Implantation sûre d'applications temps-réel critiques sur plateforme pluri-coeur

Matheus Schuh

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : VERIMAG

Implantation sûre d'applications temps-réel critiques sur plateforme pluri-coeur

Safe Implementation of Hard Real-Time Applications on Many-Core Platforms

Présentée par :

Matheus SCHUH

Direction de thèse :

Claire MAIZA MCF,	Directrice de thèse
Pascal RAYMOND CNRS	Co-encadrant de thèse

Rapporteurs :

EDUARDO TOVAR
Professeur, School of Engineering ISEP-IPP
CLAIRE PAGETTI
Ingénieur de recherche, ONERA

Thèse soutenue publiquement le **31 mai 2022**, devant le jury composé de :

CLAIRE MAIZA Maître de conférences, GRENOBLE INP	Directrice de thèse
EDUARDO TOVAR Professeur, School of Engineering ISEP-IPP	Rapporteur
CLAIRE PAGETTI Ingénieur de recherche, ONERA	Rapporteure
FREDERIC PETROT Professeur des Universités, GRENOBLE INP	Président
JOËL GOOSSENS Professeur, Université Libre de Bruxelles	Examinateur
ISABELLE PUAUT Professeur des Universités, UNIVERSITE RENNES 1	Examinatrice

Invités :

PASCAL RAYMOND
Chargé de Recherche, CNRS DELEGATION ALPES
BENOÎT DINECHIN
Ingénieur docteur, KALRAY

THÈSE

Pour obtenir le grade de

DOCTEUR DE LA COMMUNAUTÉ UNIVERSITÉ GRENOBLE ALPES

Spécialité : **Informatique**
Arrêté ministériel : 25 mai 2016

Présentée par

Matheus SCHUH

Thèse dirigée par **Claire MAIZA**, MCF
co-encadrée par **Pascal RAYMOND**, CNRS
et co-encadrée par **Benoît DINECHIN**, Kalray

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

Implantation sûre d'applications temps-réel critiques sur plateforme pluri-cœurs

Safe Implementation of Hard Real-Time Applications on Many-Core Platforms

Thèse soutenue publiquement le **31 mai 2022**,
devant le jury composé de :

Madame CLAIRE MAIZA
Maître de conférences, Grenoble INP, Directrice de thèse
Monsieur EDUARDO TOVAR
Professeur, ISEP-IPP, Rapporteur
Madame CLAIRE PAGETTI
Ingénieur de recherche, ONERA, Rapporteure
Monsieur FRÉDÉRIC PÉTROT
Professeur des universités, Grenoble INP, Président
Monsieur JOËL GOOSSENS
Professeur, Université Libre de Bruxelles, Examinateur
Madame ISABELLE PUAUT
Professeur des universités, Université de Rennes, Examinatrice

SAFE IMPLEMENTATION OF HARD REAL-TIME APPLICATIONS ON MANY-CORE PLATFORMS

MATHEUS SCHUH

An Integrated Implementation Workflow with Execution Models, Allocation Methods and Interference Analysis

PhD Candidate
Verimag / Kalray
Université Grenoble Alpes

May 2022

Matheus Schuh: *Safe Implementation of Hard Real-Time Applications on Many-Core Platforms,* An Integrated Implementation Workflow with Execution Models, Allocation Methods and Interference Analysis, May 2022

ABSTRACT

Hard real-time systems are designed to be functionally correct, but also require the guarantee of timing constraints. Completing the task at hand within a given deadline is part of the specification and failing to accomplish this can lead to serious consequences. Some examples of such systems are the central command of an airplane, electronic control units inside a car and a power plant monitoring room.

Multi/many-core architectures tend to be used in such systems. On top of having multiple cores that can run programs concurrently, these SoCs may contain several cache levels, local and global shared memories, buses or interconnections for communication. Moreover, the cores themselves have dynamic components in their pipeline such as branch prediction or instruction reordering. All these features are extremely useful to boost the average performance, but raise huge problems for hard real-time systems, as they introduce timing unpredictability.

The predictability of these systems is directly connected to being able to compute the worst-case response time of an application on a given architecture. Additionally, on multi/many-core architectures, when numerous cores access shared hardware resources at the same time, they interfere with each other, mutually slowing them down. To guarantee the respect of timing constraints of a real-time system, the interference sources must be identified and taken into account.

The use of multi/many-core architectures on safety-critical real-time systems is increasing in the industry, as well as being an intense topic of research. This thesis provides solutions and comparative studies on several problems raised by the implementation of critical applications on such platforms. We focus on providing an integrated approach in order to confidently use multi/many-core architectures for hard real-time systems. This integrated workflow covers the choice of an execution model, a strategy to map and schedule tasks and a hardware model to provide safe bounds on the response time.

The critical application to be analyzed is represented in the form of a Directed Acyclic Graph (DAG), with precedence constraints between nodes and explicit communication. This application can be issued from synchronous data flow languages or any other language or model-based development environment providing a DAG at the end of the compilation process. Most of our case studies come from the SCADE tool, some of them being industrial case studies.

The target architecture, the Kalray MPPA3 is a COTS processor but with interesting characteristics that make it a good candidate for real-time systems. At the core level, it has in-order pipeline and private caches with a predictable replacement policy. At the cluster level, it provides low latency scratchpad memory and predictable arbitration policies. At the SoC level, it provides an AXI bus connecting the different clusters with constant traversal time.

We present and explore several execution models that help to provide a predictable execution on multi-core platform. They are applied to the many-core processors Kalray MPPA2 and MPPA3 and compared to enlighten the best approach in terms of task phased execution and memory access restrictions. We show that in our context, the isolation of tasks executed concurrently is too expensive in terms of response-time. The best execution

model corresponds to a development process where interference analysis between task memory accesses are integrated in the implementation step.

An additional improvement that has a significant impact on the overall response time of a program is the task mapping and scheduling on a given platform. With a high number of cores and clusters, it has become essential to provide a good positioning and ordering, at the risk of under utilizing the potential parallelism. Therefore we provide an initial work with multiple steps, taking into account the local memory use, communication cost and clusterization. We show that our memory use criteria is a good one to be used in future work.

The Kalray MPPA3 is the main target architecture of this work and for its use in hard real-time systems, the arbitration points and shared resource access delay have been analyzed. A hardware model of intra and inter-cluster memory accesses is developed, combined with a response time analysis framework.

Throughout this thesis several extensions and improvements were made to different industrial and academic tools: SCADE code generator, a multi-core interference analysis and a high-level hardware model.

RÉSUMÉ

Les systèmes temps-réel critiques sont conçus pour garantir non seulement les fonctionnalités mais aussi des contraintes temporelles. En effet, garantir la fin d'exécution d'une tâche avant une date limite donnée fait partie de la spécification de tels systèmes, où toute erreur d'exécution peut porter atteinte à une vie humaine. Parmi ces systèmes, nous pouvons citer le domaine des transports (avionique, automobile, ferroviaire) mais aussi le domaine médical ou celui de l'énergie, en particulier nucléaire.

Cette dernière décennie, ces systèmes ont été mis en œuvre sur des plateformes complexes telles que des plateformes multi/pluri-cœurs. Ces plateformes présentent l'intérêt d'avoir plusieurs unités d'exécution et donc un parallélisme d'exécution qui rend leur utilisation plus efficace. Cependant, cette efficacité est possible grâce à l'utilisation de ressources partagées (mémoire, canaux de communication, ...). Lorsque plusieurs cœurs accèdent à une ressource partagée simultanément, l'accès génère des délais d'attente que l'on appelle interférences. Pour garantir l'exécution d'applications temps-réel critique sur de telles plateformes, il faut garantir que toute interférence est identifiée et a un effet borné dans le temps. La prédictibilité de ces systèmes est garantie par le calcul de temps d'exécution pire-cas pour une application et une plateforme données. Dans le cas de plateformes multi-cœurs, les coûts des interférences doivent être pris en compte dans les phases d'analyse pour garantir les contraintes temporelles.

L'utilisation de plateformes multi/pluri-cœurs pour les systèmes temps-réel critiques est de plus en plus répandue dans l'industrie, et elle est un sujet d'étude bien représenté dans le monde de la recherche. Cette thèse fournit des solutions et études comparatives concernant plusieurs points clés de l'implémentation d'applications temps-réel critiques sur plateformes multi/pluri-cœurs. En particulier, nous proposons une approche qui intègre les phases d'implémentation et d'analyse temporelle (délai d'interférence, calcul de temps de réponse). Notre flot de conception se concentre en particulier sur le choix de modèles d'exécution, sur le placement des tâches sur les cœurs et en mémoire ainsi que leur ordonnancement, et sur une modélisation de la plateforme cible et des interférences qu'elle induit. Au final, le flot proposé permet d'obtenir une implémentation sûre des applications critiques, avec des bornes garanties sur leurs temps de réponse.

Les applications temps-réel critique que nous étudions sont celles qui peuvent être représentées sous forme de graphe dirigé acyclique (Directed Acyclic Graph, DAG), où les nœuds correspondent au code fonctionnel (tâches), et les arrêtes représentent des contraintes de précédence et/ou des communications de données. Ce type de représentation est typique des typique de ce que produisent les environnement de développement de haut-niveu utilisés dans l'industrie. En particulier, dans cette thèse nous étudions majoritairement des applications générées par l'outil SCADE, dont des études de cas directement extraites, ou largement inspirées d'applications industrielles.

Dans le cadre de cette thèse, la plateforme cible est le processeur Kalray MPPA (génération 2 et 3) qui offre des caractéristiques intéressantes vis-à-vis de la prédictibilité. Les cœurs ont des architectures simples avec pipeline à exécution dans l'ordre et des mémoires caches privées avec politique de remplacement prédictible. Cette architecture est pluri-cœurs, c'est-à-dire qu'elle offre deux niveaux de parallélisme : un ensemble

de cœurs de calculs parallèles forme un multi-cœur (cluster), et l'ensemble des clusters parallèles forme la plateforme complète. Au niveau d'un cluster (multi-cœur), on retrouve un mémoire à accès rapide de type scratchpad avec des arbitres prédictibles. Au niveau de la plateforme complète, les communications entre clusters peuvent s'effectuer par un bus AXI qui offre un temps constant de traversée.

Dans cette thèse, nous étudions différents modèles d'exécution qui permettent d'obtenir plus de prédictibilité sur des plateformes pluri/multi-cœurs. Une méthode d'implémentation de ces différents modèles sur Kalray MPPA2 et MPPA3 est proposée, qui permet de les comparer en terme de type de modèle (exécution phasée) et d'implémentation en isolation complète ou partielle (avec prise en compte des délais d'interférence). Ces travaux ont montré qu'il était, dans notre contexte, plus intéressant de ne pas utiliser une isolation complète mais de modéliser de façon précise les interférences et leurs effets.

Pour les phases de placement sur les cœurs et la mémoire ainsi que l'ordonnancement qui ont un effet important sur le temps de réponse global d'une application, nous avons mis l'accent sur l'utilisation mémoire et les coûts de communication. Un algorithme préliminaire nous permet de montrer que ces deux facteurs sont primordiaux et devraient être centraux pour les travaux futurs.

L'utilisation d'une nouvelle plateforme (Kalray MPPA3) a nécessité une modélisation fine des temps d'accès aux ressources partagées, ainsi que le développement d'une méthode de calcul des délais dûs aux interférences. Un modèle des communications (intra et inter-clusters) a été développé et couplé à un outil d'analyse de temps de réponse.

Pour tous ces travaux, des extensions d'outils académiques et industriels ont été réalisées : modèle du MPPA3 pour l'analyse temporelle (Multi-Core Interference Analysis, MIA) et le développement d'applications (SHIM), génération du code de communication et d'orchestration pour les applications produite par SCADE.

PUBLICATIONS

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

[1] Maximilien Dupont de Dinechin, Matheus Schuh, Matthieu Moy, and Claire Maiza. *Scaling Up the Memory Interference Analysis for Hard Real-Time Many-Core Systems (Full Version)*. Tech. rep. TR 2019-1. Verimag Research Report, 2019.

[2] Maximilien Dupont de Dinechin, Matheus Schuh, Matthieu Moy, and Claire Maiza. "Scaling up the memory interference analysis for hard real-time many-core systems." In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 330–333.

[3] Matheus Schuh. "Implementation of Real-Time Data-Flow Synchronous Programs on a Many-Core Architecture." In: *ACACES 2019 Poster Abstracts*. HiPEAC (High Performance, Embedded Architecture, and Compilation), July 2019, pp. 151–154. ISBN: 978-88-905806-7-3.

[4] Matheus Schuh. *Time Critical Computing on the MPPA processor*. Tech. rep. KETD-417. Kalray S.A info@kalray.eu, Nov. 2019.

[5] Matheus Schuh, Claire Maiza, Joël Goossens, Pascal Raymond, and Benoît Dupont de Dinechin. "A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory." In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2020, pp. 283–295.

ACKNOWLEDGMENTS

In this short piece of text I would like to deliver my deepest gratitude towards everyone that has been at my side throughout these 3 years and 3 months of thesis.

First of all, to my academic thesis supervisors: *Claire* and *Pascal*. Thanks for the guidance and patience even on the most difficult moments. Your scientific knowledge and experience has pushed me forward and allowed me to deliver this work with relevant contributions to the real-time systems community. It was a pleasure working besides you and I hope we can collaborate together again shortly.

To my industrial thesis supervisor *Benoît*, who gave me the opportunity of an internship at Kalray in 2016 and then trusted me again with this CIFRE thesis. Thanks for all your technical expertise and for always supporting and highlighting the importance of research, even on an industry environment.

My gratitude goes also to the jury who accepted to evaluate my PhD thesis and on their decision to entitle me with the Doctor degree. I thank *Claire* and *Eduardo* for reviewing my thesis and also *Frédéric*, *Joël* and *Isabelle* for being examiners.

Thanks to all my colleagues at Verimag, in special the ones at the Shared Resources team with whom I interacted the most scientifically: *Lionel, Erwan, David and Florence.* To all PhD candidates and post-docs that I have meet along the way, who became friends through uncountable lunches and coffee breaks: *Hadi, Vincent, Hamzah, Aina, Thomas, Akshay, Hakim, Marie*.

The implementation of everything that is presented in this thesis would not have been possible without the expertise of everyone at Kalray. A special thanks to *Vincent* and *Arnaud* for all the hardware discussions and also to *Julien* and *Pierre* for their software and debug knowledge with low-level code.

Aos meus pais *João* e *Claise*, devo tudo o que eu sou e me tornei hoje à vocês. Obrigado por entenderem e incentivarem minhas ambições pessoais e profissionais, mesmo que isso tenha significado ficar longe de vocês por tanto tempo. Obrigado por estarem sempre ao meu lado nas inúmeras ligações que tivemos e por compreenderem os momentos difíceis que atravessei durante a tese. Essa conquista é tanto minha quanto de vocês!

Aos amigos brasileiros próximos geograficamente: *Gabriel, William, Pedro, Lucas, Igor, João, Vinícius* e aos que estão longe mas sempre perto: *Felipe, Bruno, Guilherme, Sara*. Obrigado por aguentarem todas as vezes em que o assunto tese era trazido à mesa e por todas as palavras de incentivo ao longo desses anos.

Um agradecimento especial à *Gabriela*, que embarcou nessa aventura comigo, trazendo sempre toda sua alegria, positividade e apoio incondicional. Guardo um carinho eterno por ti, pois sei que meu voo teria sido bem menor sem a tua presença.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

LIST OF ACRONYMS

AER Acquisition Execution Restitution 14, 43, 94
AET Actual Execution Time 54
ALU Arithmetic and Logic Unit 23
APIC Advanced Programmable Interrupt Controller 28, 94
ASAP As Soon As Possible 32
AXI Advanced eXtensible Interface 22, 23, 25–27, 29, 53, 67, 69, 74, 75, 78, 90, 116, 123–125

BCU Branch and Control Unit 24
BF Best-Fit 32

CC Compute Cluster 23, 25–27
CCR Computation-Communication Ratio 106–108, 113
COTS Commercial Off-The-Shelf 9, 20, 23, 41, 43, 53, 68, 123

DAG Directed Acyclic Graph 2, 3, 17, 31, 32, 55–61, 63, 70, 71, 77, 82–84, 105, 114–116, 118–121, 123–125
DAL Design Assurance Level 13
DC Data Cache 24, 66, 73
DDR Double Data Rate 26, 68, 69, 72, 76
DFG Data-Flow Graph xvii, 39, 40, 42, 44, 47, 51
DMA Direct Memory Access 25, 27, 69, 76, 77, 98, 126
DRR Deficit Round-Robin 67, 74, 78, 90
DSU Debug Support Unit 25

ECU Electronic Control Unit 109
EDF Earliest Deadline First 32

FAA Federal Aviation Administration 9
FIFO First In First Out 20
FP Fixed Priority 66, 72, 73, 77, 90
FPU Floating-Point Unit 23
FSM Finite State Machine 10

GPR General Purpose Register 24

HLFET Highest Level First with Estimated Time 59

I/O Input/Output 27
IC Instruction Cache 24, 66, 72, 73, 77
ILP Instruction Level Parallelism 23, 33
IMA Integrated Modular Avionics 32
ISA Instruction Set Architecture 66

KCG Qualified Code Generator 90, 91

LET Logical Execution Time 15
LRU Least Recently Used 20, 24
LSU Load/Store Unit 24, 69
LUF Largest Utilization First 33

MAU Multiply-Accumulate Unit 23
MCG Multi-Core Code Generator 13, 33, 40, 55, 58, 64, 90, 91, 93, 94, 102, 115, 125
MIA Multi-Core Interference Analysis 37, 39, 40, 47, 53, 56, 65, 71, 78, 81, 82, 90, 95, 100, 102, 105, 110, 112, 117, 118, 121, 124, 125
MMU Memory Management Unit 76
MPPA Multi-Purpose Processor Array 66, 75
MRTA Multi-Core Response Time Analysis 39, 71, 84
MRU Most Recently Used 20

NoC Network-On-Chip 22, 23, 25–27, 32, 53, 56, 69, 76, 90, 98, 126

OS Operating System 14

PCI Peripheral Component Interconnect 23, 69
PE Processing Engine 23, 25, 28, 29, 65, 94–96, 99
PFB Prefetch Buffer 24, 66, 72
PLRU Pseudo Least Recently Used 20
PREM PRedictable Execution Model 14, 33, 41, 43, 44

QoS Quality of Service 69

RDMA Remote Direct Memory Access 26
REW Read Execute Write 14, 15, 33, 43, 94
RM Resource Manager 25, 28, 29, 32, 96, 99
RR Round-Robin 66, 67, 73, 75, 77, 78, 90, 118, 119

SAP Smart Arbitration Policy 66, 77
SHIM Software-Hardware Interface for Multi-Many-Core 81, 97–100, 102, 103, 125
SIC Simplified Instructional Computer 23
SL Static Level 59
SMEM Shared MEMory 25–27, 39, 65, 66, 68–70, 72, 76, 113
SMT Satisfiability Modulo Theories 33
SoC System On Chip xviii, 1, 17, 19–21, 23, 26, 56, 63, 99, 100, 102, 123, 124
SPM Scratchpad Memory 68

TCM Tightly Coupled Memory 21, 22, 25, 26
TDMA Time Division Multiple Access 21, 22, 32
TLB Translation Lookaside Buffer 76

UML Unified Modeling Language 97

VLIW Very Long Instruction Word 23

WCA Worst-Case Number of Accesses 30, 39, 56, 109, 115, 116, 119
WCET Worst-Case Execution Time 1, 9, 10, 13, 18–20, 22, 30, 31, 33, 39, 56, 59, 64, 71, 75, 77, 82, 83, 86, 90, 109, 112, 115–117, 119
WCRT Worst-Case Response Time 15, 19, 22, 27, 45, 54, 56, 57, 77, 78, 81–85, 87, 110, 112, 113, 121
WCTT Worst-Case Traversal Time 22, 27, 32, 76, 77
WF Worst-Fit 32

XML Extensible Markup Language xx, 97
XSD Extensible Markup Language (XML) Schema 97

1 INTRODUCTION

1.1 CONTEXT AND MOTIVATION

Our lives are becoming increasingly digital and we are surrounded more and more by computational systems. They are ubiquitous in our houses, transportation and are the main productivity tool for knowledge workers [47].

These systems are designed to be functionally correct, but some of them also require respecting certain timing constraints. They are called *real-time systems*. Completing the task at hand within a given deadline is part of the specification and failing to accomplish this can lead to serious consequences. Some examples of such systems are the central command of an airplane, electronic control units inside a car and a power plant monitoring room.

They can be classified into *soft*, *firm* or *hard* real-time systems, according to the impact that a functional failure or timing deviation may provoke in terms of human lives or financial losses. If missing a deadline incurs into serious consequences, the system is categorized as hard real-time or *safety-critical*. This thesis is inserted into this category. If the outputs should be discarded once the deadline has expired and the system can possibly continue to work, the system is said to be firm. If meeting the deadline is only a desirable feature but missing it does not incur into serious problems, the system is considered to be soft real-time.

The given examples (airplanes, cars, power plants) belong to the *Safety-Critical Cyber-Physical* domain, as the computing systems act as controllers to actuators and sensors, processing the readings and then outputting values through algorithms. These algorithms are coded, then compiled and executed on processors. Historically the architectures used had a single core and were extremely simple, as we have had such systems since the invention of the first microprocessors [44]. However, nowadays with complex embedded systems such as self-driving cars, takeoff and landing helpers for planes, the processing requirements have grown immensely.

At the same time, multi/many-core architectures have appeared. On top of having multiples cores that can run programs concurrently, these System On Chips (SoCs) may contain several cache levels, local and global shared memories, buses or interconnections for communication. Moreover, the cores themselves have dynamic components in their pipeline such as branch prediction or instruction reordering. All these features are extremely useful to boost the average performance, but raise huge problems for hard real-time systems as they introduce timing unpredictability.

The predictability of these systems is directly connected to being able to compute the *Worst-Case Execution Time (WCET)* of an application on a given architecture. Additionally, on multi/many-core architectures, multiple cores can access shared hardware resources at the same time, interfering with each other and mutually slowing them down. To guarantee the respect of timing constraints of a real-time system, the *interference* sources must be identified and taken into account. Note that in this thesis, we consider that we are in a context of timing compositionality (no timing anomaly, see Chapter 3).

General-purpose processors are designed to be optimized for the average use. Thus, they contain several characteristics previously presented that may introduce unexpected

delays for programs at runtime. Using these processors in hard real-time systems is an active research topic and various solutions have been proposed. One of them is by defining strict *software execution models* to construct the system. They are a broad term in computing, but here we are interested in models that express guidelines for constructing applications where the memory accesses are decoupled from the overall computation part, isolating the main source of interference from the rest of the program.

