.* 8.4 (Jan. 2012), 34:1–34:25 (cit. on p. 11).

[Ray14] Pascal Raymond. "A general approach for expressing infeasibility in Implicit Path Enumeration Technique." In: *2014 International Conference on Embedded Software (EMSOFT)*. Oct. 2014, pp. 1–9 (cit. on pp. 25, 46).

[Ray+15] Pascal Raymond, Claire Maiza, Catherine Parent-Vigouroux, Fabienne Carrier, and Mihail Asavoae. "Timing analysis enhancement for synchronous program." English. In: *Real-Time Systems* (2015), pp. 1–29 (cit. on pp. 46, 67).

[Rei08] J. Reineke. *Caches in WCET Analysis: Predictability, Competitiveness, Sensitivity*. epubli, 2008 (cit. on pp. 16, 19, 100).

[Rei+06] Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. "A Definition and Classification of Timing Anomalies." In: *WCET* 4 (2006) (cit. on p. 46).

[Rei+11] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. "PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation." In: *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '11. Taipei, Taiwan, 2011, pp. 99–108 (cit. on p. 32).

[Rih+15] Hamza Rihani, Matthieu Moy, Claire Maiza, and Sebastian Altmeyer. "WCET analysis in shared resources real-time systems with TDMA buses." In: *RTNS 2015*. 23rd International Conference on Real-Time Networks and Systems. Nov. 2015 (cit. on pp. 3, 53).

[Rih+16] Hamza Rihani, Matthieu Moy, Claire Maiza, Robert I. Davis, and Sebastian Altmeyer. "Response Time Analysis of Synchronous Data Flow Programs on a Many-Core Processor." In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems (RTNS)*. 2016, pp. 67–76 (cit. on p. 3).

[Ros+07] Jacob Rosèn, Alexandru Andrei, Petru Eles, and Zebo Peng. "Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip." In: *RTSS 2007*. 2007 (cit. on pp. 34–36, 41).

[Sai+15] Selma Saidi, Rolf Ernst, Sascha Uhrig, Henrik Theiling, and Benoît Dupont de Dinechin. "The shift to multicores in real-time and safety-critical systems." In: *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. Oct. 2015, pp. 220–229 (cit. on pp. 29, 31).

[Sai+14] Abusayeed Saifullah, David Ferry, Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher D Gill. "Parallel real-time scheduling of DAGs." In: *IEEE Transactions on Parallel and Distributed Systems* 25.12 (2014), pp. 3242–3252 (cit. on p. 113).

[SGM17] Luca Santinelli, Fabrice Guet, and Jerome Morio. "Revising Measurement-Based Probabilistic Timing Analysis." In: *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. Apr. 2017, pp. 199–208 (cit. on p. 39).

[Sch+08] Simon Schliecker, Mircea Negrean, Gabriela Nicolescu, Pierre Paulin, and Rolf Ernst. "Reliable Performance Analysis of a Multicore Multithreaded System-on-chip." In: *Proceedings of the 6th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*. CODES+ISSS '08. Atlanta, GA, USA, 2008, pp. 161–166 (cit. on p. 35).

[Sch+14] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. *Patmos Reference Handbook*. 2014 (cit. on p. 19).

[Sch+15] Martin Schoeberl et al. "T-CREST: Time-predictable multi-core architecture for embedded systems." In: *Journal of Systems Architecture* 61.9 (2015), pp. 449 –471 (cit. on pp. 20, 30, 32).

[SCT10] Andreas Schranzhofer, Jian-Jia Chen, and Lothar Thiele. "Timing Analysis for TDMA Arbitration in Resource Sharing Systems." In: *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*. 2010, pp. 215–224 (cit. on pp. 22, 33, 35, 38, 46).

[Sch+10] Andreas Schranzhofer, Rodolfo Pellizzoni, Jian-Jia Chen, Lothar Thiele, and Marco Caccamo. "Worst-case Response Time Analysis of Resource Access Models in Multi-core Systems." In: *Proceedings of the 47th Design Automation Conference*. DAC '10. Anaheim, California, 2010, pp. 332–337 (cit. on p. 35).