Transitioning from one core to hundreds of them allows a huge degree of parallelism by spreading the computation among them. Safety-critical systems are mainly design to provide time consistency instead of raw performance. However a clever *mapping* of an application that is able to efficiently use the available cores in a platform can greatly improve the overall execution time. Within a single core, a good *scheduling* of the tasks that must be executed into it may also provide a substantial time improvement, as finishing early one of these tasks may unblock another on a distinct core.

An engineer designing a real-time system will be mainly focused on the functional aspect of the program. The specification about the timing requirements is important but only at a later stage when the target processor is also defined. Synchronous languages have been invented to efficiently program these systems [26] as they abstract away the timing aspect, providing an idea of instantaneous reaction from input to output. These programs can also be seen as Directed Acyclic Graphs (DAGs) where each node is a computational task, with its dependencies and communication. The *parallel code generation* from these languages to multi/many-core platforms is a challenge as when this is done the implementation must respect the specification as well as be adapted and properly exploiting the features from the target architecture.

The contributions of this thesis revolve around the last introduced topics and specifically targeting safety-critical cyber-physical systems: interference analysis, software execution models, DAG mapping and scheduling combined with parallel code generation. Indeed, the main hypothesis that we aim to validate with this work is that starting with a code generated from a synchronous language we are able to provide a safe implementation (in terms of timing) in a target multi/many-core platform. To achieve this we will explore and improve the state-of-the-art in terms of execution models, DAG mapping and scheduling and response time analysis.

1.2 SUMMARY OF CONTRIBUTIONS

The basis for this work came from two previous theses: the first focused on the timing analysis [116] and the second one on the code generation and implementation [64]. Both of them targeted the industrial many-core processor Kalray MPPA2 (Bostan). Some of the core ideas in these works were used here, as our target platform is the new generation of the same family of processors i. e. Kalray MPPA3 (Coolidge). This thesis also aims to expand and explore topics that were assumed to be given as inputs previously.

The first contribution is indeed in the timing analysis of the MPPA3 processor, with some parts published in [53]. It includes:

- **A Model of a Many-Core complete arbitration path:** a mathematical model was developed with multiple arbiters for shared resources, mainly with Round-Robin and Fixed-Priority policies. The complete path from a core in a cluster until another core situated on a distinct cluster was explored;

- **A SHIM Model:** the description of the MPPA3 processor in the Software-Hardware Interface For Multi-Many-Core (SHIM), an IEEE standard intended to help programmers to efficiently use and program any architecture;
- **The improvement of a timing analysis tool:** a complete algorithm was rewritten from an existing tool, reducing the original complexity from $O(n^4)$ to $O(n^2)$ and therefore allowing it to scale to thousands of tasks.

The second contribution is a study on different software execution models, published in [127], featuring:

- **Multiple models:** 2-Phased, 3-Phased and Memory-Centric 3-Phased models were studied and compared with diverse applications;
- **Isolation versus Interference:** analysis on the impact that running memory phases in complete isolation may introduce on the system against estimating the interference that they will generate when running simultaneously.

The third contribution is in the field of DAG mapping and scheduling on a many-core processor and was submitted for publication. Different algorithms were compared to evaluate them and the consequence on the global response time. A new algorithm has also been developed to take into account the organization of the MPPA3 into multiple clusters.

The fourth contribution is a parallel code generation framework that profits from all the previous contributions in order to generate code from synchronous data-flow languages, map and schedule the tasks, provide differente execution models and finally estimate the interference in the target platform. This framework is a deliverable of the european collaboration ES3CAP [140] program.

1.3 THESIS OUTLINE

This document is organized in three parts.

Part I introduces the state-of-the-art of required topics to understand the rest of the document:

- Chapter 2 gives a background on real-time systems and synchronous data-flow languages;
- An overview on multi/many-core architectures, on the execution models for these platforms and on the applicable mapping/scheduling techniques is given in Chapter 3.

The main content of the thesis is presented in Part II:

- A high-level overview of the framework developed is in Chapter 4;
- Chapter 5 presents the contributions in the study of software execution models for critical applications;
- Chapter 6 contains our DAG mapping and scheduling algorithm;
- Chapter 7 provides the mathematical timing model of the MPPA3 architecture.

The implementation and evaluation aspects are shown in Part III:

- Chapter 8 describes the improvements done on existing tools for interference analysis, parallel code generation and multi/many-core modeling;
- Chapter 9 expands on the brief experiments from Part 2, showing the impact of the work developed there.

The conclusions and possible future work are contained in Chapter 10.

Part I

STATE-OF-THE-ART

2

BACKGROUND: REAL-TIME

This background chapter introduces base knowledge from the real-time systems domain and the particularities of this kinds of systems. We focus more specifically on the design and implementation aspects of this discipline.

Section 2.1 begins with a general introduction on what characterizes a real-time system and the execution paradigms that distinguish them from general computational systems that are tuned towards raw performance and executing as fast as possible.

When developing critical systems, the choice of languages reflects the focus on functionality in order to deliver correct outputs. But as we also target real-time systems, the implementation timing is another key requirement during the development process. Section 2.2 investigates some of the programming languages used to reach these goals, pointing out their origin, current development state and future directions.

Historically, real-time systems were designed to run on single-cores. The industry shift to multiple cores in a processor pushed the community to create mechanisms to generate parallel code from synchronous languages, as can be seen in Section 2.3.

Finally, Section 2.4 focuses on the implementation of these systems through different execution models that have a direct impact on the overall runtime and predictability.

2.1 REAL-TIME SYSTEMS

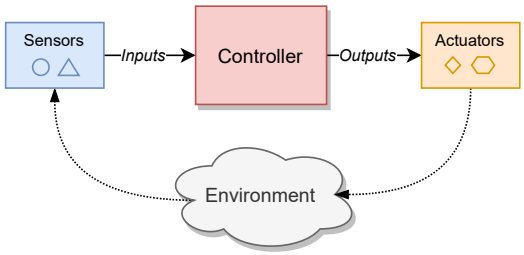
This sections enlarges some common concepts from the field of Real-Time systems that distinct them from general computing systems. We also cover the requirements and

certification, which are important parts of the design process to ensure the respect of timing constraints.

2.1.1 *Reactive and Time Triggered Systems*

Programming real-time systems means dealing with the extra-functional requirements of time at some point in the development process. This is strongly dependent on the application model at hand and may be solved by two different approaches: reaction to program inputs or time triggered static scheduling.

Reactive systems [74] are constantly waiting on an input from the environment, to then perform their programmed computation steps and finally output actions to an actuator or even another system. This is depicted graphically in Figure 2.1a and with a pseudo-algorithm in Figure 2.1b. The input to output delay is the reaction time that should respect the specification deadline. *Time Triggered* systems first arrange the execution of all the tasks composing a system and make a static deployment of them at precise times, regardless of any external interaction. These release dates are calculated to respect the specification deadline.



(a) Reactive System General Diagram

```
1 while True do
2     Inputs = ReadFromSensors();
3     Outputs = Compute(Inputs,
        Memory);
4     WriteToActuators(Outputs);
```

(b) Reactive System Pseudo-Code

Figure 2.1: Reactive system diagram and code

The synchronous paradigm that gave origin to several programming languages is well suited for both of these approaches. It helps a system engineer with a good abstraction level and leverages a programming model that is ideal to conceive and visualize the inner working of such complex systems.

2.1.2 *Requirements*

As any system, during the design phase real-time systems have a document listing the specification of the project, and in particular functional and extra-functional requirements.

2.1.2.1 *Functional Requirements*

For any computation that runs on a processor, the first vital aspect is that it outputs the correct result given the implemented function and inputs. This correct result must be guaranteed by extensive functionality test or formal verification of the implementation through abstract interpretation [43] and model checking [38], for example. There have been numerous cases of bugs either in software or hardware that incurred in terrible accidents with the losses of human lives or elevated financial cost.

The Ariane 5 [54] satellite launcher is a famous incident where a wrong cast type from a 64-bit floating point to a 16-bit signed integer resulted in an overflow that made it explode in air 40 seconds after launch. More recently two crashes involving the Boeing 737-max [81] that were a combination of a design flaw in a flight stabilizing program and loose certification by authorities, resulted in human and financial losses pushing a complete revision on Boeing's conception process and the Federal Aviation Administration (FAA) requirements for such critical systems.

2.1.2.2 *Extra-Functional Requirements*

These requirements are the ones often considered as optional during the development process: maximum temperature of the system, power consumption and execution time. However for safety-critical embedded systems they are also extremely important, and in particular the timing constraints. This thesis explores methods to perform static timing analysis that provides safe bounds on the execution times when the system becomes operational. With an accurate model of the hardware and software contained in the system, no dynamic runtime behavior may generate violations on the deadline specified as an extra-functional requirement.

2.1.3 *Certification*

Naturally, safety-critical real-time systems imply that before being utilized and going into production, they must pass through a process of certification that guarantees the conception, development and validation according to strict standards. In the avionics industry this is the *DO-178* process, that was latest revised in 2005 to include formal verification as a pertinent method to replace traditional testing [103]. The *DO-178* standard has multiple assurance levels, according to the consequences that a failure in the system may generate. Levels A and B are used to certify hard-real time systems, where an eventual failure leads to, respectively, the loss of human lives or serious injuries.

In 2014, the CAST-32A document was first released with informational content on how to use multi-core processors in an avionics environment. Due to the complex nature of these architectures, the certification process is significantly harder and the appliance of these guidelines can be extremely difficult in Commercial Off-The-Shelf (COTS) processors [4]. This new hardware must also be thoroughly analyzed to obtain the interference delay when accessing shared resources that must be added to the Worst-Case Execution Time (WCET) for single-core.

The automotive industry follows a different standard called *ISO-26262* [87] that provides guidelines for vehicle safety in several different fields, including computational and more recently autonomous systems. The number of injuries involved in a single car accident is obviously smaller than in a single plane incident, however the accumulated injury numbers of the automotive industry surpasses the avionics, which leverages the importance of such document and certification in this field as well.

Other industries that are subject to such rigorous certification are nuclear energy power plants and medical equipment. Though they have different standards that guide their development, the principles remain the same.

2.2 SYNCHRONOUS LANGUAGES

Synchronous languages have been introduced to efficiently program reactive systems [26]. They remove the timing problem burden from the developer, assuming an instantaneous propagation of signals throughout all the computation nodes of the program. This is something intrinsically hard for general purpose languages such as `C` and is one of the reasons why they are not used directly for safety-critical applications. The instantaneous reaction is achieved with a global discrete logical clock that triggers all computations and is not affected by the speed of the system where it is running. Several other features of these languages are well suited for real-time systems and will be explored in this section.

Once we talk about the implementation of such languages on concrete systems, the abstract time hypothesis receives another meaning. The deadline is described in the system specification, more or less strict according to the domain, varying from miliseconds to seconds. The instantaneous hypothesis is transformed into the following requirement:

$$\text{WCET} < \text{Specification}_{\text{deadline}}$$

Respecting the deadline is important, but giving the correct output is equally crucial for a real-time system. Being deterministic means that for the same sequence of inputs and an initial memory state, an application will always yield the same output sequence. This is crucial for testing and validating a given program and even more when certifying such systems. Synchronous languages have well defined semantics that ensure determinism and language compilers must preserve these semantics when generating code.

The languages' semantics are behind another important concept, the model-based design, where the code can be conveniently written, tested and verified in a high level paradigm and the developer can be sure that the generated code will hold the same characteristics. The verification of such programs can be done with model checkers such as Lesar [114] and SCADE Design Verifier [86].

Cyber-physical systems usually run on resource constrained devices. When code is generated from synchronous languages they do not perform dynamic memory allocation, mitigating any memory shortage problem. Moreover the loops are statically bounded, avoiding unpredictable execution times. This makes the code suitable for any WCET analysis which is an important characteristic when the reaction time between input and output must remain within a given deadline.

2.2.1 *Data Flow, Control Flow and Graphical View*

The synchronous languages have two distinct programming styles: data flow or control flow. They are basically different ideas of how to describe an application, similar to the concept of programming paradigms.

Indeed, the control flow style is tightly related to imperative programming, where one uses statements that are executed in sequence or in parallel, changing consecutively a program's state. They can be graphically represented with states and transitions between them, triggered by data or time changes, in a similar way to concurrent Finite State Machines (FSMs). On the other hand, the data flow style is more related to declarative programming. The program is divided into nodes, and furthermore into subnodes, that are the language computation and compilation units. These nodes are interconnected through their data exchange using inputs and outputs. In this case, concurrency is implicit.

In general synchronous languages are textual, nonetheless visualizers do exist, as the automatic diagram recovery tool in the Kieler project [68]. There are also some alternatives that provide a direct graphical development environment such as SCADE [28] (data flow) or Argos [101] (control flow), that helps users with a visual editor of nodes, their functionality and the overall program. In the next subsections we will dive a little bit deeper into some of these languages.

2.2.2 *Lustre*

A well-known academical and industry adopted data flow synchronous language is Lustre [73]. One of its main goals was to construct verifiable programs with its temporal logics semantics and also facilitate the adoption by being similar to tools that were already being used for reactive system design by control engineers.

Lustre programs are composed of entities interconnected through their communication. They are functionally independent nonetheless, making them the minimal execution and compilation unit. These entities are more commonly called nodes and they contain a declarative set of equations: logic or arithmetic data manipulation or delay operations. The variables are called flows due to their value being directly tied to the point in time where they are evaluated.

In a program there is a root node, possibly composed of subnodes that form a network. The compilation process takes each node separately, compiles and then schedules them respecting the data dependencies of the network. The classical compilation process generates sequential code to be run on a single core platform, but due to the entities' isolation, communication phases can be easily extracted and the program becomes parallelizable.

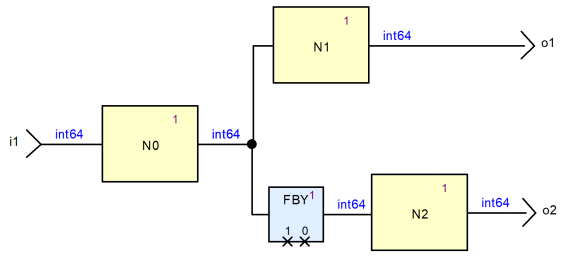
Since its appearance, it has inspired the creation and evolution of several other languages. In the next section we will introduce SCADE that was heavily inspired on Lustre's simplicity while also adding important features within an industrial point of view relevant to the system design and certification of the generated code.

2.2.3 *SCADE*

Using Lustre version 3 as base, a synchronous language to comply with avionics and nuclear plants requirements appeared. Named SCADE, it was initially a more friendly development environment for Lustre with minor feature additions and a graphical visualization. A strong differential was the creation of *KCG*, a certified code generator able to comply with avionics and automotive standards such as DO-178C and ISO 26262. This allowed system designers to be confident that the final code would be equivalent to the original SCADE model.

To incorporate more advanced functionalities while still maintaining the formalisms and semantics from its original Lustre core that allowed its use in safety-critical applications, SCADE 6 [40] was built from the ground up. The base language was *RELUC* [41] with the addition of control flow programming style, called SCADE automata, functional arrays and iterators and mixing all of this in a consistent and safe way.

In Figure 2.2 we show a SCADE sample program with the textual and graphical version side to side for comparison. In Section 8.2 we explore the challenges of generating code for multi-core architecture for parallel execution of SCADE programs.



(a) SCADE example program diagram

```
function N0(i:int64) returns (o:int64)
let
  o = i * 3;
tel

function N1(i:int64) returns (o:int64)
let
  o = i + 6;
tel

function N2(i:int64) returns (o:int64)
let
  o = i + 6;
tel

node root(i:int64) returns (o:int64)
var x, y:int64;
let
  x = N0(i);
  y = fby(x; 1; 0);
  o1 = N1(x);
  o2 = N2(y);
tel
```

(b) SCADE example program code

Figure 2.2: SCADE example program

2.2.4 *Other languages*

In this section we briefly introduce other languages that were important for the consolidation of the synchronous paradigm but took different approaches than Lustre or SCADE.

Esterel [29] is the oldest synchronous language, published originally in 1983 and conceived to provide a control flow programming style. For this reason it was widely adopted for circuit conception using state machines.

SIGNAL [27] was published in 1991 following the same data flow style as Lustre. It is relatively similar in its principles, but tackles the problem with a relational paradigm, in contrast to the equational nature of Lustre. Some operators are different, such as delay, undersampling, merge and of course, its keywords and general construction.

KIELER is a research project that offers a graphical model-based design environment [58] for its user and internally has an original synchronous language called Sequentially Constructive Charts (SCCharts) [76], derived from SyncCharts [10], itself inspired by StateCharts [79].

There is also a whole family of derived languages that followed the principles of Lustre and Esterel. Instead of creating a new language from the ground up, they extended existing languages with the reactive paradigm. Some examples are Reactive C [31] and Reactive ML [100].

More recently, industry researchers from Bosch developed Blech [67], an imperative synchronous programming language. It aims at solving some of the problems that the first synchronous languages did not set out to deal with, such as: built-in concurrent execution, easy composition of large projects through subprograms and productivity and code quality in general. It is still in early stages of development but seems to have a good future due to its open source nature, as well as real use cases from its conception.

2.3 CODE GENERATION

Generating code from synchronous programs in a safe and semantic preserving way is an intrinsically challenging task. The target platforms in the early days of synchronous languages were single cores, so the compilation to low-level code has been optimized and done in a sequential way.

Lustre and SCADE for instance, which follow the data-flow programming style, have a clear parallel visualization of the entities that compose the program. Nonetheless, decoupling the nodes to generate parallel code is not as trivial as it seems, even more when taking into account time operators or complex array structures.

2.3.1 *Sequential Code Generation*

SCADE provides an automatic generation of embedded C code through the KCG compiler. This compiler is certified at Design Assurance Level (DAL) A on the avionics DO-178 guideline and SIL 4 on the automotive IEC 61508 guideline. This means that the low-level C code that is generated can be considered correct-by-construction with regard to the SCADE source specification [28]. However the code that it generates is sequential and can only be deployed on single-core processors.

In the original Lustre paper [113] it was also demonstrated how to generate efficient sequential code from the language semantics. Even with the evolution of the language later on, it remained mainly an academic language and no certified compiler was created.

2.3.2 *Parallel Code Generation*

SCADE 6 rebuilt the language from the ground up and allowed the later development of a Multi-Core Code Generator (MCG) [39]. This code generator relies on annotations, added by the developer, to indicate that a node may be executed in parallel. With this knowledge, structures are put in place to allow these nodes to communicate, even if they are mapped to multiple cores. These structures are called *communication channels*, and their implementation is specific to each target platform. The functional code inside each node is then generated as usual, similar to the sequential code generation process.

Other works from the real-time community have proposed different solutions to generate parallel code from synchronous languages. Graillat *et. al* [66] starts from a Lustre program and a mapping onto a platform, generating code suitable to be run on this multi/many-core platform. Pagetti *et. al* [105] combines functional nodes from Lustre and uses Prelude for assembling and automatically generate a WCET-aware mapping and scheduling of programs. Yuan *et. al* [145] constructs a similar framework but directly by compiling Esterel

applications targeting multicore systems with or without a manager Operating System (OS).

2.4 EXECUTION MODELS FOR REAL-TIME SYSTEMS

Traditional software has a generic execution model that allows the mix of instructions related to the execution of the program with instructions that access the memory, either to perform a load or a store. For real-time systems, the incertitude about when a memory access will be performed leads to overly pessimistic results from a response time analysis.

In order to precisely determine when a memory access will happen, the idea to split a task into execute and memory phases appeared. This way, a mapping and scheduling algorithm could decide how to place and order each individual phase of a given task, reducing the unexpected delays that happen with a generic model.

In this section we present a short survey on research work on these phased execution models, sometimes called *predictable*, and tailored specifically for real-time systems. We split these works into two categories, according to what method they are using to improve the time-predictability. They are:

- *Memory partitioning*: the use of memory banks to place and reserve code and data that will be use by a specific core during runtime, as well as the decoupling of tasks into multiple phases;
- *Memory interference*: either allowing interference in the system and estimating it or enforcing isolation.

The term PRedictable Execution Model (PREM) comes from original works developed on mono-core systems in [102, 109, 139]. However we enlarge the use of this term to the research showed in the survey as they all target real-time systems and ways to make them more predictable, but not necessarily using the same techniques.

2.4.1 *Memory Partitioning*

Regarding the memory partitioning, related work can be classified as follows:

- **2-Phased model:** a model with *execute* and *write* phases. The execute phase has embedded read operations and for this model to work properly it is important to have some memory privatization mechanism to partition and reserve memory spaces to specific cores. This implementation model is only used in some works [66, 118] and is mostly applied to architectures that provide these bank or memory privatization features, such as the MPPA2 and MPPA3 processors.
- **3-Phased model:** a model with *read*, *execute* and *write* phases. With this phased execution model it is possible to completely run the memory phases in isolation with a careful schedule, without any need of additional hardware support for memory partitioning. This model is the most used in related work, with the terminology Acquisition Execution Restitution (AER) [98] or Read Execute Write (REW). The shared resource to read/write is not always a memory and the model may be used for I/O access [90]. The shared memory may be DRAM main memory [7, 119, 143]

or scratchpad local memory [22, 55, 105, 120, 121]. It may also take into account the DMA load/unload [132]. In some articles, the architecture is not realistic but the focus is on the bus access model [60, 94].

- **Memory-Centric 3-Phased model:** a model with *read*, *execute* and *write* phases but with a dedicated core to manage the memory phases. This model is useful for architectures and systems with an intensive use of the global memory or external resources, where a core would be blocked during these *read/write* phases. This model is studied in [23, 119, 143, 144].

2.4.2 *Memory Interference*

Concerning the memory interference, related work can be classified as follows:

- **no interference:** In [22, 119, 144], the mapping/scheduling ensures no interference by isolating memory phases. The isolation may be enforced by scheduling the task phases and may be combined to partitioning. This partitioning is used to isolate execution phases from memory access phases. The partitions may be based on time-division [33, 90, 105, 132] or round-robin software partitions. Also, the partition may be preemptive and combined with a priority promotion technique.
- **analyzed interference:** The interference delay is estimated and taken into account as part of the Worst-Case Response Time (WCRT) in [60, 94]. In some related work, the interference is taken into account as a parameter to improve the scheduling on each core or the global mapping of tasks onto cores [7, 23, 120, 144]. In [121], the memory phases are even fragmented to improve the precision of the interference analysis. The scheduling may use a software partitioning to get a preciser interference analysis (less interfering tasks) [9, 61, 143].

2.4.3 *Applicability to a broader scope*

A few works also address the challenge of producing phased code as part of the code generation/compilation step [57, 105, 132] or in the operating system [119]. A distinct approach is to provide predictable execution using different models, such as the Logical Execution Time (LET) in [108] where synchronization points are used for write phases and the read phases are always executed at the beginning of a task's period.

Scheduling algorithms for the memory phases of tasks when they run in isolation have been extensively developed. A comparison of the efficiency of such algorithms is presented in [15] but in a dynamic runtime and without precedence constraints between the tasks. Rouxel *et. al* [120] presents a forward list scheduling algorithm but their REW task model is said to be contiguous, limiting the flexibility of the schedule.

Implementation of data-flow industrial applications to multi-core platforms were presented in [55, 105]. These papers target different platforms than the Kalray MPPA2 or MPPA3, but with similar local shared memory. In both of these papers, isolation between memory access phases is implemented through hardware isolation (TDMA bus [105]) or software isolation [55].

2.5 CONCLUSION

In this background chapter we presented an overview on root concepts from the real-time systems domain. The reactive and time-triggered paradigms found in safety-critical applications motivated the creation of synchronous data-flow languages. The compilation and code generation from these language is a vital step that must preserve the semantics even when targeting multi/many-core platforms. Finally, we present the literature on execution models that can drastically improve the execution time or predictability of a real-time system.

The following chapter complements the information provided here by showing the challenges of applying these approaches and techniques to multi/many-core architectures in order to obtain time predictability.

3

BACKGROUND: MULTI/MANY-CORE

This background chapter presents the research and important concepts on the multi and many-core class of processors and their interaction with surrounding areas that are important for real-time systems.

Section 3.1 starts with a short history on single-core microprocessors and the transition to multiple cores for critical systems. This is followed by a study on past and current multi/many-core architectures that are particularly suitable for time-critical systems, highlighting what characteristics they possess that leverage computation determinism. The Kalray MPPA2 is among these architectures and was the target of some of our experiments due to its availability at the time when they were conducted. Section 3.2 is however dedicated to the main target platform of this thesis, the many-core Kalray MPPA3. It contains details about the microarchitecture and System On Chip (SoC) features that are exploited later on in the thesis.

Any processor is subject to suffering from contention when accessing shared resources, even with specialized features and configurations to reduce time variation. Section 3.3 provides clarity on how timing analysis is done to provide a safe bound on execution times, how it started on single-cores and how it evolved on modern multi-core platforms.

Finally, running applications on systems that contain multiple cores require clever placement and ordering between the tasks. Section 3.4 investigates well known algorithms on Directed Acyclic Graph (DAG) mapping and scheduling, as this is the expected

application format in our context, providing a common ground for the new developed methods on Part II.

3.1 MULTI/MANY-CORE HARDWARE ARCHITECTURES

3.1.1 *Processor architecture in real-time systems*

The first single-core microprocessors appeared in the early 1970's [12] and brought the first major breakthrough that allowed computers to get known by the general public, outside academia or industry. Since then, microprocessors have evolved tirelessly. The complexity in terms of pipeline stages, instruction set, number of cores and functionality per chip area increased exponentially turning them into much more than the calculation machines they were initially set out to be [44]. This evolution has turned the modeling and analysis of their internal structures in an active research field for diverse purposes. In the next sections we explore the impact of this evolution, specially from the safety-critical domain perspective.

3.1.1.1 *Single-Core Processors*

The first processors used in any computing system had a single core. In real-time systems, single-core processors continue to being used, even nowadays, due to their simplicity and execution predictability. These architectures were thoroughly studied, resulting in accurate models that can precisely compute the Worst-Case Execution Time (WCET) of a given application using tools such as OTAWA [17] and Heptane [78] or the industrial aiT [56] by AbsInt. The industry has been struggling to find viable and safe alternatives, as using single-core processors become more and more difficult due to their low performance and availability scarcity.

DEFINITION 1 (WORST-CASE EXECUTION TIME): *Given an application that runs on a specific architecture, its WCET is an upper bound on the execution time of this code in isolation [141]. For a single-core processor, without access to external resources and without overhead due to dynamic scheduling (preemption or event-triggered execution), this is sufficient for a response time analysis. For multi/many-core processors, the delays due to the on-chip communication and the interfence they generate must be taken into account.*

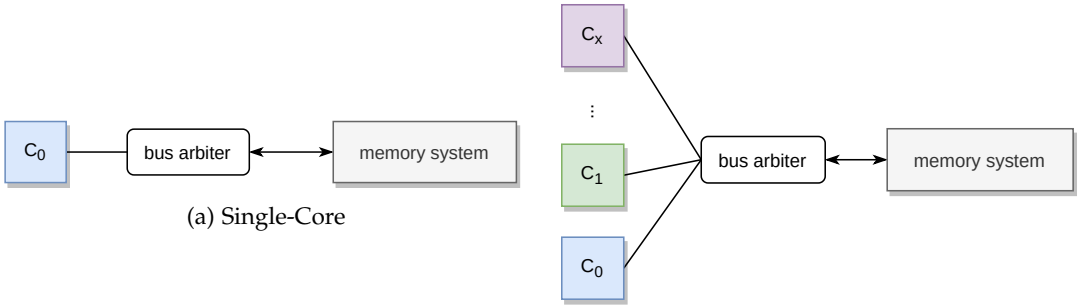
3.1.1.2 *Multi and Many-Core Processors*

Once Moore's law settled down and the increase in clock frequency was not sufficient anymore to keep improving microprocessors performance, hardware designers started increasing the number of cores and unfolding the potentials of parallelism. The new architectures that came from this were indeed more powerful than the single-core ones, as long as the software keeps up with the newly established programming models [32].

Furthermore, with concurrent execution and access to shared resources, such as the memory, the firmly established WCET methodology to ensure bounds on computation time was not sufficient anymore. In order to properly estimate a worst-case on the runtime of the whole system, the shared resources accesses (e. g. shared memory) should now be precisely modeled. Figure 3.1 illustrates the access path to the system's memory in a single-core with an optional arbiter (Figure 3.1a) and multi-core platform (Figure 3.1b).

We can see that there are multiple cores going through the same arbiter, which generates unexpected timing delays called *interference*.

DEFINITION 2 (INTERFERENCE): *An interference happens when multiple cores send a request to the same shared resource at the same time. An arbiter receives these requests one at a time and schedules them according to a certain policy. During the time that the shared resource serves the request that was first elected, the other cores are blocked waiting. The delay perceived by them will be longer than simply the time that it would have taken to access the resource in isolation.*



(a) Single-Core

(b) Multi-core

Figure 3.1: Memory system arbitration path

The advancements in the manufacturing process of integrated circuits allowed the appearance of SoCs. They combine into a single chip the functionality that was found in traditional computers with separate motherboard, memory and graphical processor. A SoC therefore contains a microcontroller or microprocessor (nowadays with multiple cores), a high speed memory, coprocessors or graphical accelerators and other auxiliary peripherals for input/output.

3.1.2 *Challenges on transitioning to multiple cores*

With the rise of multi and many-core architectures, a slow transition and change in the real-time systems community were introduced. In an architecture with multiple cores, the WCET of a program was not enough anymore to provide safe bounds of runtime [45]. The Worst-Case Response Time (WCRT) [9] concept started being used to account for the complex nature of these architectures.

DEFINITION 3 (WORST-CASE RESPONSE TIME): *Given an application that runs on a specific multi/many-core architecture, its WCRT is an upper bound on the execution time of this code, taking into account the interferences generated by concurrent execution of code on other cores, more specifically their requests to shared resources [126]. It also considers delays due to task preemptions or migrations.*

When multiple initiators try to access the same shared component at the same time, the hardware must decide which one goes first. They generate *interference*, possibly slowing each other down. If we have a timing compositional [72] hardware running the program, we can add up the delays coming from different sources, on top of the WCET in isolation and obtain a bounded response time for the application.

A naive, yet effective solution would be to temporally isolate the access to the shared resources on the system, so that interference is not possible by construction. This eliminates

the need of any hardware models or even specialized hardware, allowing Commercial Off-The-Shelf (COTS) processors to be used for safety-critical systems.

Another concern that appeared with concurrent accesses is the presence of timing anomalies [59], where a local best case scenario may lead to a global worst-case scenario. These are usually edge cases, hard to identify and predict, and they automatically violate the timing compositional assumption for a given hardware [8].

In the next sections we investigate hardware components that have a huge impact on the time predictability and response time analysis of a SoC containing a multi/many-core processor.

3.1.2.1 *Core*

The core is the main processing element of the system. For general purpose applications, they tend to be optimized for the average use-case and include components that may introduce dynamic behavior to improve performance, such as out-of-order execution and branch prediction. For predictability, it is desirable to privilege in-order execution without any sort of speculation [80]. It is also a requirement for timing analysis frameworks that the cores are fully timing compositional.

3.1.2.2 *Memory System*

In modern and complex SoCs the memory system tends to be also complicated, multi-level and heterogeneous. There are different memory types with distinct purposes at each level of the platform.

CACHE The cache memory is the closest memory to the core and usually has the smallest capacity, holding a few kilobytes in its first level up to a few megabytes in the last level. Even more important than the size of the cache is its replacement policy. The WCET analysis uses information about data locality to decide upon the delay when loading or writing data to the memory. If the cache replacement policy is complex, an analysis of the whole execution history of a program may be needed to determine if data is in the cache or not. This leads to a high complexity and sometimes unfeasible WCET estimation. Measurement based techniques can also be used in these complex or random replacement policies [1].

The true Least Recently Used (LRU) replacement policy is known to offer good performance and predictability at once, even for multi-level caches [77]. The Pseudo Least Recently Used (PLRU) and the First In First Out (FIFO) replacement policies lead to extremely pessimistic WCET estimation, while the Most Recently Used (MRU) replacement policy, used in several industrial architectures, was shown to lead to similar results in terms of analysis precision as compared to the LRU policy [70].

Having a private cache for data and instruction also increases the microarchitectural predictability. The separation eliminates dependencies between data accesses and instruction prefetch. This eases the analysis and can also speed up the execution, allowing the private instruction cache to be connected to a prefetch buffer, for example.

Another important source of unpredictability in cache memories is the coherency protocol. The coherency is used in multi-core architectures to synchronize the memory state between the private caches of these multiple cores once one of them performs a write operation to a higher cache level or the main memory. It helps the development of parallel

applications by transparently handling the propagation of update commands to all the private caches.

However, these hardware coherency protocols are kept proprietary by most vendors, have multiple transient states and introduce additional delays in memory operations [2] that are impossible to estimate without proper documentation. Therefore, for real-time systems it is preferred to bypass this mechanism either by deactivating it or by placing shared data only in the shared cache levels or the main memory [19], and controlling the accesses at the application level.

TIGHTLY COUPLED MEMORY After the cache memory, in an heterogeneous system, it is typically found a Tightly Coupled Memory (TCM), a fast, low latency memory that can act as a scratchpad [18] or even another cache level. In embedded systems with hardware constraints, a TCM might be the only available memory.

An important difference between the TCM and the cache, is that the former does not contain any mechanism to buffer frequently used code or data. Therefore the memory accesses time to the TCM should always be constant. However, as the TCM is shared between multiple cores, the access may suffer from interference delay, which is directly determined by the arbitration logic [21]. The cache has a bigger access time variability according to the locality of the data: either it is fast if the data is already in cache (hit) or the hardware must perform an upper-level access (miss) and sometimes even perform eviction.

A solution to improve the predictability of the TCM access is to decouple them into memory banks that are independently arbitrated. These banks can then be privatized to a specific core or group of cores, so that the accesses to private data and code can be performed in isolation [123]. Similar techniques can be found for locking and partitioning in shared cache [134] or DRAM memory controllers [115].

DISTANT MEMORY A global distant memory is usually available in SoCs, with bigger size providing a huge pool of memory for large data and code. However, this is typically SDRAM, which does not provide constant access time due to refresh mechanisms managed by a controller. On top of this variable access time, the global memory is also a shared resource subject to arbitration in the system.

Nonetheless, research has shown that it is possible to use SDRAM in real-time systems. One solution is to customize the controller [5] in order to provide guaranteed minimum bandwidth and a bounded maximum latency in the response time. A different perspective is to manage the static schedule of tasks in the system so that the access patterns [6] to the SDRAM are predictable and analyzable by a response time analysis tool.

3.1.2.3 *Arbiters*

Typically, there are several points of arbitration in a multi or many-core system. The most relevant for this thesis are in the memory access and communication bus paths. This memory can be a scratchpad, local to a group of cores or even a global one, and the implemented arbitration mechanisms can have a huge impact on the response time of an application. For time-critical systems, arbitration algorithms that privilege a fair share between the different initiators should be used. Some commonly known examples are Time Division Multiple Access (TDMA) [117] and Round-Robin [46].

3.1.2.4 *Inter-Core communication*

Another key point impacting the performance and being a source of timing uncertainty is the communication between different cores. In a multi-core setup, the memory is usually close to the cores and the communication can happen through this memory, without the need of a complex network to provide core-to-core communication.

In a many-core chip, the global memory can be far from the cores, serving requests with a high latency. These cores are commonly subdivided into clusters (sometimes called tiles) with a local TCM memory. Because of this, inter-cluster communication mechanisms such as a Network-On-Chip (NoC) [82] and Advanced eXtensible Interface (AXI) [34] must be used to reduce latency of inter-cluster accesses. These components must be well configured and have their timing models incorporated into the total response time of the system as the Worst-Case Traversal Time (WCTT) [14].

DEFINITION 4 (WORST-CASE TRAVERSAL TIME): *Given a communication packet that must traverse a certain network or bus, its WCTT is an upper bound on the time needed for this packet to go from source to destination [128].*

3.1.3 *State-of-the-art of Predictable Architectures*

Section 3.1.2 has exposed the challenges raised by the transition to multi and many-core inside the real-time systems domain. The key components that dictate the predictability of the system have been discussed in the literature, resulting in guidelines to build architectures that are time-predictable by design [42] and, therefore, friendly for WCET and WCRT analysis [124]. With this knowledge, the community has developed several such architectures that are investigated here.

The PRET (Precision-Timed) processor [97] is a SPARC-based architecture, single-core but multi-threaded, that provides predictable timing and extensions to the instruction set in order to manage the timing control. The rationale behind this architecture is to reduce the complexity from modern processors, that aims at average-case performance, and dedicate the same amount of effort into building a processor that has temporal characteristics in pair with its functionality. A high-level language with explicit timing features is also part of the contributions.

The T-CREST project [125] delivered a multi-core (but extensible to many-core) time-predictable architecture that is optimized for the worst-case execution instead of the average. Several hardware components were developed: the PATMOS processor, core-to-core TDMA arbitrated NoC interconnect and a predictable memory controller. Software tools for efficient programming and analysis were also a contribution, with a dedicated compiler and WCET analyzer. A distinguish feature is the *method* and *stack* caches that provide more predictable buffering and replacement of data, which eases the response time analysis.

The CompSoC [75] and CoreVA-MPSoC [13] projects are architectures created mainly for mixed-criticality use, relaxing in some of the guidelines proposed for time-predictability, in favor of performance and energy consumption, for example. Nevertheless, they still maintain several key features useful for real-time systems, such as TCM memories and TDMA NoC.

MINOTAur [69] is a recent contribution taking the open-source RISC-V core, modifying it to make it provably time-predictable, following the Simplified Instructional Computer (SIC) processor approach [71]. It also explores how to allow speculative execution while still maintaining the predictability. Though this is still only the core microarchitecture, it can be the basis for a future multi/many-core RISC-V SoC.

The Kalray MPPA3 is a COTS many-core processor that succeeds the MPPA2, maintaining the main features that made it a great fit for safety-critical systems. Being the main target of the experiments carried out in this thesis, this processor is described with more details in Section 3.2.

In this thesis we consider that the target platform is timing compositional. Note that in a particular configuration, the MPPA2 may exhibit a timing anomaly [85], but we are not concerned by this configuration, thus maintaining our timing compositionality assumption. No proof of timing compositionality exists until now. As seen in Section 3.1.2, any work that supposes a penalty for a worst-case scenario, including interference, is based on the timing compositionality hypothesis.

3.2 THE KALRAY MPPA3

This section outlines the MPPA3 architecture, covering its main characteristics for general purpose use, while highlighting the key details that make it a good candidate for use in real-time applications.

The third generation of Kalray Many-Core processors [51] called MPPA3–80 or Coolidge features 80 Processing Engines (PEs) distributed into 5 Compute Clusters (CCs) running at up to 1200 MHz. The clusters are interconnected by a NoC and an AXI fabric. The external connection is provided through its Ethernet subsystem and Peripheral Component Interconnect (PCI) Express interface.

3.2.1 *Cores*

The Kalray core family has been renamed to `kvx`, with `x` being the number of the generation. Thus for Coolidge the core is called `kv3` and it implements a 32-bit and 64-bit 6-issue Very Long Instruction Word (VLIW) architecture with an 8-stage in-order pipeline.

3.2.1.1 *Predictability and Simple Hardware*

The VLIW design philosophy goes against the more traditional architectures, `x86` or `arm` for example, that usually contain complex dynamic mechanisms such as out of order execution and branch prediction.

Instruction Level Parallelism (ILP) ensures the good performance and predictability of VLIW architectures. At compile time instructions are grouped into bundles, which are executed concurrently in the cores. The compiler must try to use as much of the available execution units most of the time. The burden of complexity is taken away from the core and delegated to the compiler [129], which greatly simplifies the hardware design and analysis.

The `kv3` core accepts bundles of up to 5 instructions and deploys them to two Arithmetic and Logic Units (ALUs), a Multiply-Accumulate Unit (MAU)/Floating-Point Unit (FPU),

a Load/Store Unit (LSU), and a Branch and Control Unit (BCU). Figure 3.2 shows this organization.

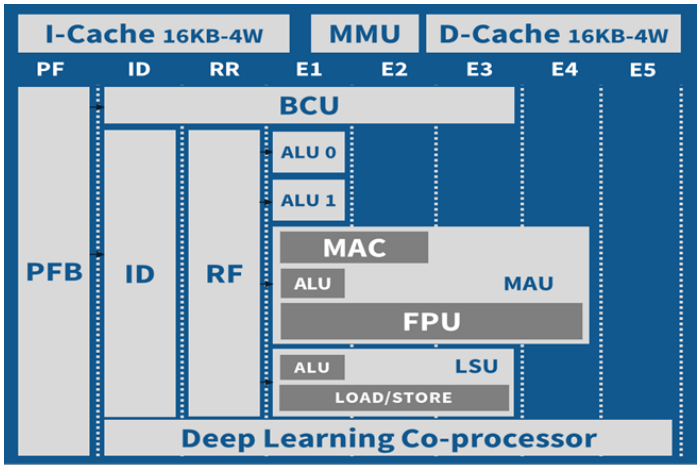


Figure 3.2: kv3 Core Pipeline – © Kalray

The kv3 core is associated with a coprocessor that operates as a tightly-copuled accelerator (TCA), mainly used for deep learning computation. It has its own specific set of instructions, datapath and registers, but can be used together with the main core to offload and speed up operations.

3.2.1.2 *Registers and other components*

There are 64 General Purpose Registers (GPRs) of 64-bits that can be associated in pairs or quadruples to ease and speed up data loading and storing. There is also a Prefetch Buffer (PFB) of 64-bytes responsible for feeding the execution units with instructions from the cache ahead of time.

3.2.1.3 *Cache*

Each kv3 core contains private caches for instructions and data. They are 4-way set associative, true LRU with 64-byte lines and 16KB of size.

The Data Cache (DC) is write-through and no write-allocate. This means that a store will update the target address when cached, but when it is not the case, it will just write to the main memory without allocating a line for the data.

At cluster level, the DC is coherent. This is ensured by a hardware cache coherency protocol. It guarantees that two cores of the same cluster will perceive the same data at a certain address after the store has been finished plus a fixed number of cycles. The `fence` instruction stalls the core until the coherency protocol has finished.

The Instruction Cache (IC) does not implement any hardware coherency protocol. For DC and IC there are special instructions that globally invalidate the cache and can be used to create a software-based coherency protocol. They are `dinval` and `iinval`, respectively.

3.2.2 *Compute Cluster*

Each CC comprises a secure and a non-secure zone, tied by a local interconnect. The secure zone contains a Resource Manager (RM) core, a 256KB secure memory bank and up to two dedicated cryptographic accelerators.

The non-secure zone contains 16 application PEs and 16 memory banks of 256KB with 32-byte words composing the 4MB of local memory. It also has Direct Memory Access (DMA) receive (R_x) and transmit (T_x) engines connected to the NoC, a dual channel AXI read and write interface, a Debug Support Unit (DSU) and may contain the cryptographic accelerators if they are not reserved to the secure zone during the boot. An overview of all these components can be seen in Figure 3.3.

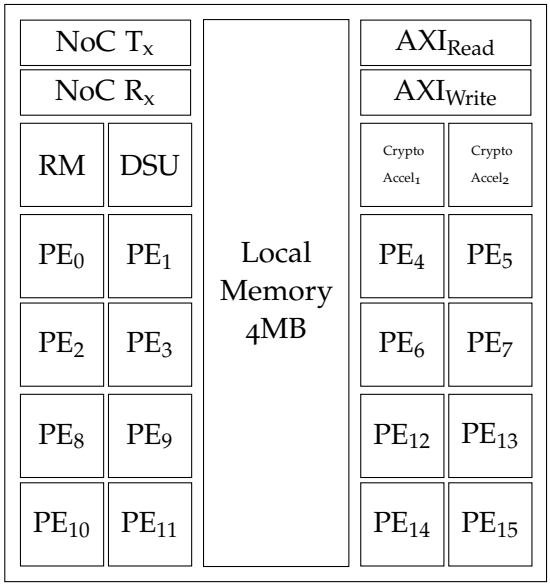


Figure 3.3: MPPA3 Cluster overview

3.2.3 *Memory System*

There are 3 levels of memory hierarchy on the architecture. The first one is the core private data and instructions caches, as described in Section 3.2.1.3. They provide transparent cached accesses to the second level but can also be bypassed through load and store instruction modifiers, or completely deactivated.

The second level is called the Shared MEMory (SMEM), a high-bandwidth and low-latency cluster-wide accessible memory. As stated in Section 3.2.2, it is composed of 16 memory banks of 256KB providing 4MB of memory for each cluster. It is able to provide up to 32-bytes of throughput per cycle per bank.