[SS16] Stefanos Skalistis and Alena Simalatsar. "Worst-Case Execution Time Analysis for Many-Core Architectures with NoC." In: *Proceedings of the 14th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS*. Springer International Publishing, 2016, pp. 211–227 (cit. on p. 96).

[Sou+07] Jean Souyris, Erwan Le Pavec, Guillaume Himbert, Guillaume Borios, Victor Jégu, and Reinhold Heckmann. "Computing the Worst Case Execution Time of an Avionics Program by Abstract Interpretation." In: *5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*. Vol. 1. OpenAccess Series in Informatics (OASIcs). 2007 (cit. on p. 22).

[SGS10] John E Stone, David Gohara, and Guochun Shi. "OpenCL: A parallel programming standard for heterogeneous computing systems." In: *Computing in science & engineering* 12.3 (2010), pp. 66–73 (cit. on p. 131).

[Swe] *SWEET*. URL: http://www.mrtc.mdh.se/projects/wcet/sweet/ (cit. on p. 38).

[Ten14] Pranav Tendulkar. "Mapping and Scheduling on Multi-core Processors using SMT Solvers. (placement et ordonnancement sur les processeurs multi-core en utilisant un solveur SMT)." PhD thesis. Joseph Fourier University, Grenoble, France, 2014 (cit. on p. 17).

[Ten+14] Pranav Tendulkar, Peter Poplavko, Ioannis Galanommatis, and Oded Maler. "Many-Core Scheduling of Data Parallel Applications Using SMT Solvers." In: *2014 17th Euromicro Conference on Digital System Design*. Aug. 2014, pp. 615–622 (cit. on p. 41).

[The+03] S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. "An abstract interpretation-based timing validation of hard real-time avionics software." In: *2003 International Conference on Dependable Systems and Networks, 2003. Proceedings.* June 2003, pp. 625–632 (cit. on p. 22).

[Til12] Tilera Corporation, ed. *Tile Processor Architecture Overview for the TILE-Gx Series*. Tilera Corporation. 2012 (cit. on pp. 17, 20, 30).

[TK02] Takao Tobita and Hironori Kasahara. "A standard task graph set for fair evaluation of multiprocessor scheduling algorithms." In: *Journal of Scheduling* 5.5 (2002), pp. 379–394 (cit. on p. 111).

[Tou+17] Valentin Touzeau, Claire Maïza, David Monniaux, and Jan Reineke. "Ascertaining Uncertainty for Efficient Exact Cache Analysis." In: *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. 2017, pp. 22–40 (cit. on p. 23).

[VYF16] P. K. Valsan, H. Yun, and F. Farshchi. "Taming Non-blocking Caches to Improve Isolation in Multicore Real-Time Systems." In: *RTAS*. Apr. 2016, pp. 161–172 (cit. on p. 11).

[WN15] Jörg Walter and Wolfgang Nebel. "Energy–Aware Mapping and Scheduling of Large–Scale Macro Data–Flow Applications." In: *1st International Workshop on Investigating Dataflow in Embedded Computing Architecture*. 2015 (cit. on pp. 17, 33).

[Weg17] Simon Wegener. "Towards Multicore WCET Analysis." In: *17th International Workshop on Worst-Case Execution Time Analysis (WCET 2017)*. Vol. 57. OpenAccess Series in Informatics (OASIcs). 2017, pp. 1–12 (cit. on pp. 31, 32).

[Wil+08] Reinhard Wilhelm et al. "The Worst-case Execution-time Problem – Overview of Methods and Survey of Tools." In: *ACM Trans. Embed. Comput. Syst.* 7.3 (2008), 36:1–36:53 (cit. on pp. 10, 22, 25, 107).

[YYA16] Kecheng Yang, Ming Yang, and James H. Anderson. "Reducing Response-Time Bounds for DAG-Based Task Systems on Heterogeneous Multicore Platforms." In: *Proceedings of the 24th International Conference on Real-Time Networks and Systems*. RTNS '16. Brest, France: ACM, 2016, pp. 349–358 (cit. on p. 33).