In its default *interleaved* mode, data is spread out between all the banks at a 64-byte granularity, which is great for the average use case of accesses. In its *banked* mode, contiguous addresses are all directed to the same bank. By reversing a bank to a single core and well placing code and data the amount of interference can be greatly reduced.

These 4MB of SMEM can be configured in three different ways:

1. 4MB of TCM;

2. 3MB of TCM and 1MB of L2 cache;
3. 2MB of TCM and 2MB of L2 cache.

The first configuration is the prefered one if the computation is exclusively done inside the cluster. The L2 cache can be extremely handy for bigger programs that need to interact with the global memory. There is a supplied firmware for the L2 cache protocol that uses some hardware features to speed up its software nature. This cache is 16-way set-associative, read allocate and write-back.

The third level is composed by the SMEM of other clusters and the main global memory, implemented with two DDR3/4 subsystems and shared by the whole processor.

3.2.3.1 *SMEM Access Latency*

There is a relatively long path from the core to the memory bank that stores the data. This path is filled with 2 slots FIFO buffers in order to reduce the critical path of the circuit and reach a higher core clock frequency. If the core must stall due to a dependency on a data access, in case of a L1 cache hit, the latency is 2 cycles and in case of a L1 cache miss it is 22 cycles. This is a *load use penalty*, meaning that the latency is expressed in terms of the difference in number of cycles between when the data is ready to be used and when it was requested. For instance, in the cache hit scenario, the data request is issued at `E1` of the execution pipeline (see Figure 3.2) and the data is ready to be used at `E3`. For the cache miss, the answer arrives at `E23` (20 cycles later).

Figure A.1 in Appendix A details the elements composing this path from core to SMEM. This diagram does not aim to be complete and neither translate perfectly what is synthesized in hardware, but rather gives the arbitration path from the point of view of a single core. In particular the boxes *Tree from MS to Bank* and *Tree from Bank to MSR* show only part of the tree structure from cores to memory banks and vice-versa.

3.2.3.2 *Global Memory Access Latency*

There are two SDRAM DDR controllers in the MPPA3, providing 2GB of memory in the SoC, extensible up to 32GB of external memory. These memories can operate on a frequency from 600 to 3200 Megatransfers per second. The main goal of the SDRAM is to provide high density of information in a small area, if compared to the SRAM technology used in the cluster SMEM. On top of that, the requests are sent to a controller that can reorder them (if they do not target the same address) and that must respect certain timings defined in the DDR protocol.

Therefore, the accesses to the global memory can have an important and variable latency. For real-time systems this is a huge issue. Some works [88, 112] integrate the DDR into their timing analysis process. In this thesis we consider that all code and data of applications fit into the set of clusters SMEMs

3.2.4 *On-Chip Interconnects*

CCs can communicate through two global interconnects, the Remote Direct Memory Access (RDMA) NoC and the AXI bus.

The NoC is of wormhole switching type, designed primarily to serve the two 100 Gbps Ethernet controllers located on the chip. It is more suited to carry out asynchronous RDMA

operations targeting high-performance use, even though works [52] have shown that it can be used for time-critical purposes.

The AXI fabric is the new interconnect introduced in the MPPA3, a full crossbar of buses that connects the CCs, the external DDR memory controllers and the Input/Output (I/O) controllers with direct, memory mapped access to one another. An overview of the available connections can be seen in Figure 3.4.

Once a core accesses a memory space that is comprised into the memory space of another cluster, it automatically uses the AXI. On the other hand, to use the NoC, the core must go through a DMA request that is subject to a thread request pool. Therefore, even if an AXI request is subject to arbitration between cores of the same cluster, it is more suited to time-critical systems as it does not rely on a DMA mechanism.

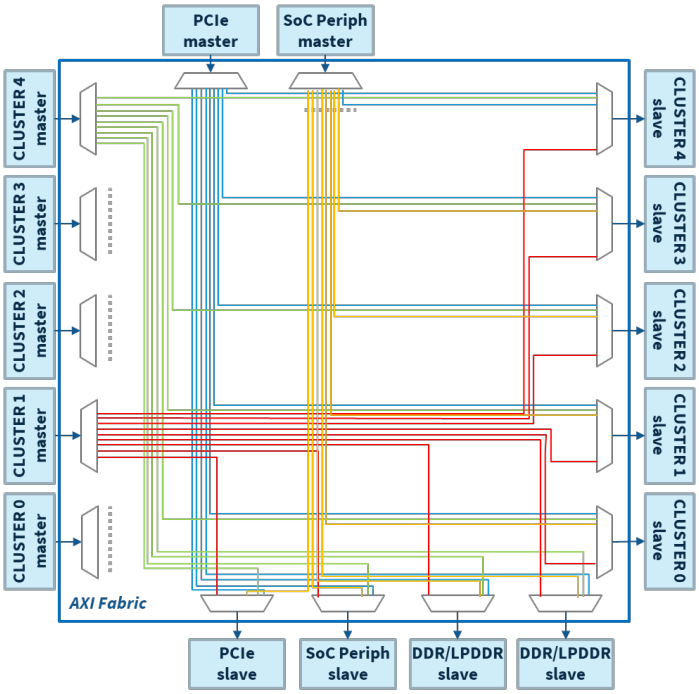


Figure 3.4: AXI Crossbar endpoints – © Kalray

On top of allowing direct access to the bus fabric between CCs, the AXI provides constant traversal times once a data packet has passed the source cluster arbiter. Thus, instead of using WCTT and network time calculus, a response time analysis framework can incorporate these traversal times with the arbiter models to provide a global WCRT, already accounting for multi-cluster communication level. This will be later explored in Section 7.4.

The AXI traversal times were measured in a MPPA3 processor with a carefully coded benchmark. Starting from the cluster where the program is running, the remaining clusters are targeted in ascending order, writing a random value at the address space of this target cluster. The target address also changes at each iteration in order to stimulate the different SMEM memory banks. This operation is repeated 1 000 times for each target before moving on to the next cluster and the maximal value is retrieved. Table 3.1 presents the results.

The results show that the traversal times are constant and symmetrical, but change according to the source and target cluster. This is due to the physical placement of the CCs

Source \ Target	Cluster 0	Cluster 1	Cluster 2	Cluster 3	Cluster 4
Cluster 0	23	108	100	92	100
Cluster 1	108	23	124	116	92
Cluster 2	100	124	23	92	116
Cluster 3	92	116	92	23	108
Cluster 4	100	92	116	108	23

Table 3.1: AXI traversal cost between clusters

in the MPPA3 chip and the inherent cost to traverse portions of the bus in the circuit with different lengths.

3.2.5 *Synchronization components*

The Kalray MPPA3 has several internal and external interrupt sources that are useful for various purposes. In this section we are interested in detailing the 4 Advanced Programmable Interrupt Controller (APIC) interrupt lines and how to use them to efficiently implement synchronization protocols between PEs of the same, or even another cluster. The APIC is a module responsible for routing and dispatching the different interrupt sources to the RM and PEs.

This module is directly connected to the mailbox system. The mailbox is a smart memory location that is configurable through different input and trigger functions. There are 128 mailboxes per cluster, each one with 8 bytes. The available trigger functions are:

- *Doorbell:* raises an interrupt anytime the mailbox is written;
- *Match:* raises an interrupt when the content of the mailbox matches exactly a configured mask;
- *Barrier:* same as the match function but clears the content of the mailbox when the interrupt is raised;
- *Threshold:* raises an interrupt when the content of the mailbox reaches or goes over a configured value.

The input functions are:

- *Write:* the value is directly written in the mailbox and the previous value is lost;
- *Or:* the value is logically ORed with the content of the mailbox ($m = pre\ v \vee v$);
- *Add:* the value is added to the content of the mailbox ($m = pre\ v + v$).

Through a proper configuration of the mailboxes trigger and input functions, and by attaching their interrupt generation mechanisms to the 4 APIC lines, it is possible to create a set of synchronization methods. Some examples are individual PE to PE

notification channels, barriers between PEs or the RM of the same cluster, or even from other clusters through the direct memory access provided by the AXI. Section 8.2.3.2 explores these components to generate initialization and communication code required for the implementation of critical software in the MPPA3.

3.2.6 *Clock components and performance measurement*

The RMs and PEs of the MPPA3 have multiple components that can be used to measure time or various events (clock cycles or cache misses for example) that happen at the microarchitecture level.

3.2.6.1 *Timers*

Each core contains two 64 bit general purpose timers. Their current values can be obtained by accessing the `$t0v` and `$t1v` registers. These registers are reset to 0x0 at boot time.

Timers can be controlled by the Timer Control Register (`$tcr`), and in particular be started or stopped. When a timer is enabled, it is decremented on each tick of the core clock, until it reaches 0. When this happens, the respective reset values of each timer (`$t0r` and `$t1r`) are loaded to the current values registers and a corresponding status bit is set.

The hardware automatically catches this status bit change corresponding to a timer underflow and, according to an interrupt enable bit (`$t0ie` and `$t1ie`), a pulse on the corresponding interrupt line is generated.

The timers are used for the time-triggered execution method detailed in Section 8.2.3.2. Each task that is scheduled in a processor has a defined release date, calculated statically prior to the execution. There are two options to obtain the desired behavior:

1. Set a start of time reference value and then actively query the current timer value until the difference corresponds to the defined release date (*polling*);
2. Set the timer value to the computed release date and put the core in a sleep state. Once the timer expires, an *interrupt* wakes up the core and the execution proceeds with the task call.

3.2.6.2 *Performance Counters*

Each core contains four 64 bit counters that allow to monitor various architectural events, optionally triggering interrupts on overflow. The values of these counters are accessed by the registers `$pm0, $pm1, $pm2, $pm3`. If the event they are configured to measure happens, they are incremented by one or more units.

The performance counters are configured through the Performance Monitor Control Register (`$pmc`). In particular, a 6 bit value is used to define the code of the event measured by each counter (`$pm0c, $pm1c, $pm2c, $pm3c`). Some relevant events that can be measured are:

- Processor Clock Cycle (PCC): +1 at every clock cycle, even during idle modes;
- ICache Miss Event (ICME): +1 for each ICache miss. Does not increment on uncached accesses;

- DCache Miss Event (DCME): +1 for each DCache miss in the L1 data cache. Does not increment on uncached accesses.

These performance counters are used in our workflow Chapter 4 to measure the WCET and Worst-Case Number of Accesses (WCA) as there is no formal model of the MPPA3 core to estimate these values. The Processor Clock Cycle event can also be used to measure time in a time-triggered execution method.

3.3 RESPONSE TIME ANALYSIS

Section 3.1.2 gave a general overview of key points of multi or many core architectures that need to be closely looked upon when using them in safety-critical systems. It also introduced the concepts of execution in isolation and interference. In this section we go deeper into these topics. A typical response time analysis method starts with a set of tasks $\Gamma = \{\tau_1, \tau_2, \cdots, \tau_n\}$, their data dependencies (that are inherent in a data-flow program). Then, either using a given mapping [118] or also performing this step [112], the method wants to find suitable release dates and response times for all tasks in Γ such as the deadline is respected and all tasks are scheduled. The WCET in isolation of the tasks is needed and also the knowledge of the hardware to be able to compute or avoid interference.

3.3.1 *Temporal Isolation*

To schedule a set of tasks avoiding interference either they have to be run in completely isolation, which breaks the parallelism and drastically reduces performance, or the tasks should be divided into execution and memory phases [92]. These memory phases are a read before execution, to fetch data, and a write after execution, to output data. The scheduling framework should then avoid any concurrent memory phases on the system, temporally isolating accesses to this shared resource, such as in [109].

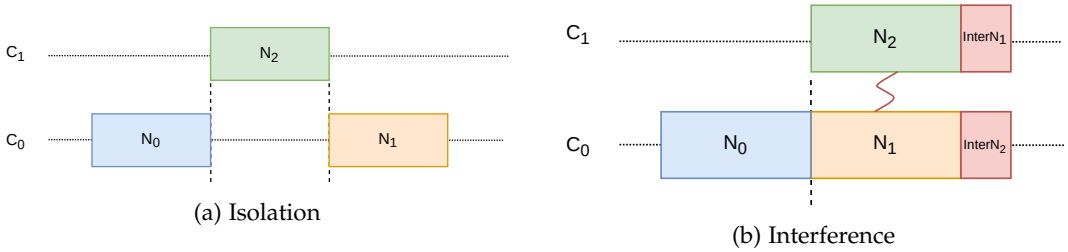
3.3.2 *Shared Resources Interference Analysis*

This approach [9, 118] tries to conciliate and bound multiple accesses to the memory. It aims to be more efficient than the isolation method and needs the number of accesses to the memory of a given task on top of its WCET in isolation. With knowledge of the hardware, it uses mathematical models to account for the worst-case in terms of interference from all running tasks on the system at that moment. The difficulty here is similar to the WCET computation problem: we want to give safe bounds but not overestimate them so we do not lose performance and are still able to schedule the task set. In terms of resource utilization this method usually yields best results, but if the models are inaccurate it can lead to pessimistic bounds or even worse, too optimistic bounds that do not correspond to the real execution.

3.3.3 *Comparison*

Figure 3.5 shows the difference in release dates and response time from imposing temporal isolation or allowing concurrent execution with interference. It reuses the example program

from Figure 2.2, removing the `fby` operator. We remind here that tasks N_1 and N_2 have a data dependency towards task N_0. The proposed mapping is tasks N_0 and N_1 to C_0 of a multi-core platform and task N_2 to C_1 of the same platform. It is also worth mentioning that the tasks are not decoupled into memory phases and we can see more clearly the huge impact in performance that temporal isolation can incur. If well computed, the interference analysis allows parallel execution while still providing safe and predictable runtime.



(a) Isolation

(b) Interference

Figure 3.5: Response Time Analysis Methods

3.4 MAPPING AND SCHEDULING

DAGs are a common abstraction that is used to represent systems or programs that need to be parallelized in a given platform. A DAG is composed of several nodes, that represent the tasks of a system, and the edges connecting them, exposing their dependencies and enforcing a certain degree of sequentialization. According to the precise methods exploring the DAG they can also be enriched with additional information: timing constraints, deadline, WCET, memory size.

The actual allocation of a DAG to a processor is a well researched topic. The allocation process involves the mapping, i.e. the assignment of a node to a core, and the scheduling, i.e. the order of execution within a certain core. Papers containing these allocation techniques for real-time systems ranges from surveys [49, 99], in depth comparisons [96] or proposing new methods [93]. We are interested here in works that follow our defined task model: a task set obtained from a DAG and the mapping and scheduling process being static and prohibiting preemption.

Our focused scope reduces the amount of related work, but we still found diverse approaches to solve the problem of mapping and scheduling an application to a many-core platform. In this section our goal is to categorize them into pure mapping techniques (assigning tasks and memory space to cores), scheduling methods (that may take contention into account) or mixed approaches combining both of them. Note that in the mixed approach we consider only work that combine both steps; most approaches rely on two separated steps (first mapping and second scheduling) that we take into account in each respective category.

3.4.1 *Mapping and Partitioning*

This section aims to present the state of the art on DAG mapping on multi or many-core platforms through papers that either focus solely on them or at least provide a clear distinction between the mapping and the scheduling phases.

Task mapping can be seen as a form of the bin-packing problem, where we have t tasks of different sizes (in execution time and memory space) to be packed into a finite number of cores with a certain memory capacity. In a *multi-core* system, a recurrent solution is to employ an As Soon As Possible (ASAP) allocation [50], while respecting the dataflow dependencies, in order to increase the system utilization. Then, the task at hand is mapped to the core that provides the earliest possible end date. For a *many-core* platform, often the goal is to minimize the number of clusters used. This is motivated by the larger communication delay between different clusters in comparison with communication through the local memory. The work presented in [35] reflects this, first partitioning the tasks into groups and then assigning each group to an island (cluster), while trying to minimize the number of occupied islands.

An extension of classical bin-packing heuristics, such as Best-Fit (BF) and Worst-Fit (WF), is proposed in [24], to incorporate the particularities of allocating real-time DAG task on NoC-based architecture using TDMA for message passing.

Also some recent publications [20, 133] grasp ideas from *genetic algorithms*, such as the Single-objective and Particle Swarm Optimization and adapt them for the real-time task mapping problem. The target platforms have a RTNoC, which is a special type of NoC tailored for real-time systems, aiming to improve routing and topology to reduce the WCTT.

3.4.2 *Scheduling*

The scheduling of real-time applications is a well researched topic for single as well as multi processor systems [16, 93, 96]. Nevertheless, there is always room for innovation, making novelties in scheduling methods and applications constantly appear in the community. For critical-systems we are mostly focused on static scheduling methods that come after a mapping step, in order to increase the predictability. In our scheduling related work research we have found that either the mapping is assumed to be given or it is the first step on a bigger framework. In either case, the scheduling is decided upon a set of tasks for each core of the system. When the task model comes from a DAG, generally a list scheduling based algorithm is employed [20, 33, 50]. .

In opposition to the static scheduling methods, when the task model allows it, dynamic scheduling can be used to increase the flexibility, at the expense of software overhead and loss of predictability. Without any DAG model, there is a higher deegree of freedom on the execution order and a classical dynamic scheduling may be used, such as Earliest Deadline First (EDF) [24, 37] and RM [35].

When the real-time system requires temporal isolation some solutions such as the Integrated Modular Avionics (IMA) concept may be employed leveraging a more complex problem. In [90] 2 techniques for scheduling applications partitions in cores are studied: (i) constraint programming and; (ii) random distribution refined with adaptive search.

3.4.3 *Mixed approaches*

The mapping and scheduling procedures in a system are intrinsically connected. A suboptimal output in any of these steps will incur in under-utilization of the platform or even deadline misses. That is why a lot of works focus on taking a combined approach and solving these problems together.

A first category of mixed solutions for this problem employs *constraint programming* to obtain an optimal solution given a set of constraints. An ILP formulation is used in [135] where the memory partitioning and data allocation are part of the problem. ILP is also seen in [22, 23] where the tasks are split into Read Execute Write (REW) phases, following the PRedictable Execution Model (PREM) approach. Targeting industrial solvers, such as IBM CPLEX, there is a complete WCET-aware code generation and multi-core deployment framework presented in [105] and the works [111, 112] that provide temporal isolation between tasks during their communication phases. Finally Satisfiability Modulo Theories (SMT) are applied in [131] with additional improvements iterative phases to compute the final solution.

The second category, on the other hand, uses global heuristic methods. One of them is the well-known list scheduling. This is the default allocation in SCADE's Multi-Core Code Generator (MCG) [39]. In [136] the list scheduling results are compared against an exact solution from SMT. A worst-fit partitioning of tasks is done in [36] followed by a Largest Utilization First (LUF) allocation. Some approaches are based on the critical path in the DAG as a priority to be scheduled [132, 146].

A rather distinct algorithm in [62] starts with a random assignment of tasks to cores with later refinement by simulated annealing, which searches for the best neighbor solution candidates. Finally several publications that use the PREM task model use some kind of code orchestration either through the schedule [22, 23], within an operating system [119] or at compiler level [132].

3.5 CONCLUSION

Real-time systems are using more and more multi/many-core architectures as their target platforms. The state-of-the-art study presented here provides the reader with knowledge about the problems that appear when resources are shared between multiple cores. A precise model of the hardware is required to confidently perform a response time analysis on the software that will be executed. A review on mapping and scheduling strategies is given, leveraging their importance for an efficient use of the platform.

Finally, the Kalray MPPA3 processor is detailed in this section, being the main target of this thesis. In particular, we analyze key components important for real-time systems as the memory hierarchy, on-chip communication and timing and synchronization components.

Part II

CONTRIBUTIONS

4

WORKFLOW OVERVIEW

The thesis until this point introduced background knowledge on several topics that are required to understand the methodology in order to generate code for real-time systems on a multi/many-core architecture. The workflow presented in this chapter receives an application, in the form of a data-flow graph, and transforms it into time-critical code for the many-core MPPA3, which is our target.

The chapter starts by presenting an overview of this workflow, supported by a quick introduction on each step that is explored with more details in the following chapters. It concludes with a comparative evolution from a similar workflow that existed with the Kalray MPPA2 as the target.

4.1 GENERAL IDEA

A graphical representation of the workflow is shown in Figure 4.1. The main contributions of this thesis appear in the red blocks that define each major step of the workflow. There is also a strong contribution on the MPPA3 arbitration model used by the Multi-Core Interference Analysis (MIA) tool.

The initial input of this methodology is provided through an external multi-core code generator that receives a data-flow application and generates: (i) the functional code from each of the application nodes, (ii) a data-flow graph with precedence constraints and (iii) the definition of memory transactions (communication) to be implemented later on. The rest of the methodology will explore these intermediate data in conjunction with the chosen partitioning and interference model, as well as mapping and scheduling algorithms to generate the additional code to run the application in the platform.

4.2 MEMORY PHASES GENERATION

The **Step 1** of this workflow consists in generating the code for the memory transactions. Based on the description of the data structures used for communication (also called channels), the communication graph and the execution model, the code for each memory transaction is produced. The kind of execution model determines the number of phases that are generated, as explained in Section 2.4. The 2-Phased model only has a write phase, while the 3-Phased and Memory-Centric models (3-Phased with a master core for memory management) have read and write phases.

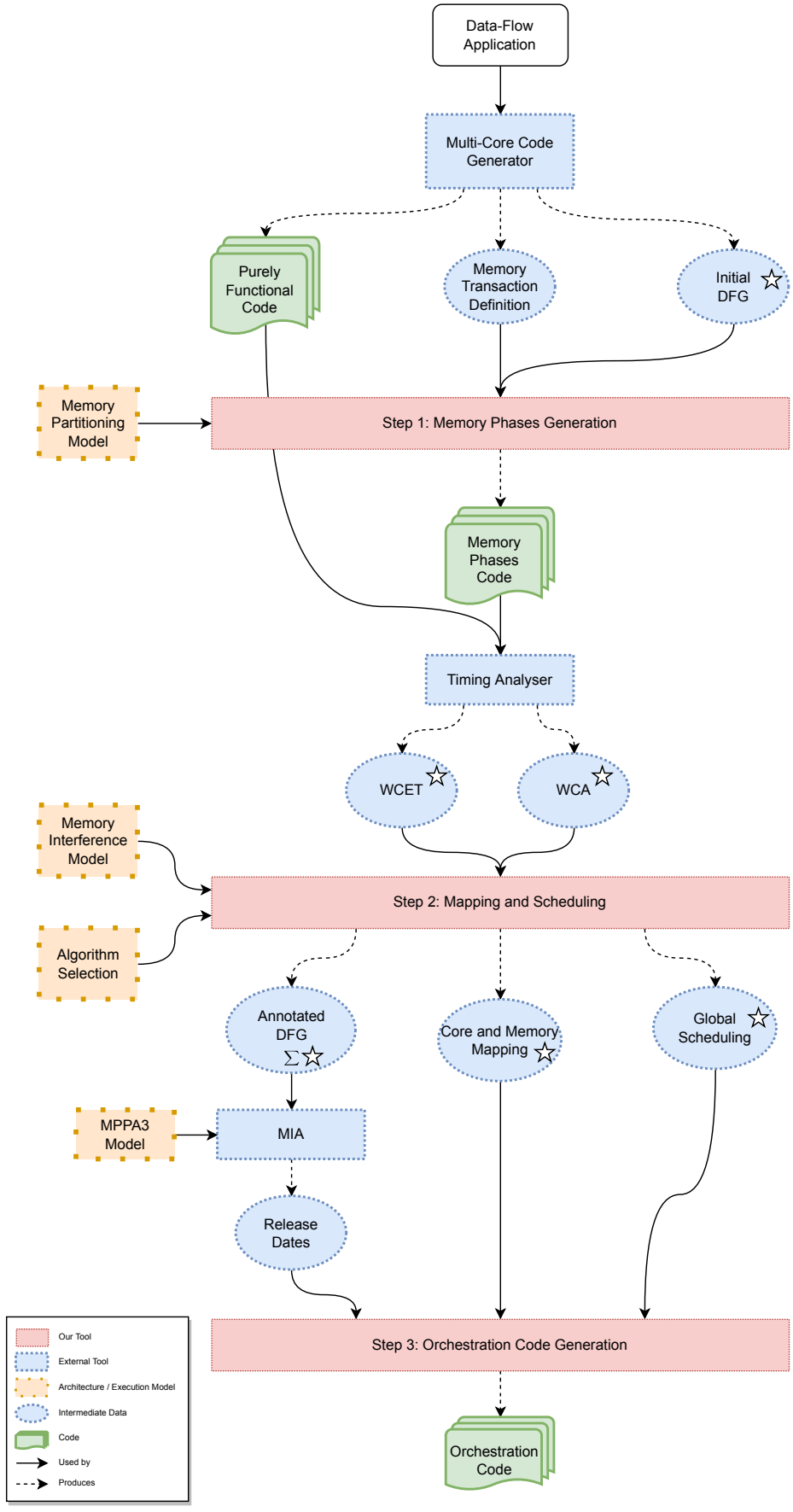


Figure 4.1: From Data-Flow to Critical-Code workflow

Chapter 5 presents in detail this step, enumerating all the studied phased-execution models and illustrating their differences through examples. It also discusses the implementation methodology taking into account some specificities from the data-flow synchronous language used (SCADE) and also the target processor (MPPA3). In particular we take advantage of its cluster banked Shared MEMory (SMEM) to reduce the interference or enforce isolation.

After this step the code for the memory phases is ready. A timing analyser is then used to obtain the Worst-Case Execution Time (WCET) and Worst-Case Number of Accesses (WCA) of the functional and memory phases code of each task contained in the application. The WCA corresponds to the worst-case number of memory accesses generated by data and instruction cache misses, which are potential sources of interference that must be taken into account later on in the process, during the response time analysis.

4.3 DAG MAPPING AND SCHEDULING

The **Step 2** of the method is responsible for taking the initial Data-Flow Graph (DFG) representation and deciding on a mapping of tasks to cores and a global scheduling between all tasks that respects the initial dependencies and communication constraints. The core mapping also intrinsically defines a memory mapping as the placement of data and code from each task is done in specific SMEM memory banks.

The code and data privatization techniques for each core, as well as shared data placement strategies are explored in the memory interference models presented in Chapter 5. This chapter also presents new scheduling algorithms to define ordering between tasks within the proposed memory partitioning and interference models.

With a higher degree of freedom, Chapter 6 contains the work done in creating and exploring algorithms for the mapping and scheduling process, without any restriction due to previously established task or memory partitioning. The goal is to improve upon classical list-scheduling algorithms in order to benefit from the multiple cores and clusters of the target platform.

The final output of this step is an annotated DFG: a file that summarizes all necessary information to be given to the MIA tool. It incorporates all components tagged with a star in Figure 4.1: the initial DFG for precedence constraints, the mapping that provides task to core awareness, the global scheduling and the timing information. Additional dependencies between memory transactions can be added to enforce isolation.

MIA is then used at this point to generate release dates that respect the information contained in this annotated DFG: the mapping, the scheduling and the dependencies. The tool also verifies if the global response time after interference and release date calculation satisfies the deadline, i. e. if the DFG is schedulable with the given timing constraints.

4.4 TIMING ANALYSIS

The response time analysis tool used in our framework is MIA, developed originally in [116], based on the Multi-Core Response Time Analysis (MRTA) framework [9]. This tool allows the user to define architecture models to compute the interference generated when several cores access the memory at the same time or traverse a communication bus.

Chapter 7 displays a study on the different arbitration levels contained in the MPPA3 processor, which culprits in mathematical models of the interference delay expected at

these convergence points of the system. The implementation of these mathematical models in MIA is detailed in Section 8.1.5.

Even though MIA is considered an external tool within this framework, an improved version is a contribution of this work. Details of the new algorithm that allows the analysis to be scaled to thousand tasks in feasible time is in Section 8.1.4.

4.5 ORCHESTRATION CODE GENERATION

In **Step 3** we generate the platform-specific code to orchestrate the application. The release dates are incorporated in this code with one orchestration function per core. Note that a cluster in the MPPA3 processor is mesochronous, thus we use a global barrier synchronization during an initialization phase to ensure that afterwards the execution flow and timings are respected.

The adaptations required in the Multi-Core Code Generator (MCG) script from SCADE Suite, particularly to produce the different code blocks from Figure 4.1, are described in Section 8.2. The section contains the process to go from a synchronous data-flow program to low-level C code and how the target platform influences the generated code, particularly in the initialization, communication and orchestration code.

4.6 COMPARISON WITH EXISTING WORKFLOW

A similar workflow existed [64] with the Kalray MPPA2 processor as a target with several differences that are highlighted here.

First of all, it is worth insisting on the fact that the existing workflow gave an original contribution on the generation of parallel code from Lustre/SCADE through an intermediate representation and parallelism extraction. This step has disappeared from our version since it has been developed in the industry as the SCADE MCG [39]. As a consequence, we directly use the result of MCG: the application functional code and an initial DFG.

Steps 1 and 2 of our workflow present contributions that were not in the existing tool. We offer the possibility of different memory partitioning and interference models, as well as several mapping and scheduling strategies. All of this was assumed to be given as input in the previous work.

Finally as the Kalray MPPA3 is our target, we implemented its new arbitration model, used in this workflow through the MIA tool from [116]. The previous workflow did not implement such model, it simply used the Kalray MPPA2 model that was already included in MIA. The Step 3, which produces the system and communication code for each specific target, is a common implementation in both workflows.

5

EXECUTION MODELS FOR REAL-TIME SYSTEMS

Software written with performance and general purpose processors in mind presents several problems when applied to the safety-critical real-time domain. Some of these inconveniences are for example, memory accesses spread throughout the whole makespan of the program, variable execution times due to synchronization mechanisms and disregard about the delay that shared resource accesses can take.

However this kind software can be adapted or rewritten following some specific execution models that improve significantly the predictability of the system. One of the most famous is the PRedictable Execution Model (PREM), which proposes the decoupling of memory access phases from the rest of the program and their execution in isolation.

We expand on the idea of the original PREM and performed a study on different software execution models that allow one to use Commercial Off-The-Shelf (COTS) processors in hard real-time systems. The main idea of all models is to coordinate the memory transactions so that there is no interference possible in the system or it is severely reduced. The results of the study are in Section 9.2.

This chapter starts with Section 5.1 describing in detail the problems behind regular software targeted for general purpose applications. Section 5.2 then introduces the selected execution models for our comparison study and Section 5.3 presents new scheduling algorithms developed to better place the memory and execution phases generated with this method. Section 5.4 generalizes everything that was seen here to other hardware

and software contexts. Section 5.5 concludes the chapter and provides some future work perspectives.

5.1 TRADITIONAL SOFTWARE MODELS

First we show the software execution models that are historically used in multi-purpose programs, their flaws and how execution models targeted for real-time systems addresses and solves them.

5.1.1 *Context*

As stated in previous chapters, the implementation of critical applications must satisfy timing constraints, therefore it must bound the execution time and communication delays of its tasks through interference estimation. Implementation on multi-core processors classically uses spatial and temporal isolation to ensure the absence of interference. Any interference causes a potential delay that requires bounding. Ten years ago, this was seen as a main issue for time-predictability [42]. However, recent work has shown that such interference could be taken into account without scalability issues [118].

This discovery has pushed the community into adopting response time analysis as a viable solution to understand how generic programs behave in a multi-core processor and the amount of interference delay that this change generates. Another branch of research [109] continued to investigate how one could mitigate interference and have a predictable execution without any interference. This was done through specific software models that avoid concurrent memory accesses on systems with multiple initiators. In order to do that efficiently these models must identify and eventually separate these accesses from the rest of the program.

5.1.2 *Memory access uncertainty*

Let us start with an example of a Data-Flow Graph (DFG) that will be used throughout this chapter to show how the different models work.

EXAMPLE 1 (DATA-FLOW GRAPH): *We use a simple data-flow application to illustrate the implementation of the execution models under study. Figure 5.1 shows the DFG: each square represents a task (N_i for Node i in data-flow terminology); each edge represents a communication (data transfer) and thus a precedence constraint.*

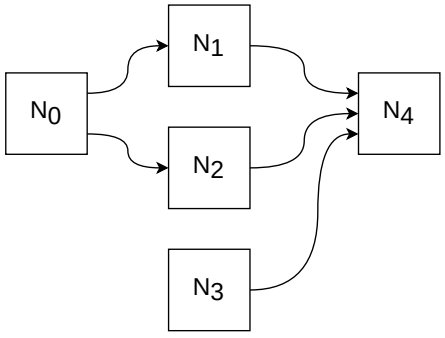


Figure 5.1: Example DFG

In a single-phased execution model, each task N_i is considered an indivisible execution unit, assuming also that no preemption is allowed in this model. Therefore the memory accesses that must be done by the nodes to communicate with each other are mixed in this single unit. When we apply a response time analysis technique or implement isolation we are faced with the uncertainty about when there are memory accesses being made and therefore the analysis must assume that they are possible at any time. Figure 5.2 shows a possible schedule from the application in Example 1 mapped to two cores.

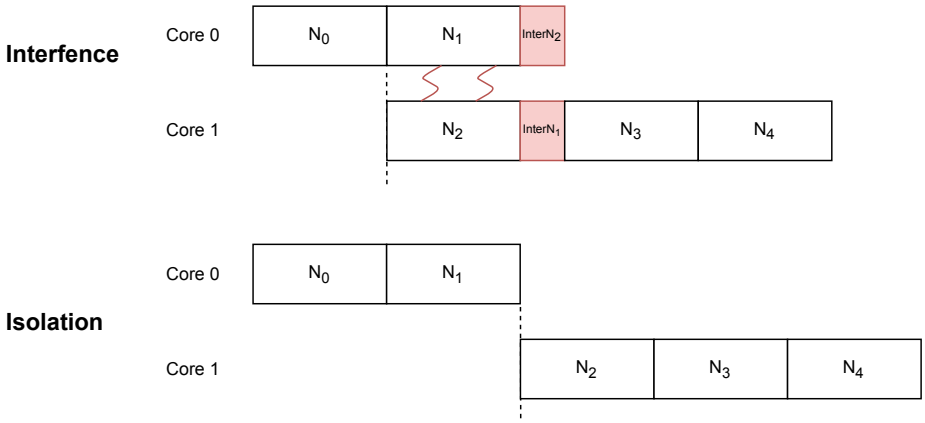


Figure 5.2: Single-Phased Model Schedule

We can see that when N_1 and N_2 run in parallel, the response analysis must assume that interference can happen throughout their whole duration. This leads to overly pessimistic results. Similarly, if we schedule the system aiming for complete isolation when accessing the memory, task N_2 can only start after the completion of task N_1, thus hugely penalizing the execution time, making it equal to a single-core schedule.

5.1.3 *Divide to better analyze*

Phased execution models appeared as a solution to the problem presented in the last section: not being able to distinguish the different parts of a task, in particular its execution and memory access phases. They also appeared from the observation that some applications already follow a phased structure with memory phases split from the main execution part. The proper introduction of phased execution models enabled time-predictable execution of applications on embedded systems that used COTS processors.

The main characteristics of phased execution models are:

- division of jobs into a sequence of non-preemptive scheduling intervals;
- time-predictable execution of some of these scheduling intervals by splitting them into shared memory access phases and local execution phases.

The original phased execution model work appeared with PREM and was motivated by the issue of distant and long-latency shared memory accesses with a lot of potential interferences. Further work has applied the PREM ideas to multi-core processors that include multiple on-chip local memories, in particular the Acquisition Execution Restitution (AER) model [55], which is a variant of 3-Phased execution models, also known as Read Execute Write (REW).

5.2 THE STUDIED EXECUTION MODELS

The following section contains details about the phased execution models that were selected for implementation in this work, with schedule examples at each step for easier understanding of their particularities.

5.2.1 *Model parameters and memory organization*

Inspiration was taken from the initial PREM works [55, 109] to investigate different software execution models by varying the following parameters: (i) a timing analysis that takes into account the delays of shared memory interference, versus an implementation with isolated phases to avoid any memory access interference; and (ii) the mapping of the inter-task communication buffers into the on-chip shared memory either distributed to memory banks assigned to the cores, or centralized into a dedicated memory bank.

The implementation of these models was tailored to multi-core processors: clusters of the Kalray MPPA. We used the MPPA2 at the beginning of the experiments due its availability but later on we ported the work to the MPPA3 when it was released. The important feature that appears in both of them is the shared multi-banked on-chip memory with a dedicated bus arbiter to access each memory bank with service guarantees. Taking advantage of these features enables higher performance while enforcing time-predictability through software-defined privatization of the local memory banks. Nevertheless the techniques seen here can also be applied to other memory systems by broadening the interference scope.

Remember that we use SCADE MCG (see Section 8.2) to generate parallel code from the DFGs of our applications. This generator performs an initial schedule between some of the phases that restricts the amount of flexibility that we can have in the final schedule. When this is the case we explain and justify the choices in text.

5.2.2 *Models overview*

In this section we present the execution models, inspired by PREMs [55, 109], that we selected for our study. We compare their implementation according to two criteria: memory partitioning and memory interference.

- Memory partitioning
  1. a 2-Phased model with execute and write phases, see Figure 5.3. The memory is partitioned such that each partition is local to a core and the tasks may access another partition only during the write phases, to send the shared data. This way any read of shared data is done in the local memory during the execute phase.

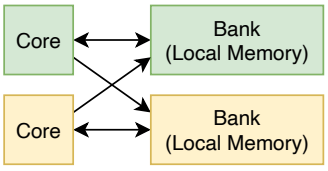


Figure 5.3: 2-Phased: Execute-Write

2. a 3-Phased model (read, execute and write phases) with a local partition for each core accessed during the execution phase and one global *shared partition* accessed by each task during read and write phases, see Figure 5.4.

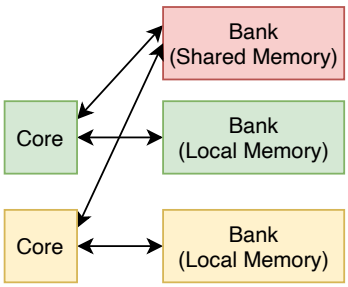


Figure 5.4: 3-Phased: Read-Execute-Write with Shared Bank

3. a Memory-Centric 3-Phased model with a local partition for each core accessed during the execute phase and a global shared partition managed by a *dedicated core* that orchestrates the read and write phases for all tasks, see Figure 5.5.

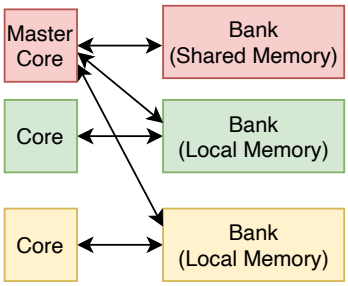


Figure 5.5: Memory-Centric 3-Phased with Master Core

- Memory interference
  - a- no interference: the mapping and scheduling ensures no interference by a software isolation between memory phases;
  - b- analyzed interference: an architecture model is used to estimate the interference delay and take it into account as part of the Worst-Case Response Time (WCRT).

5.2.3 *Schedule Analysis*

Recall Example 1 from Section 5.1.2. We are interested now in showing the differences that can occur when splitting and scheduling this example on a system with two cores using the models presented in the previous section. The schedules presented in the following figures do not aim to properly represent time. The diagram scale may change to accommodate a higher number of node phases, without necessarily meaning that a schedule in particular takes more or less time due to this representation form. The evaluations about the models and algorithms are in Section 9.2.

Figures 5.6 to 5.10 give a possible schedule for the 5 execution model implementations: in white are the execute phases (ExN_x), in green the write phases (WN_x), in yellow the read phases (RN_x), and in red the delays due to interference. In all of the phases names, N_x represents the node identifier associated with that phase.

Figures 5.6 and 5.7 represent two final schedules for the 2-Phased model where each task reads data locally and writes data to the reader memory. We observe that when interference is considered (Figure 5.6) there may be additional delay to take into account. Here, the write phase of task N_1 and N_2 interfere due to the fact that they both write in the local memory of task N_4. In the isolated implementation model (Figure 5.7) these two write phases cannot occur simultaneously, to prevent any interference. We see that WN_2 starts after WN_1 has finished.

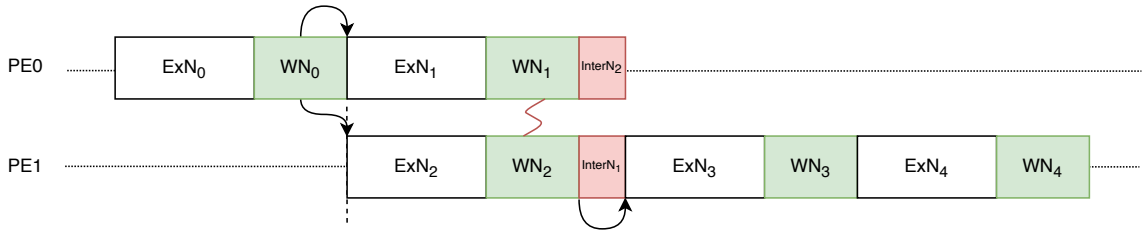


Figure 5.6: Example of scheduling for the 2-Phased model with interference cost

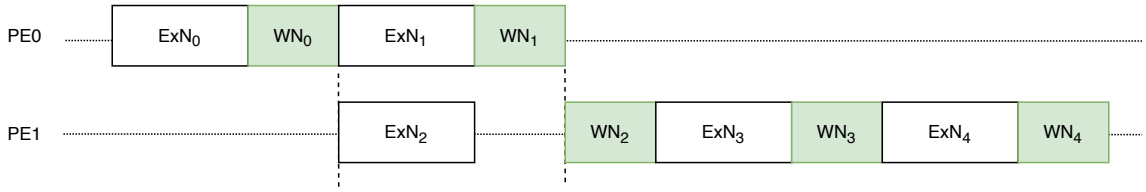


Figure 5.7: Example of scheduling for the isolated 2-Phased model

For the 3-Phased execution models (Figures 5.8 and 5.9), there are the additional read phases as each task reads from the shared memory. We observe that there is additional interference in the read phases of tasks N_1 and N_2 (Figure 5.8). We also see that a scheduling algorithm is required to achieve temporal isolation: here a priority is given to task N_1 to start its read phase RN_1 before task N_2 (Figure 5.9).

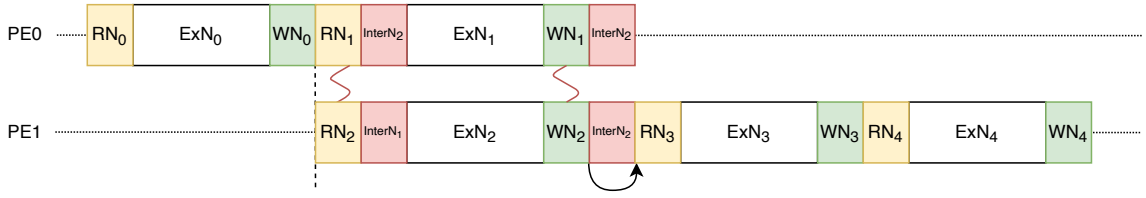


Figure 5.8: Example of scheduling for the 3-Phased model with interference cost

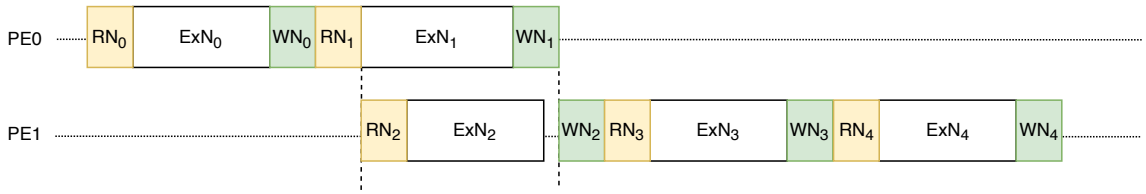


Figure 5.9: Example of scheduling for the isolated 3-Phased model

For the Memory-Centric execution model implementation (Figure 5.10), there is no possibility of interference due to the fact that each memory transaction is done by the same core. Thus, the isolated schedule given is the only one possible for this example. Here we also observe that a scheduler is required: for instance, the read phase of task N_2 (RN_2) is scheduled before the read phase of task N_1 (RN_1).