ABSTRACT

Predictability is of paramount importance in real-time and safety-critical systems, where non-functional properties – such as the timing behavior – have high impact on the system's correctness. As many safety-critical systems have a growing performance demand, classical architectures, such as single-cores, are not sufficient anymore. One increasingly popular solution is the use of multi-core systems, even in the real-time domain. Recent many-core architectures, such as the Kalray MPPA, were designed to take advantage of the performance benefits of a multi-core architecture while offering certain predictability. It is still hard, however, to predict the execution time due to interferences on shared resources (e.g., bus, memory, etc.).
To tackle this challenge, Time Division Multiple Access (TDMA) buses are often advocated. In the first part of this thesis, we are interested in the timing analysis of accesses to shared resources in such environments. Our approach uses Satisfiability Modulo Theory (SMT) to encode the semantics and the execution time of the analyzed program. To estimate the delays of shared resource accesses, we propose an SMT model of a shared TDMA bus. An SMT-solver is used to find a solution that corresponds to the execution path with the maximal execution time. Using examples, we show how the worst-case execution time estimation is enhanced by combining the semantics and the shared bus analysis in SMT.
In the second part, we introduce a response time analysis technique for Synchronous Data Flow programs. These are mapped to multiple parallel dependent tasks running on a compute cluster of the Kalray MPPA-256 many-core processor. The analysis we devise computes a set of response times and release dates that respect the constraints in the task dependency graph. We derive a mathematical model of the multi-level bus arbitration policy used by the MPPA. Further, we refine the analysis to account for (i) release dates and response times of co-runners, (ii) task execution models, (iii) use of memory banks, (iv) memory accesses pipelining. Further improvements to the precision of the analysis were achieved by considering only accesses that block the emitting core in the interference analysis. Our experimental evaluation focuses on randomly generated benchmarks and an avionics case study.
Keywords: shared resource interference, many-core processors, worst-case execution time, response time, timing analysis, real-time systems.

RÉSUMÉ

La prédictibilité est un aspect important des systèmes temps-réel critiques. Garantir la fonctionnalité de ces systèmes passe par la prise en compte des contraintes temporelles. Les architectures mono-cœurs traditionnelles ne sont plus suffisantes pour répondre aux besoins croissants en performance de ces systèmes. De nouvelles architectures multi-cœurs sont conçues pour offrir plus de performance mais introduisent d'autres défis. Dans cette thèse, nous nous intéressons au problème d'accès aux ressources partagées dans un environnement multi-cœur.
La première partie de ce travail propose une approche qui considère la modélisation de programme avec des formules de satisfiabilité modulo des théories (SMT). On utilise un solveur SMT pour trouver un chemin d'exécution qui maximise le temps d'exécution. On considère comme ressource partagée un bus utilisant une politique d'accès multiple à répartition dans le temps (TDMA). On explique comment la sémantique du programme analysé et le bus partagé peuvent être modélisés en SMT. Les résultats expérimentaux montrent une meilleure précision en comparaison à des approches simples et pessimistes.
Dans la deuxième partie, nous proposons une analyse de temps de réponse de programmes à flot de données synchrones s'exécutant sur un processeur pluri-cœur. Notre approche calcule l'ensemble des dates de début d'exécution et des temps de réponse en respectant la contrainte de dépendance entre les tâches. Ce travail est appliqué au processeur pluri-cœur industriel Kalray MPPA-256. Nous proposons un modèle mathématique de l'arbitre de bus implémenté sur le processeur. De plus, l'analyse de l'interférence sur le bus est raffinée en prenant en compte : (i) les temps de réponse et les dates de début des tâches concurrentes, (ii) le modèle d'exécution, (iii) les bancs mémoires, (iv) le *pipeline* des accès à la mémoire. L'évaluation expérimentale est réalisé sur des exemples générés aléatoirement et sur un cas d'étude d'un contrôleur de vol.
Mots clés : interférences sur ressources partagées, processeurs pluri-cœurs, temps de réponse, temps d'exécution pire-cas, analyse temporelle, système temps-réel.