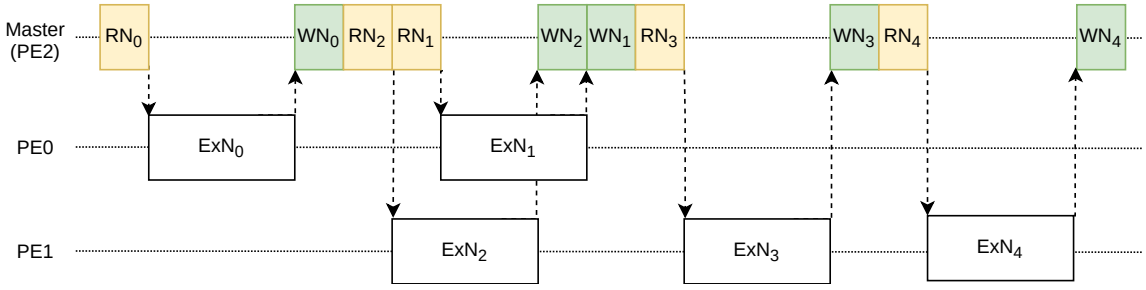


Figure 5.10: Example of scheduling for the isolated Memory-Centric model

In all schedules we observe that there is the same sequence of execute tasks on core PE_0 and on core PE_1: this is to illustrate that we work from an initial schedule given by SCADE that must be preserved. The freedom in the schedule is only between the memory phases. Note that these new schedules (isolated 3-Phased and Memory-Centric) are *global* because they must take into account the global data-dependencies and they are constrained by what is executed on other cores. For instance, in Figure 5.9 tasks RN_1 and RN_2 are executed on two distinct cores but must be executed in isolation nevertheless.

5.3 SCHEDULING ALGORITHMS

This section presents 3 global scheduling algorithms to isolate the memory transactions of the memory phases. The general goal of these algorithms is to anchor the order between these transactions across all cores. This execution order is then incorporated into the DFG in the form of additional dependencies and priority information. Later on, this enriched DFG is given to Multi-Core Interference Analysis (MIA), a response analysis tool that will take into account the graph, dependencies and communication information to compute release dates for each task and verify the global schedulability. These release dates are used for a time-triggered execution that can ensure temporal isolation, if the model requires, between any task that access the memory of the system.

5.3.1 *Background concepts*

Before introducing the general principle of the algorithms and their listings, we recall and properly define some terms that will be used onward.

In Section 5.2.2 memory transactions and phases have been introduced and it is important to clarify their difference: a *transaction* is an indivisible task that access the memory, while a *phase* can be composed of multiple distinct transactions. The algorithms presented here perform the schedule at the transaction level but must respect precedence constraints at the phase level.

The definition of a dependent memory transaction is strongly related to a typical data-flow task dependency, with some small particularities. A memory transaction *m* is

constituted of a memory operation: either read or write, respectively m_r and m_w. It also has two compute transactions associated with it, here called c_1 and c_2. Each transaction has an associated release (*rel*) and end (*end*) date.

The definition then depends on the memory partitioning model:

- For the 2-Phased model: there are only write operations, so given a memory transaction m_w from c_1 to c_2, it is dependent on c_1 being finished, which imposes that $rel_{m_w} \geq end_{c_1}$. There is no newly developed algorithm for this model as the optimal ordering is always obtained by scheduling the write phase right after the execute. By doing this, the task dependent on this write phase is unblocked as early as possible.
- For the 3-Phased and Memory-Centric models: there are read and write operations. Given a memory transaction m_w from c_1 to c_2, it is dependent on c_1 being finished, which imposes that $rel_{m_w} \geq end_{c_1}$. Given a memory transaction m_r from c_2 to c_1, it is dependent on its mirror write transaction, e.g. m_w, which imposes that $rel_{m_w} \geq end_{m_r}$.

5.3.2 *Overview and shared utilities*

All algorithms have the same input and output. The starting point is a set ℓ composed of read and write phases, ordered in the scope of a core, following the initial mapping and scheduling.

The general idea is to pick transactions from ℓ and put them in the set g, which is the globally ordered set of memory transactions across all the cores. We start with the first transaction from the first phase of the first core, which always perform a write operation, meaning that we consider data-flow applications which are either closed (they do not require external inputs) or already initialized. Then, according to the execution model and the algorithm, we either continue scheduling the other transactions from this phase or start looking for other transactions that can be scheduled because their dependencies are satisfied. The algorithms end as soon as all memory transactions are scheduled: either if ℓ and g have the same number of elements, or if we have already iterated through all memory transactions in ℓ. Remember that the set g contains the scheduled transactions and in the end it must be equivalent in size to the set ℓ.

In the algorithms we assume the existence of certain utility functions used in the listings and that have their core functionality explained here:

- `DepOk`(t) — checks if all the dependencies of a transaction t are already in g;
- `AreOnSameCore`($t1, t2$) — returns true if $t1$ and $t2$ are mapped to the same core;
- `GetWrPhase`(t) — returns the write phase associated with the execute task t;
- `GetRdPhase`(t) — returns the read phase associated with the execute task t;
- `GetMirror`(t) — returns the mirrored t transaction, i.e. for a *write* transaction between N_1 and N_0, its mirror transaction is a *read* transaction between N_0 and N_1. Note that in case of a mirror transaction the read is only subject to the precedence of the corresponding write. Thus, it is eligible for scheduling as soon as the write transaction ends.

5.3.3 *Algorithms presentation*

For the 3-Phased execution, the memory phases belonging to a task are *run on the same core* as the execute phase. This restricts the algorithm, as they must be sequentially placed in this core due to the intrinsic SCADE execution model. Any memory transactions belonging to other tasks of the same core cannot be interleaved as they would violate the initial mapping and scheduling. To explore the impact of this restriction, while also proposing a solution, we introduce two variants of an algorithm for this 3-Phased model.

Algorithm 1 shows the first variant. As in [120, 121], the 3 phases (REW) are considered as a contiguous entity, without any idle time between the memory or execute transactions. The algorithm follows the general idea from Section 5.3.2 but it always schedules the remainder of a write phase after placing a write transaction on g (Line 5). Each write transaction unblocks its mirror read, which will only be scheduled if all other read transactions of the same phase are also unblocked. Otherwise, this read transaction waits in a leftover set.

Algorithm 1: 3-Phased contiguous memory phases, referred to as *Cont* in Section 9.2

```
1  w_sched = list(); r_leftover = list();
2  foreach t in ℓ do
3      if 'write' in t and t not in g then
4          g.append(t); w_sched.append(t);
5          foreach t2 in GetWrPhase(t) do
6              g.append(t2); w_sched.append(t2);
7          foreach t2 in w_sched do
8              if DepOk(t2) then
9                  foreach t3 in GetRdPhase(t2) do
10                     g.append(t3);
11                     if t3 in r_leftover then r_leftover.remove(t3);
12                 foreach t3 in GetWrPhase(t2) do
13                     g.append(t3);
14             else r_leftover.append(t2) ;
15         w_sched.clear();
16     foreach t2 in r_leftover do
17         if DepOk(t2) and t2 not in g then
18             g.append(t2);
19             if t2 in r_leftover then r_leftover.remove(t2);
```

EXAMPLE 2: *To illustrate the difference of behavior between the 3 algorithms we will use the program in Figure 5.1. We consider that the scheduling algorithm is at the point of deciding about the schedule of the write phase of the task* N_0*. This phase is composed of two transactions:* `N0_write_N1` *and* `N0_write_N2`*. We also know that* N_0 *and* N_1 *are mapped to the same core (*PE_0*) and* N_2 *is mapped to a distinct core (*PE_1*). Algorithm 1 would globally schedule the transactions:* `N0_write_N1` $\rightarrow$ `N0_write_N2` $\rightarrow$ `N1_read_N0` $\rightarrow$ `N2_read_N0`*. Note that with Algorithm 1 the* PE_1 *remains stuck until* `N1_read_N0` *finishes, even though the data dependency has already been satisfied.*

Algorithm 2 shows the second 3-Phased variant. We flexibilize the contiguous schedule constraint by adding the possibility of introducing idle time between memory transactions of the same task and give priority to scheduling read transactions. This allows to unblock execute phases earlier than the previous algorithm and have a smaller response time. We use a double-ended queue (abbreviated here as deque) for the scheduling candidates that are popped at each iteration. Intuitively, once a transaction is scheduled, its mirror transaction may be:

- A read or write that needs to be placed contiguously with the transactions of the same phase;
- A read transaction belonging to another core that may be scheduled directly and unblocks this other core from an idle state.

Thus, according to the mirror transaction operation and mapping, it is placed at different positions in the deque to be scheduled in the next iterations.

Algorithm 2: 3-Phased with idle memory phases, referred to as *Opt* in Section 9.2

```
1  sched_cand = deque(first_task_write_transactions);
2  while ℓ.size() ≠ g.size() do
3      c ← sched_cand.popleft();
4      if c not in g then
5          if DepOk(c) then
6              g.append(c);
7              m ← GetMirror(c);
8              if 'write' in c then
9                  tr ← GetWrPhase(c) + GetRdPhase(c);
10                 idx ← −1 ;
11                 foreach c2 in sched_cand do
12                     if c2 in tr then idx ← c2.idx();
13                 if idx ≠ −1 then sched_cand.insert(idx + 1, m);
14                 else
15                     if AreOnSameCore(c,m) then sched_cand.append(m);
16                     else sched_cand.appendleft(m);
17             else if 'read' in c then
18                 foreach t in ℓ do
19                     if m in t then sched_cand.append(t);
20         else sched_cand.append(c);
```

EXAMPLE 3: *For our illustrative program in Figure 5.1 and the moment of scheduling the write phase of the task* N_0*, Algorithm 2, instead of blindly sequentially scheduling all write transactions, searches for unblocked read transactions (mirror) mapped to another core. Due to SCADE code generation restrictions the algorithm cannot break the initial ordering between the transactions and interleave them if they belong to the same core. However, it introduces idle time and give priority to scheduling read tasks of other cores. Therefore, the global schedule given here is* `N0_write_N1`

→ `N0_write_N2` → `N2_read_N0` → `N1_read_N0`. *This allows* PE_1 *to start running the execute phase of node* N_2 *earlier, which gives an overall shorter response time.*

Remember that both algorithms respect the SCADE semantics and preserve the DFG order as well as other constraints. The only optimization done in Algorithm 2 is to allow idle slots between memory and execute transactions.

For the Memory-Centric execution model, the sequential constraint of SCADE is loosened as memory transactions are mapped to a different core. Thus, there is room for a lot more flexibility when scheduling these memory transactions: we have the possibility to arrange them in any order as the execute phase will not happen on the same core. We can freely add idle intervals or not, interleave read/write transactions from other tasks and the execute phases will naturally follow the master core local scheduling due to the data-flow dependencies.

Algorithm 3 presents the method used. It follows the overview methodology but uses the mirror searching as in the Algorithm 2 to schedule read transactions as they are ready (Line 8), accelerating the parallelism deployment throughout all cores. If the dependencies for the read transactions are not satisfied they are placed in a leftover list that is revised in Line 13 before searching for the next write transaction in ℓ.

Algorithm 3: Memory-Centric isolation scheduling

```
1  sched_leftover = list();
2  foreach t in ℓ do
3      if 'write' in t and t not in g then
4          if DepOk(t) then
5              g.append(t);
6              m ← GetMirror(t);
7              if DepOk(m) then
8                  foreach t2 in GetRdPhase(m) do
9                      g.append(t2);
10                     if t2 in sched_leftover then sched_leftover.remove(t2);
11             else sched_leftover.append(m) ;
12         else sched_leftover.append(t) ;
13     foreach t in sched_leftover do
14         if DepOk(t) then
15             g.append(t);
16             if t in sched_leftover then sched_leftover.remove(t);
```

EXAMPLE 4: *Coming back at the program in Figure 5.1 and the moment of scheduling the write phase of the task* N_0*, Algorithm 3, after scheduling a write transaction, searches for unblocked read transactions (mirror) mapped initially to any core (due to the dedicated core for memory transactions, the SCADE code generation restrictions are no longer applicable). Therefore, the global schedule is* `N0_write_N1` → `N1_read_N0` → `N0_write_N2` → `N2_read_N0`*. As the memory operations are mapped to a single core, this algorithm tends to behave worse than Algorithm 2.*

5.3.4 *Termination proofs*

- Algorithm 1: terminates because its main loop iterates over the elements of the finite set ℓ, which contains the memory transactions.
- Algorithm 2: terminates when g equals the size of ℓ, the initial set of memory transactions, meaning that it has successfully defined a global schedule for all tasks. There is also an auxiliary structure c that contains schedule candidates. At each iteration we pop one candidate from c and it is either added to g or put back into c. If it is added to g one or more transactions from ℓ are then added to c as candidates. The insertion process avoids any duplication. As the number of transactions is finite, once all candidates in c have been scheduled, the termination point is reached.
- Algorithm 3: terminates due to the same reason as Algorithm 1, its main loop iterates over the finite set ℓ

5.3.5 *Complexity Analysis*

The complexity is given in terms of n which is the number of transactions to be scheduled.

- Algorithm 1: $O(n^3)$, as there are up to three nested loops (Line 2, Line 7 and Line 9)
- Algorithm 2: $O(n^2)$, as there are up to two nested loops (Line 2 and Lines 11;18)
- Algorithm 3: $O(n^2)$, as there are up to two nested loops (Line 2 and Lines 8;13)

To provide a comparison baseline we looked at the complexity of algorithms developed or referenced by [30] and [63]. Our three algorithms stay under $O(n^3)$ which is reasonable for an offline scheduling method. Moreover, similar algorithms found in these two references range between linear and cubic complexity, which reinforces for the $\mathcal{NP}$-hardness nature of the mapping/scheduling problem on multi-core architectures. In terms of scalability, [53] has showed that thousands of tasks can be scheduled in a reasonable time with an $O(n^2)$ complexity, which is the case for the majority of our algorithms.

5.4 GENERALIZATION TO DIFFERENT SOFTWARE AND HARDWARE PLATFORMS

This sections aims to give guidelines on porting the presented execution models to platforms that do not have the same characteristics as the MPPA. In particular a different memory hierarchy that requires a different analysis than the one presented until here. It also explores the adaptability to software environments outside of the SCADE programming language.

5.4.1 *Single Shared Memory*

Throughout the chapter we have shown that a multi-banked shared memory can greatly reduce the interference potential during the access to this resource. It is not a required feature to implement phased execution models though, the biggest difference being that one cannot assume that the execute phase of a task will run in isolation, as there is no memory bank privatization scheme.

Therefore, the response time analysis tool used must be able to take this into account to compute the appropriate delays. The MIA tool, which will be presented with more details in Section 8.1, has a single-bank option that in brief considers all memory accesses as possibly conflicting if they can happen at the same time.

To implement isolation, even the execute phases must run solely in the hardware. This could be improved if there is a guarantee that the data needed by the execute phase fits the private cache level available and it is preloaded.

5.4.2 *Cache privatization*

Several works [104, 134] have explored cache privatization techniques that allows a system to mimic the idea of a multi-banked memory as long as there is enough cache available to implement it correctly. This enables a platform that does not have independent arbiters to the memory bus to provide to its cores a way to run execute phases of the presented models in isolation.

By using this technique the assumptions made in the chapter about code and data privatization can be sustained and the interference scope is still greatly reduced. Nevertheless, using cache mechanisms in real-time systems must be taken cautiously, the refill policy should be analyzed so that no unexpected delays are introduced in the system.

5.4.3 *Distant DDR memory*

It is typical for COTS nowadays to only have access to a distant and long latency DDR memory that acts as a global contention point for the multiple cores in the processor. In [110] the authors explore the use of the DDR memory in the MPPA2 processor and how to manage the memory accesses through the NoC to ensure time-predictability.

For a generic system, the biggest chance is that the memory phases tend to be larger and become the main bottleneck of the system. A good scheduling algorithm is therefore even more important than in other scenarios. The literature presents the timing problems involved in using DDR memory [107], as well as task scheduling techniques that efficiently solve them [91].

5.4.4 *Multi-Cluster applicability*

The Kalray MPPA family of processors appears as a friendly target for time-critical systems, and in particular for phased execution model implementation, as the multi-banked memory of a compute cluster is large enough for the applications studied. Using more than one cluster is possible and has been exploited through the use of the NoC and network time calculus in [65]. However, inter-cluster data transfers introduce additional latency, leading to longer memory transactions. Inside a cluster, the memory transactions are identical to the ones exploited in this chapter, but each inter-cluster data transfer may enlarge the memory transaction and lead to different results.

For our target, the MPPA3, the AXI transfer should be the preferred method as it is directly accessible by the cores and leads to a more deterministic communication method than the NoC. Additional details as well as the traversal times can be found in Section 3.2.4 and Table 3.1.

5.4.5 *Software generalization*

Our work may be generalized out of the SCADE context. It may be applied to any data-flow application, as Simulink ones for example. The minimal initial information for our method is a data-flow graph, a definition of the communication data (the structure size and which task access what) and an initial mapping/scheduling. For the initial scheduling/mapping, we could use any state-of-the-art method if it is not supplied with the code. Finally, a difference with minor impact is that the generated `C` code from SCADE is not identical to the generated `C` code by other data-flow languages, which may require modifications on the generated orchestration code.

5.5 CONCLUSION

As we have seen in this chapter, predictable execution with reasonable response times and easier analysis can be obtained by incorporating phased execution models into a workflow that produces low-level `C` code from data-flow programs. Among the available models we have seen that each one may be more adapted for a specific scenario. For example the 2-phased model for architectures that allow remote write and interference-free read operations; the Memory-Centric for architectures that only contain distant memories.

Possible extensions to this work are:

- Use a large external DDR memory. With this configuration, further work is needed as the memory phases may last significantly longer than the execute phases, unlike in our study.
- Incorporate runtime adaptation of the generated time-triggered schedule, as proposed in [130]. Such mechanism can reduce the pessimism introduced by computing bound WCRT looking at the Actual Execution Time (AET) of tasks, regardless of their phases.

6

DAG MAPPING AND SCHEDULING

Many-core platforms provide a huge parallel computational power but are inherently subject to interference. The concurrent use of the available clusters and communication mechanisms leads to resource sharing and bottlenecks. These architectures provide a great opportunity for performance increase, but for real-time systems, they still require rigorous methods to ensure respect to timing constraints and provide a deterministic and predictable runtime behavior.

Performance improvement on many-core architectures is also expected for safety-critical applications, and one important step that can greatly impact the overall execution time is the mapping and scheduling of tasks onto these multiple cores and clusters. This chapter aims to investigate, analyze and propose solutions for this challenge.

We set out to solve the problem of mapping and scheduling a partially ordered group of tasks on a many-core platform. This group of tasks can be abstracted by a Directed Acyclic Graph (DAG), where each *node* or vertex represents one *task*. These terms are equivalent and used interchangeably in this chapter. The whole DAG is executed periodically on the system, repeating itself, and must respect a global deadline. The hard real-time requirements favor an offline mapping and non preemptive scheduling methodology that lacks flexibility. In contrast, it allows the framework to have a higher computational complexity, containing for example solvers or high degree polynomial algorithms.

The chapter starts by defining the system model, aiming to be as generic as possible, only supposing a many-core architecture organized into a fixed number of clusters. Then we clarify the general hypotheses assumed for the system, applications and the solutions proposed. The problem formulation follows with theoretical definitions on the DAG application structure and the overall communication cost between nodes when applied to the system model.

Algorithms to solve the formulated problem are then presented. We start with a classical list-scheduling algorithm [3], included in the SCADE Multi-Core Code Generator (MCG).

A new algorithm is then introduced, taking into account several aspects that the existing one ignores: memory use, communication cost and cluster organization. It is an initial work to provide an heuristic-based solution that can be applied to many-core platforms.

6.1 SYSTEM MODEL

The target platform is composed of M clusters $\times$ N cores, meaning that it has M clusters, each one grouping together N cores. Thus, each cluster has the same number of cores, local memory size and contains the same architectural components. Moreover, the cores within a cluster are also identical and have the same delay when accessing the local memory.

The communication between tasks can be intra-cluster, using only the local memory, or inter-cluster, using a specialized bus or some kind of Network-On-Chip (NoC). The platform is *topological* in this aspect, i. e. intra-cluster or inter-cluster communication have different delay values. This should be taken into account during the mapping and scheduling process to improve over a naive approach that ignores this difference. Furthermore, we consider a constant delay for intra-cluster communication that fits the System On Chip (SoC) topology.

The task model comes implicitly from the DAG definition. For each task, their Worst-Case Execution Time (WCET) and Worst-Case Number of Accesses (WCA) can be estimated using a tool or benchmarked on the real hardware. This information is later used and refined to compute the Worst-Case Response Time (WCRT) of each task and of the DAG, to guarantee the respect of the timing requirements of the specification. It is important to state that on top of the global deadline of the DAG, each task has a *constrained deadline* that it is strictly smaller than the task's period. In our model, this is actually defined by the release date of the next task.

We employ the Multi-Core Interference Analysis (MIA)[1] tool for WCRT computation. Our method aims to provide a good input for this tool: a fixed mapping and the order of the tasks on each core of the platform. MIA having a description of the target architecture is then capable of computing the interference cost and outputting safe and correct release dates for each task, to later on orchestrate a time-triggered execution.

A visual representation of the system model and the steps required to move from the initial problem to a final schedule can be seen in Figure 6.1. This chapter is strongly focused on the *Step 1* depicted in the figure.

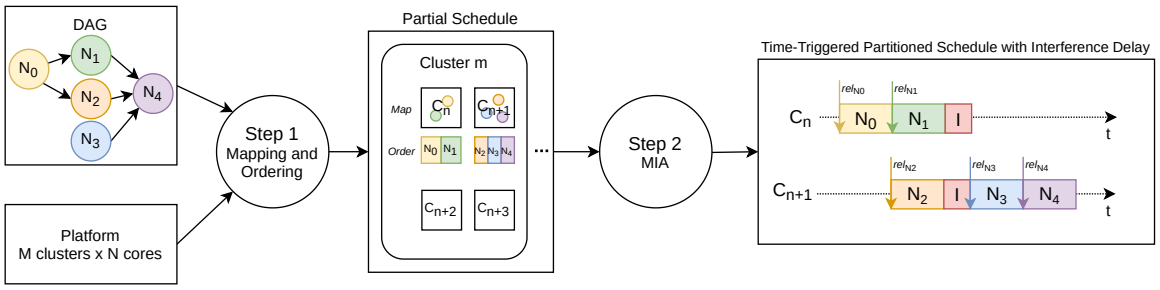


Figure 6.1: Workflow from DAG to time-triggered execution

1 Open source software under the CeCILL-C license and available at `https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/synchrone/mia`

6.2 HYPOTHESES

In order to clearly define the scope of this work we assume a certain number of hypotheses regarding the application, the platform and our method. They are:

1. *Mono-periodic* system at the DAG level, possibly expanding different task periods to their least common multiplier. A multi-periodic mode can be seen as an improvement for future work.
2. The local memory is sufficient for the data and code size required by each task belonging to the application. This should be verified and ensured during the mapping process.
3. We perform a *partitioned scheduling*, which is a static offline map and schedule that does not allow migration of tasks between cores or clusters at runtime.
4. The schedule is *non preemptive*, which means that once a task is released it executes until its completion.

6.3 PROBLEM FORMULATION

Our final goal is to map and schedule all nodes from a DAG onto an arbitrary platform composed of M clusters $\times$ N cores. We also want to do this while minimizing the WCRT of the system. In this section we will develop a set of functions and equations allowing us to give an expression of the time that an individual task will take to execute on the system.

6.3.1 *Definitions*

We begin by precisely defining our DAG named G, which is composed of a set of vertices (or nodes) V and a set of edges E:

$$G = (V, E)$$
$$E \subseteq V \times V$$

We classicaly note $succ(i) = \{j \in V / (i,j) \in E\}$ the set of successors of node i in the graph.

For any node v in the set V of G we are given the following information:

$$\forall v \in V \begin{cases} S_v & \text{memory size (instruction + data)} \\ W_v & \text{execution time in isolation} \end{cases} \tag{6.1}$$

In an analog manner, for any edge $e = (i,j)$ in the set E of G, connecting nodes i and j we have an implicit data exchange:

$$\forall (i,j) \in E \ : \ Q_{i,j} \qquad \text{size of the data exchange between } i \text{ and } j \tag{6.2}$$

The placement of any given node v on the system is given by a tuple (m,n) where m identifies the cluster index and n identifies the core local index:

$$\mathcal{P} : v \rightarrow \{ \, (m_v, n_v) \mid \max(m_v) = M \text{ and } \max(n_v) = N \, \} \tag{6.3}$$

6.3.2 *Communication cost*

Given two nodes i and j from G, the number of data blocks that are exchanged between i and j in memory units is given by $Q_{i,j}$, as introduced in Section 6.3.1.

Therefore, the time delay associated with the memory phase of a task i communicating with j is given by the number of data blocks multiplied by the time to transport each individual block plus the interference (if taken into account at this moment):

$$D_{i,j} = Q_{i,j} \times C_{i,j}^{M} + I_{i,j} \tag{6.4}$$

The interference value $I_{i,j}$ depends directly on the mapping of i and j and also on concurrent tasks being executed on the system. In this work its precise computation will be done by the MIA tool, specialized in interference analysis. Nonetheless we keep I in this equation and in Equations (6.6) and (6.7) because an approximate interference value can help in the mapping process.

The cost of each individual block C is constant and expressed according to the mapping of the source i and target j, either local or external:

$$C_{i,j}^{M} = \begin{cases} C_{\ell} & \text{if } i,j \text{ are on the same cluster} \\ C_{x} & \text{if } i,j \text{ are on different clusters} \end{cases} \tag{6.5}$$

Note that C_x depends on the topology of the platform.

6.3.3 *Total time*

Using the previously definitions and functions from Sections 6.3.1 and 6.3.2 we are able to compute the total execution time of a vertex from the DAG. For a given node i this time is:

$$\begin{aligned} T_i = & W_i + \\ & \sum_{\substack{j \mid i,j \in V \\ m_i = m_j}} Q_{i,j} \times C_{\ell} + I_{i,j} + \\ & \sum_{\substack{j \mid (i,j) \in V \\ m_i \neq m_j}} Q_{i,j} \times C_{x} + I_{i,j} \end{aligned} \tag{6.6}$$

Which, using Equation (6.4), can be simplified to:

$$T_i = W_i + \sum_{i,j \in V} D_{i,j} \tag{6.7}$$

6.4 EXISTING SOLUTION FOR DAG MAPPING AND SCHEDULING

The investigated existing solution for the mapping and scheduling of a DAG is the list scheduling integrated into the SCADE MCG script. It uses the same definitions introduced

in Section 6.3.1, but focuses only on the W_v (the execution time in isolation of a node v). The algorithm deliberately ignores S_v (the memory size of a given node) and $Q_{i,j}$ (communication cost associated with edges connecting two nodes i and j).

The list scheduling is a family of algorithms that keep a priority queue of schedulable tasks. The task with the highest priority is mapped to the core which allows the earliest start date for this task. Once this task is scheduled, its successors may become schedulable and eventually are put into the priority queue. This process is repeated until all tasks are mapped. If two tasks are mapped to the same core, they are scheduled in the same order as their allocation.

The methods for application of the list scheduling algorithm only differ in the choice of the priority function for the queue. With the Highest Level First with Estimated Time (HLFET) heuristic [3], the priority of a task is defined by its Static Level (SL), which is the length of the longest path from the task to an exit node (a node with no successors in the task graph).

6.4.1 *Static Level Computation*

The general idea of the algorithm is to start from a DAG and first compute the static level of all tasks to use as the priority for the queue. This static level is computed using the WCET in isolation as a cost function. For a task τ_i with WCET W_i, its static level is given by:

$$SL(\tau_i) = W_i + \max_{\tau_j \in succ(\tau_i)} SL(\tau_j) \tag{6.8}$$

6.4.2 *HLFET List Scheduling Algorithm*

The priority queue is initialized with the entry node of the application, always unique, and its static level as the priority value (Algorithm 4, Line 6). While the queue is not empty, the task with the highest priority is dequeued and mapped to a core. The core which allows the earliest start time is chosen (Algorithm 4, Line 10). This can be computed from the end time of dependencies of the task and the end time of the last tasks allocated to each core. And to conclude, the algorithm adds in the priority queue all task which dependencies have been scheduled (Algorithm 4, Line 19).

NOTATIONS AND VARIABLES

- ComputeStaticLevels: computes the static level of each node in the DAG according to Equation (6.8)
- *entry* is the unique entry node of the DAG
- PRIO_QUEUE is the priority queue of nodes sorted by static level
- getCoreWithStartTime: returns the core structure with the start time given as parameter
- earliestStartDate: returns the minimum time between the start time of cores, taking into account the node dependencies
- succ(x) is the set of the successors of x in the DAG

Algorithm 4: List-Scheduling algorithm

```
1  cores = [{'index': i, 'stime': 0, 'tasks': [ ]} for i in range(NumberCores)];
2  endTimes = ∅;
3  sLevels = ComputeStaticLevels(allNodes);
4  foreach i in allNodes do
5      remDeps[i] = len(i.deps)
6  enqueue(PRIO_QUEUE, (entry, -sLevels[entry]));
7  while (PRIO_QUEUE <> empty) do
8      x = dequeue(PRIO_QUEUE);
9      minDate = earliestStartDate(x, cores);
10     core = getCoreWithSartTime(cores, minDate);
11     core['tasks'] += x;
12     nodeEndTime = earliestStartDate(x, cores) + x.WCET;
13     endTimes[x] = nodeEndTime;
14     core['stime'] = nodeEndTime;
15     SuccSet = succ(x);
16     foreach i in SuccSet do
17         remDeps[i] -= 1;
18         if remDeps[i] == 0 then
19             enqueue(PRIO_QUEUE, (i, -sLevels[i]));
```

6.5 PROPOSED SOLUTION FOR DAG MAPPING AND SCHEDULING

This section presents an initial work in proposing an algorithm to solve the DAG mapping and scheduling problem, taking into account the memory use and the communication cost. Our allocation algorithm is composed of 3 phases and is based on the DAG semantics:

- 2 nodes in sequence in the DAG are never executed concurrently due to precedence constraint;
- an arrow in the DAG represents a communication: our algorithm will look for a minimization of the communication costs.

6.5.1 *Step 1: Node to virtual processor assignment*

We introduce the following notations in addition to those presented in Section 6.4.2.

NOTATIONS AND VARIABLES

- $f(core) = \sum_{j \in core} S_j$ (memory size), where $j \in V$ is a node in the DAG that is mapped to a virtual core
- *entry* is the unique entry node of the DAG
- Π_j is the set of nodes mapped to the virtual core j
- FIFO is the queue of nodes to proceed
- MAXMEM is the maximal memory size for a set of nodes mapped to a core

- assigned(x) is a Boolean, true if the node $x \in V$ is already assigned to a core, false otherwise (initially assigned(i) is false $\forall i \in V$)
- NumberCores: maximal number of cores in the platform (80 in our case)

Our first algorithm, Algorithm 5, maps the nodes to virtual cores. The main structure of the algorithm is based on the two previous DAG semantics points. First, we start with the entry node (Algorithm 5, Line 3) of the DAG and we assign in a depth-first order, as much as possible the following successors. The criteria to select the nodes is the cost of the communication: larger is the communication size, better it is to place on the same core (no additional communication cost) — Algorithm 5, Line 17. On each core, we assign nodes until the local memory is full (or the next node can not fit entirely in it — Algorithm 5, Line 14). The following nodes to restart the process described above is the next unassigned node in FIFO order, Algorithm 5, line 31 (among the followers of the start node, in the second iteration, if any). In this FIFO, are also added all successors of nodes already assigned to a core. A node without successor is assigned alone to a core, which has no influence on the global computation time, but tends to use more cores than necessary. Note that this is the main point to be refined in future work.

6.5.2 *Step 2: Virtual core to virtual cluster assignment*

This step of the algorithm takes as input the Step 1 output (node to processor assignment) and the original DAG. It takes as parameter the *NbMaxCorePerCluster*: the maximal number of processors per cluster (16 in our case). Finally the goal is to assign virtual cores to virtual clusters to prepare the last step of the algorithm.

PRE-PROCESSING. Build an undirected positive weighted graph based upon Phase 1 and the original DAG:

- ignore edges which connect the same core;
- merge (and sum the weights) of edges which connect the different cores.

We introduce the following notations in addition to those presented in Section 6.4.2 and Section 6.5.1.

NOTATIONS AND VARIABLES

- LVC = List of Virtual Cores to be mapped
- NumberClusters = number of clusters
- NbMaxCorePerCluster = number of cores in each cluster
- MaxComNeighbour(j) = one virtual core with the largest communication size with the virtual cluster $x \in LVC$
- neighbour(j) = the set of virtual cores with at least one communication with a virtual core mapped to cluster j

Algorithm 5: Step 1: Mapping algorithm

```
1  j=0;
2  Π_j = ∅;
3  enqueue(FIFO, entry);
4  while (FIFO <> empty) && (j < NumberCores) do
       /* invariant: Π_j = ∅                                          */
5      x = dequeue(FIFO);
6      if f(x)>MAXMEM then
7          Exception;
8      Π_j = {x};
9      if succ(x)==∅ then
10         j++;
11         Π_j = ∅;
12     else
13         StopProc=false;
14         while f(Π_j) < MAXMEM && !StopProc do
15             SuccSet = succ(x);
16             maxQ = 0;
17             foreach i in SuccSet do
18                 if f(Π_j)+f(i) ≤ MAXMEM && !assigned(i) then
19                     if Q_{x,i}>maxQ then
20                         SuccMax=i;
21                         maxQ=Q_{x,i};
22             if maxQ>0 then
23                 assigned(SuccMax)=true;
24                 add(Π_j, SuccMax);
25                 SuccSet -= SuccMax;
26                 if SuccMax in FIFO then
27                     FIFO -= SuccMax;
28                 x = SuccMax;
29             else
30                 StopProc = true;
31             foreach w in SuccSet do
32                 if !assigned(w) then
33                     enqueue(FIFO, w);
34         j++;
35         Π_j = ∅;
```

The second phase, Algorithm 6, based on the undirected weighted graph will compute a partitioning of the virtual cores to virtual clusters. First, we check if the current cluster is not full otherwise we "open" a new (empty) cluster and assign an arbitrary core to start (Algorithm 6, lines 5-9). Then we add a neighbour core unassigned with the maximal communication cost with the cores already assigned in the current cluster (the function MaxComNeighbour Algorithm 6, line 12). We repeat the same procedure until we have assigned all the virtual cores to a virtual cluster.

Algorithm 6: Step 2: Clustering algorithm

```
1  j=0;
2  Cl_j = ∅;
3  while LVC <> ∅ do
4      if |Cl_j| == NbMaxCorePerCluster then
5          j++;
6          Cl_j = ∅;
7      if Cl_j == ∅ then
8          Cl_j = {c ∈ LVC};
9          LVC = LVC \ c;
10     else
11         if neighbour(j) <> ∅ then
12             c=MaxComNeighbour(j);
13         else
14             c = e ∈ LVC ;
15         Cl_j+ = c;
16         LVC = LVC \ c;
```

6.5.3 *Step 3: Virtual to physical cluster assignment*

The final step consists in assigning virtual clusters and cores to real ones. For this we take into account the different communication delay between clusters for the MPPA3 platform. The traversal times can be found in Table 3.1 and we can see that they vary according to the source and target cluster due to the physical SoC placement. From the topology of the AXI communication, we explore all possibilities. As there are only 5 clusters in the Kalray MPPA3, this approach is feasible. We compute a communication cost from the multiplication of the traversal time and the amount of transfers. The physical attribution is done choosing the permutation that gives the lowest communication cost. The virtual cores can be assigned to physical cores in any order without impact on the response time.

6.6 CONCLUSION

A preliminary work proposing a new algorithm for the DAG mapping and scheduling problem on many-core platforms has been presented in this chapter. We defined our system model, problem and the hypotheses assumed before solving the allocation question.

A classical list-scheduling algorithm taken from SCADE MCG was presented, using the WCET and earliest start date as metrics for task placement. It completely ignores the memory use in the system, which is unrealistic, and also does not take into account the heterogeneous communication delays found in modern multi/many-core architectures.

Our proposed solution takes both of these metrics into account to propose an algorithm split into multiple steps. The first step assigns nodes to virtual cores, trying to group together a node and its successors, without violating the maximum memory constraint for each core. The second step places these virtual cores onto virtual clusters, looking to minimize the inter-cluster communication amount. The third and last step attributes virtual to physical clusters in our target platform, the Kalray MPPA3, exhausting the possibilities and choosing the one with less communication overhead.

This chapter presents an initial work with a single solution to challenge the traditional list-scheduling method. Future works may explore different algorithms and strategies such as a node coloring step, different heuristic for grouping nodes to virtual cores, and even exact solutions using an ILP solver. In step 1 we suppose an unlimited number of virtual cores as a starting method that needs to be refined to regroup together isolated nodes. In step 2 an improvement path is to use a clustering approach based on a balanced graph partitioning [11].

7

TIMING MODEL OF AN INDUSTRIAL MANY-CORE ARCHITECTURE

This chapter presents the timing model of the Kalray MPPA3 processor, which is the target platform of the workflow introduced in Chapter 4. The implementation of this theoretical analysis is in the Multi-Core Interference Analysis (MIA) tool as one of the available shared resource interference models. The outline of this chapter is as follows: Section 7.1 contains the arbitration details inside a cluster of the processor, while Section 7.2 explores the arbitration when accessing other clusters. In Section 7.3 we precise the valid execution model for this analysis to work and in Section 7.4 the timing model is presented. If the execution model is not respected Section 7.5 shows the changes and adaptations required. Section 7.6 contains the conclusion.

7.1 INTRA-CLUSTER ARBITRATION

We recall the information given about the memory system and hierarchy in Section 3.2.3. Now the focus is on understanding the path and components a Processing Engine (PE) must go through to reach the cluster memory. The analysis is split into two levels, one for each arbiter up until arriving at the Shared MEMory (SMEM). A diagram overview can be seen in Figure 7.1.

7.1.1 *Level 1*

Leaving the core, the first level is composed of a Fixed Priority (FP) arbiter between the Data Cache (DC) and the Instruction Cache (IC). The DC has the priority over the IC, implying a possible starvation for the IC. Write operations and special instructions in the Instruction Set Architecture (ISA) of the MPPA that bypass the cache also have preference over the IC when accessing the memory.

Thus, this starvation is limited by the number of program instructions that the core is able to process until it must stall to load more code. The hardware component responsible for this limit is the Prefetch Buffer (PFB) [137] that issues requests to the IC and then forward these instructions to the execution pipeline.

7.1.2 *Level 2*

The SMEM introduced in Section 3.2.3 is composed of multiple banks, each one with its own arbiter. The existence of multiple arbiters helps to minimize the interference between initiators, if each one is associated with a bank. This configuration is explored later on Section 7.3.1.2, but a summary is that a memory access to one bank has no impact on the access time to another bank. An initiator is defined as any element that may issue a request to an arbiter, in this case by accessing the SMEM.

The arbiter is a special Round-Robin (RR)[1]. We divide the initiators into two groups:

- Cores ($G1_c$): P_i, for $i = 0 \ldots 15$
- Others ($G1_o$): RM, DSU, CryptoAccel$_1$, CryptoAccel$_2$, NoC T$_x$, NoC R$_x$, AXI$_{\text{Write}}$, AXI$_{\text{Read}}$

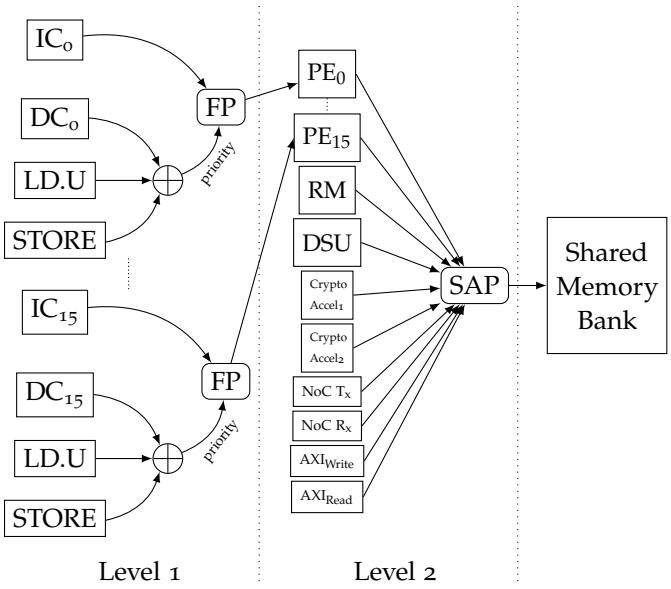


Figure 7.1: Intra-Cluster Arbitration

1 The arbiter is called Smart Arbitration Policy (SAP) and it is basically a configurable RR. The configurable value is n, providing $n + 1$ consecutive grant rounds to each entry.

7.2 INTER-CLUSTER ARBITRATION

We recall here the information given about the on-chip interconnects in Section 3.2.4 and in particular about the AXI fabric. At the endpoints of the AXI crossbar, the arbitration policy is a Deficit Round-Robin (DRR)[2] and the initiators are also divided into two groups (see Figure 3.4):

- Compute Clusters ($G2_{cc}$): CC_i for $i = 0 \dots 4$
- Others ($G2_o$): DDR_0, DDR_1, PCIe, SoC Peripheral

As the arbitration inside the cluster, the arbitration interference between them may also be seen as a problem that can be split into multiple levels (cf. Figure 7.2):

1. An intra-cluster arbitration, seen in Section 7.1, from the source cluster (Level 1)
2. A DRR arbitration at the edge of the arrival cluster (Level 2)
3. Inside the arrival cluster another intra-cluster arbitration to access the destination memory bank (Level 3)

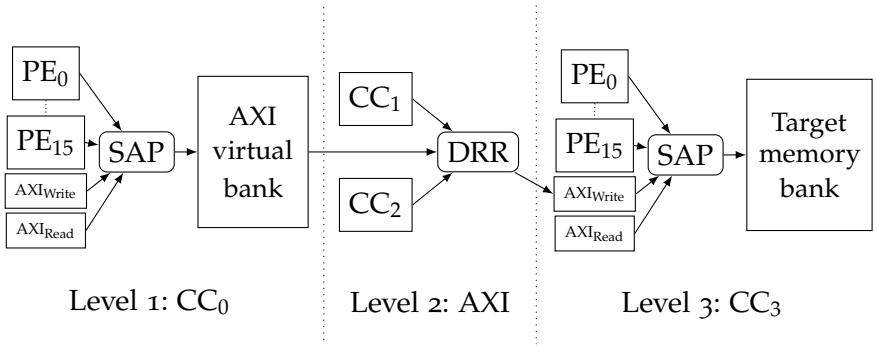


Figure 7.2: Inter-Cluster Arbitration

Note that a simplification done in our model is to treat the DRR arbiter as a regular RR due to two reasons. The first reason is that the Advanced eXtensible Interface (AXI) protocol supports bursts that may count as a single access from the arbiter point of view. The second reason is that the hardware implementation of the DRR has a problem, the arbitration is actually performed on the address and not on the data. This means that the deficit counter never reaches zero for any initiator and therefore the behavior is identical to a RR. The initiators are always able to send the totality of the data in their buses.

7.3 CONFORMANT EXECUTION MODEL

This section describes the proposed system design and processor configuration for the time-critical domain targeted in this thesis. We detail what and how specific components of the architecture are used and also leverage hypotheses about the software, communication and source code generation methods.

2 A generalization of the round-robin arbitration scheme where each initiator at a given round is allowed to send at most Q_i bytes (quantum) and the remaining, if any, is reported to the next round. At the beginning of each round the initiator refills its deficit counter by one quantum.

7.3.1 *Architecture Configuration*

For general systems, raw speed is really important and the key metric of optimization is most times the average performance of the program. Being an average, this ignores the variation that can be observed between the slowest and fastest runtimes. On the other hand, for real-time systems, raw speed is not that relevant. What matters is the predictability that the system shows, i. e. the execution time of the tasks must be as constant as possible. That is why some mechanisms that are great for general purpose performance are not used or, if possible, configured in a different manner when using Commercial Off-The-Shelf (COTS) processors [42] for safety-critical systems.

7.3.1.1 *Cache Configuration*

The hardware implemented L1 cache coherency protocol is deactivated in our configuration. This avoids unexpected performance throttles when updating data used by multiple cores. The responsibility of managing cache coherency is thus delegated to the programmer (as it was the case with the MPPA2): special instructions such as `DINVAL`, `IINVAL` and `FENCE` must be used at a convenient time to keep correct assumptions between memory values that may be present in the data/instruction cache of distinct cores.

The MPPA3 has a L2 firmware system where part of the cluster internal memory is used as a L2 cache. This is also deactivated in this configuration, as another memory level introduces more latency and it is also a source of unpredictability within the response time analysis of the system. The majority of the computation of a real-time system is performed within the same local memory space and a L2 cache for the MPPA is mainly used to provide better average response times when accessing global memory spaces such as the Double Data Rate (DDR). If sharing memory spaces between different clusters, the same data/instruction cache maintenance instructions must be used in the software to ensure coherence.

7.3.1.2 *SMEM Configuration*

The SMEM of the cluster can be configured in *banked* or *interleaved* mode. We use the SMEM of the cluster in its *banked* mode. The default configuration of the platform is the interleaved mode. For general use cases, this tends to be better as it balances the requests from all cores to different memory banks, effectively distributing the workload between multiple arbiters. This avoids possible bottlenecks when accessing contiguous blocks of memory. The interleaved mode incurs in access patterns that are difficult to predict, model and may introduce a lot of interference. The banked mode allocates contiguous memory addresses to the same bank, and linking the data and code that a core will use to this bank ensures its isolation and minimize interference.

Within the time-critical context of this study, we will assume the use of the SMEM as a Scratchpad Memory (SPM), with contiguous memory areas in the banks where the cores may work independently. We also suppose that the program's data and code fit inside a memory bank and more generally into the cluster memory and that everything is previously loaded from the high-latency and hard to predict DDR memory or directly injected into the SMEM when loading the program.

7.3.1.3 *Memory Access Model*

When computing the response time of a program, we can consider that the memory accesses issued from a core are either *blocking* or *non-blocking*, according to the architectural state when the access was made. In particular, for the MPPA we are interested in knowing if the access has properly exited the Load/Store Unit (LSU) and the core pipeline may continue to charge further instructions.

The definition of non-blocking access is intuitively simple: it is an access that may slow down other cores, but does not block the issuing core. If the system is not overcharged, a write access is non-blocking, but if the FIFOs between the core and the SMEM are full, it becomes a blocking access.

The additional wait time provoked by this FIFO saturation cannot be bigger than the scenario where all accesses are blocking. With blocking accesses, they do not leave the LSU and stall the core directly for the 22 cycles of latency.

Therefore, we have two options when modeling the memory accesses:

- Assuming that all accesses block, which leads to safe but overestimated bounds;
- Considering non-blocking accesses, which leads to tighter but still safe bound. Improving the estimation can be achieved by modeling the FIFOs in the SMEM path.

We stick with all blocking accesses as a first approach to the problem to abstract the analysis of the access return path.

7.3.1.4 *Inter-Cluster Communication*

As seen on Section 3.2.4, there are two ways to connect a cluster with the others on the platform, to the global memory (DDR), or to the external interfaces (Ethernet, Peripheral Component Interconnect (PCI)). They are the Network-On-Chip (NoC) and the AXI.

In order to send data through the NoC in the MPPA3, a thread must be created in the Direct Memory Access (DMA) controller and then arbitrated. The implemented policy may introduce unfairness and unexpected delays and should be avoided for time-critical applications. Nonetheless, the NoC is extremely well suited to handle Ethernet and PCI traffic. It also has builtin Quality of Service (QoS) parameters to ensure system responsiveness.

The AXI is a great choice for real-time systems due to its simplistic approach of directly connecting a cluster with all the others in a constant traversal time. Moreover, the memory of the system is flatted out and when reading from or writing to a particular address outside of the current cluster memory zone, the AXI is indirectly used to perform the operation. All of this avoids some NoC related problems such as route definition and packets deadlock. In conclusion, the AXI is more predictable, allows to use a simpler multi-level arbitration model and for these reasons is taken as a first approach.

7.3.2 *System Design*

This section mainly explores the system details that require attention when designing programs, writing code or linking compiled objects to a final binary. These details define recommendations to improve the predictability of the system in the platform.

7.3.2.1 *Code and Data Alignment*

Code should be always aligned on a 8-byte boundary to avoid additional delays that can be created when accessing the memory. The SMEM bus size is 256-bit and if a memory access is near this boundary and misaligned, the hardware protocol will actually split this access into two, damaging performance.

Data also should be *naturally aligned*, according to its type. Their address must be a multiple of the type size, as computed by the `sizeof` operator in C language. This ensures that dealing with data accesses will not generate the same split behavior as the code.

7.3.2.2 *Reading versus Writing*

We expect the time-critical applications to be divided into individual tasks with inputs and outputs. These tasks can also eventually communicate with each other. Their schedule is static and does not allow migration, i. e. one task will always run on a unique and identical core. This matches the kind of code that is usually generated from synchronous data-flow programs.

The aforementioned communication between the tasks can be done by either reading from or writing to the memory. The path of this operation strongly depends on the mapping of the source and target tasks. Three situations can occur:

- Source and target tasks are mapped to the *same* core: they cannot execute concurrently, and the communication will happen on the same memory bank;
- Source and target tasks are mapped to *different* cores of the *same* cluster: the communication will go through distinct memory banks of the cluster;
- Source and target tasks are on *different* clusters: the communication will go through the AXI bus.

Hereby we assume only writing operations between the tasks, i. e. a bank that belongs to another task, either on the same cluster or on another, is never read, only written. This simplifies the interference model as there is no need to compute the read time completion. Also this helps to provide a consistent code generation guideline.

7.3.3 *Software Framework*

The application code is generated from Data-Flow Synchronous languages, SCADE being the most well-known industrial language constructed upon this paradigm. Kalray provides an integration script with SCADE Suite that generates C code with the required low-level code for synchronization and time-triggered execution of the tasks. These tasks are periodic and non-preemptive, running until completion.

The reader must keep in mind that a general purpose program cannot be easily fitted into this analysis. The application should have a clear separation in tasks, with additional information about the precedence and the communication scheme between them. In other words, there must be a way to represent the program as a Directed Acyclic Graph (DAG).

This special kind of graph does not allow cycles between the vertices and the edges are directed from one vertex to the other. This ensures that the graph can always be topologically ordered, which is essential for the response time analysis and C code

generation. Moreover for data-flow programs, DAGs are a convenient way to represent connected and repetitive operations. They also also expose the possibility of parallel computation that can be exploited when implementing the program.

All these DAG characteristics are used in the next section to properly compute the response time of these well behaved and organized applications.

7.4 RESPONSE-TIME ANALYSIS

The MIA tool [118] was built upon a generic framework called Multi-Core Response Time Analysis (MRTA) by Davis et al. [48] with its capability for instantiating different hardware components, memory models, bus arbitration policies and application use cases. The MPPA3 is an eligible hardware architecture to be used with this framework due to its fully timing compositional nature. This section revisits the main concepts of the framework with some additional notations and optimizations allowed by the software model in the first two subsections, while the last three subsections introduce the actual new interference equations for the MPPA3 and their implementation.

7.4.1 *Main Concept*

The framework receives a set of n periodic tasks $\Gamma = \{\tau_1, \ldots, \tau_n\}$, where each task τ_i has a period T_i, a deadline D_i and is statically assigned to a core P. With these information the framework is capable of calculating the response time R_i taking into account the interference suffered at different hardware points throughout the task τ_i execution time.

From now on we use P_x to denote the core under analysis and Γ_x the subset of tasks mapped to this core P_x. In contrast, we use P_y to denote other cores than the one under analysis and Γ_y to indicate the subset of tasks mapped to this core P_y.

The tasks are represented as a set of ordered traces leveraging the demands issued to distinct hardware resources, such as processor or memory. This is then used to compute the response time R_i of task τ_i running on core P_x with the following equation:

$$R_i = \mathrm{W}_i + I^{\mathrm{BUS}}(i, x, R_i) \tag{7.1}$$

where W_i is the processor demand, i.e. the Worst-Case Execution Time (WCET) of the task in isolation and I^{BUS} is the interference on the bus calculated using a mathematical model. The original generic MRTA framework had also terms that accounted for preemption interference and DRAM memory refreshes, which does not apply in our scenario as we assume in Section 7.3 that tasks run until completion and fit in the cluster local memory.

All the necessary inputs for the framework are obtained either during the code generation or by an external tool. The dependencies between the tasks are extracted when generating C code, either from Lustre or SCADE. The mapping and scheduling is an orthogonal work and it can be automatically done by the integration script mentioned in Section 7.3.3. The WCET of tasks in isolation can be estimated by tools such as OTAWA [17] or Heptane [78] as long as they have implemented a model of the core architecture in analysis. If that is not the case, an approximate value can be obtained by measurement with an additional step in the framework.

7.4.2 *Additional Definitions and Simplifications*

We add up to the main concept the notion of a release date rel_i for each task τ_i as an offset from the start of a program's period. The set of release dates is given by $\Theta = \{rel_i \mid i \in \Gamma\}$ and the set of upper bound on response times by $\mathcal{R} = \{R_i \mid i \in \Gamma\}$.

Using these new terms, Equation (7.1) becomes:

$$R_i = W_i + I^{\text{BUS}}(i, x, \mathcal{R}, \Theta) \tag{7.2}$$

We recall here that the tasks run on their isolated memory space due to the SMEM banked memory mode, introduced in Section 7.1.2. They may access each others' banks only when communicating. The bus interference function is therefore given by:

$$I^{\text{BUS}}(i, x, \mathcal{R}, \Theta) = \sum_{b \in \beta_i} \text{BUS}_b(i, x, \mathcal{R}, \Theta) \times d \tag{7.3}$$

where β_i is the set of memory banks accessed by task τ_i, $\text{BUS}_b(i, x, \mathcal{R}, \Theta)$ is a function that, according to the arbitration policy, gives an upper bound on the number of accesses to memory bank b which may delay task τ_i and, finally, d is the time to perform a memory access (bus latency) on the worst-case. We recall here Section 3.2.3.1 where the d value for the MPPA3 platform is given. When a memory access is needed, as we target the SMEM and not the DDR, we are in a L1 cache miss scenario, thus $d = 22$ cycles.

In the next sections, we will focus on calculating this bus function. For the memory demand of a task τ_i on memory bank b we will use the notation $S_i^{x,b}(\mathcal{R})$. For the 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 on memory bank b we will use $A_i^{y,b}(\mathcal{R}, \Theta)$.

7.4.3 *Intra-Cluster Interference*

In Section 7.1 we have divided the memory access path into two levels. These levels are hereby reproduced helping to divide the modeling problem into smaller parts that are later on composed together.

7.4.3.1 *Level 1*

The analysis will be further split into cached and cache bypassed accesses as they change the actual behavior of the FP arbiter.

CACHE BYPASS Within this context, the worst-case happens when an IC access is always delayed by uncached loads until the PFB is empty. The PFB holds a maximum of 16 instructions and 1 instruction is issued each cycle. Moreover, the uncached instructions that may already be in the execution pipeline must be also considered, adding up a maximum of 4 instructions. A first approximation is to consider the total interference as the number of IC accesses from a task τ_i multiplied by 20 cycles, as formulated in Equation (7.4). These 20 cycles account for the 16 instructions in the PFB plus the 4 instructions on the pipeline.

$$\text{BUS}_b^{L1_U}(i, x, \mathcal{R}, \Theta) = 20 \times \sum_{k=0}^{S_i^{\text{IC}x,b}} l_k \tag{7.4}$$

Figure 7.3 exposes this worst-case scenario in detail.

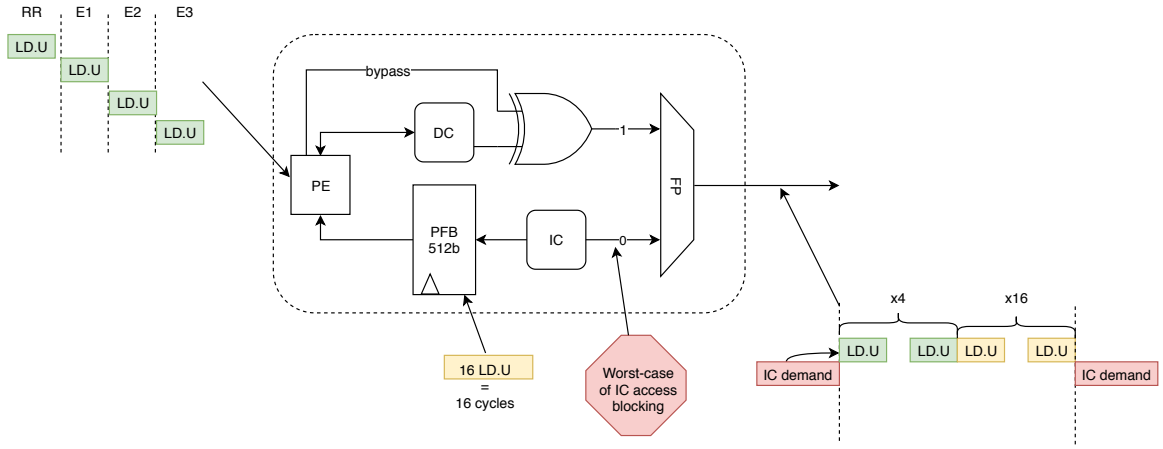


Figure 7.3: Cache bypass worst-case

With a static code analysis method, it is possible to minimize this worst-case. For instance, with the knowledge of the maximum number of uncached loads in sequence from a given task τ_i, the factor of multiplication would be reduced. Note that this is orthogonal work that we do not explore in this thesis.

CACHED ACCESS When instructions that use the cache are issued, there are two possible cases at the FP arbiter when there is a DC request:

1. Cache Hit: IC may be granted access afterwards, as the DC already has the data required by the processor and as the cache latency is of 2 cycles, no other DC request may be issued before this time elapses;
2. Cache Miss: IC is blocked at most 2 cycles due to the no hit under miss[3] policy and the 2 accesses needed to retrieve a cache line.

These restrictions and details on when the requests from DC and IC are sent and what happens next actually modify the behavior of the FP arbiter. It brings it to a RR arbitration from the point of view of the DC and a RR with duplicated DC entries from the IC point of view, which is the worst-case. The interference is thus given by the doubled minimum between the IC and DC memory demand:

$$\mathrm{BUS}_b^{L1_C}(i,x,\mathcal{R},\Theta) = 2 \times \min\left(S_i^{\mathrm{IC}_x,b}, S_i^{\mathrm{DC}_x,b}\right) \tag{7.5}$$

7.4.3.2 *Level 2*

From now on the level 1 arbitration problem will be shortened into $S_i^{x,b}$, which can either use $\mathrm{BUS}_b^{L1_U}(i,x,\mathcal{R},\Theta)$ or $\mathrm{BUS}_b^{L1_C}(i,x,\mathcal{R},\Theta)$. The choice depends on the available information in terms of instruction use and cache misses. At level 2, the interference is given by the RR arbitration formula introduced in [116], a sum of the minimum between

3 The pipeline will stall either with another DC request or if a subsequent instruction has a data dependency on the missed access

the core issuing a memory request and any other initiator. Specifically here, the first line computes the interference after level 1 and any other core ($G1_c$) of the same cluster. The next two lines continue the sum, but considering the other initiators group ($G1_o$). Precisely, the AXI interfaces Write or Read requests (dual-channel protocol) that may come from other clusters, targeting the memory bank b:

$$\begin{aligned}\text{BUS}_b^{L2}(i,x,\mathcal{R},\Theta) = &\sum_{y\in G1_c \wedge y\neq x} \min\left(A_i^{y,b}(\mathcal{R},\Theta), S_i^{x,b}(\mathcal{R})\right)\\ &+ \min\left(A_i^{\text{AXI}_{\text{Write}},b}(\mathcal{R},\Theta), S_i^{x,b}(\mathcal{R})\right)\\ &+ \min\left(A_i^{\text{AXI}_{\text{Read}},b}(\mathcal{R},\Theta), S_i^{x,b}(\mathcal{R})\right)\end{aligned} \tag{7.6}$$

7.4.4 *Inter-Cluster Interference*

In Section 7.2 we have also divided the interference problem between multiple clusters into three levels as illustrated in Figure 7.2. A scenario is created here with a task τ_i running on core P_x in CC_u that communicates via AXI with a task τ_j running on core P_w in CC_v.

Starting at the first arbitration level, inside of the departure cluster (CC_u), using Equation (7.6) to access the AXI virtual bank (either read or write):

$$\text{INTERF}_{\text{SAP}_1} = \text{BUS}_{\text{AXI}}^{L_2}(i,x,\mathcal{R},\Theta) \tag{7.7}$$

Moving on to the second arbitration level, at the edge of the AXI bus on the arrival cluster (CC_v) the DRR arbitration is applied to any access from a core outside of the concerned clusters (CC_u and CC_v) targeting any memory bank inside CC_v:

$$\text{INTERF}_{\text{DRR}} = \sum_{\substack{y\notin G1_{CC_{u,v}}\\ b\in B_v}} \min\left(A_i^{y,b}(\mathcal{R},\Theta), S_i^{x,b}(\mathcal{R})\right) \tag{7.8}$$

Finally, at the third and last arbitration level, we will account for the interference accessing the destination bank b inside the cluster:

$$\text{INTERF}_{\text{SAP}_2} = \sum_{y\in G1_{CC_v}} \min\left(A_i^{y,b}(\mathcal{R},\Theta), S_i^{x,b}(\mathcal{R})\right) \tag{7.9}$$

To take into account the complete interference perceived by the core in analysis during the outbound of the AXI path, we must sum up the previous equations:

$$\text{INTERF}_{\text{Outbound}_{\text{AXI}}} = \text{INTERF}_{\text{SAP}_1}^{CC_u}(i,x,\mathcal{R},\Theta) + \tag{7.10}$$

$$\text{INTERF}_{\text{DRR}}^{CC_u}(i,x,\mathcal{R},\Theta) + \tag{7.11}$$

$$\text{INTERF}_{\text{SAP}_2}^{CC_u}(i,x,\mathcal{R},\Theta) \tag{7.12}$$

7.4.4.1 *Inter-Cluster Return Path*

For completeness sake, the delay perceived throughout the return path of a core waiting for data or an acknowledgement after performing an inter-cluster access will be analyzed here. Nonetheless, this analysis is only required if write operations are not followed by a `fence` instruction[4] or if read operations are being performed, which is against the system design proposed in Section 7.3.2.2.

A detailed view of the AXI connection diagram can be seen in Figure A.2 in Appendix A and specifically the return arbitration points that we are interested in are the 3 RRs traversed during the return path of either an AXI Read data or AXI Write acknowledgement.

The use-case explanation about each RR is provided in Figure A.4 in Appendix A along with a path overview of an AXI transaction in the system. In particular we will investigate only the RR1 and RR2 arbiters, as the RR3 blocks the core pipeline and must be taken into account during the WCET measurement.

For the equations developed here, the same scenario from the previous section will be used: a task τ_i running on core P_x in CC_u that communicates via AXI with a task τ_j running on core P_w in CC_v. These generic identifiers are realized into concrete tasks, cores and clusters in Figure A.4.

The interference at the RR1 arbiter is created by 2 or more different banks of the target cluster (CC_v) that are sending an answer (either Write ACK or Read DATA) back to any cluster after they performed an AXI access.

$$\text{INTERF}_{\text{RR1}_{\text{return}}} = \sum_{\substack{y \notin CC_v \wedge y \neq x \\ b \in B_{CC_v}}} \min\left(A_i^{y,b}(\mathcal{R}, \Theta), S_i^{x,b}(\mathcal{R})\right) \tag{7.13}$$

The interference at the RR2 arbiter is generated by 2 or more different clusters that are sending an answer (either Write ACK or Read DATA) back to the depart cluster (CC_u).

$$\text{INTERF}_{\text{RR2}_{\text{return}}} = \sum_{\substack{y \in G1_{CC_u} \wedge y \neq x \\ b \notin B_{CC_u}}} \min\left(A_i^{y,b}(\mathcal{R}, \Theta), S_i^{x,b}(\mathcal{R})\right) \tag{7.14}$$

7.5 NON-CONFORMANCE WITH THE EXECUTION MODEL

This section exposes the work that has to be done if one wants to use a different hardware configuration, or another processor that does not have the same capabilities as the MPPA. It also covers the adaptations needed if one decides to follow a distinct system design concept. Now that we have explored the response time analysis in the previous section, we will go through all the points in Section 7.3 giving the necessary modifications.

7.5.1 *Architecture Configuration*

We discuss here what needs to be changed on the models to accommodate the possible temporal perturbations or anomalies created when privileging performance over time-predictability.

4 This instruction ensures that all issued memory operations are committed and visible to other cores before unblocking the pipeline

7.5.1.1 *MMU*

The MPPA3 has a Memory Management Unit (MMU) that provides virtual to physical memory address translation and assumes an unified address space for instruction and data. It can be activated through setting a specific bit on the Processing Status (PS) register. On top of address translation, it is also used to ensure memory protection between processes and different caching policies for each virtual page.

These features are typically used in richer OS that deal with multiple processes and typically will also use the Level 2 cache, discussed in the next section. In our framework the MMU is disabled to avoid any timing jitter introduced by Translation Lookaside Buffer (TLB) page lookups, which are heavily dependent on the number and size of pages. Certain configurations and features of the MMU [25] can indeed be used to provide virtual addresses while maintaining time predictability. Static mapping, for example, provide flexibility with low timing impact. For other scenarios, a statistical analysis of the different parameters could be done to estimate the introduced jitter.

7.5.1.2 *Level 2 Cache*

Using part of the SMEM as a L2 cache is a valid configuration on the MPPA3 and can be used to increase performance. For time critical applications, activating the L2 cache comes at the expense of requiring a cache hit or miss analysis to only then apply the appropriate SMEM latency values.

The L2 cache controller is implemented through an optional firmware in the MPPA3 and thus, its software should be analyzed to see if it cannot introduce unexpected timing variations to the response time. Another possibility is to use a richer OS enabling the use of cache partitioning techniques [89] in order to reduce the interference without requiring any additional hardware support.

7.5.1.3 *Cluster Memory*

The L2 cache utilization reduces the amount of available memory inside the cluster. This may have an impact on the assumption that all of the code and data of a program fit inside this space. Therefore, if DDR accesses are required, the controller and arbitration on this memory level needs to be modeled, as well as its access latency [142]. Defining precise rules for DDR accesses [110] such that the system is predictable is also a valid option, even if there is a loss in performance.

If the program's size is reasonable, its execution can remain within the cluster bound memory. However, if the interleaved mode is used, or there is no concept of independent memory banks, the amount of interference will increase significantly, as all the access from all the cores will pass through the same arbiter.

7.5.1.4 *Inter-Cluster Communication*

The NoC can also be used to perform accesses to the SMEM belonging to other clusters. In this case, the Worst-Case Traversal Time (WCTT) needs to be calculated [65] using real-time network calculus and a route planning algorithm. The DMA mechanism used to issue NoC packets works differently than the MPPA2 version and must be taken into account.

In order for a packet to be sent, the user must create a thread in a DMA queue that will be scheduled later on. This implies that the scheduling policy needs to be modeled to tightly estimate and then add this time to the WCTT.

7.5.2 *System Design*

Some software restrictions were established in Section 7.3.2 and now the analysis will be opened up to more general purpose written programs.

7.5.2.1 *Alignment and Access Size*

Code compiled without care regarding the recommended 8-byte alignment can incur into duplication of accesses and more latency when interacting with the memory.

Firstly, as already stated in Section 7.3.2.1, a misaligned load near the 256-bit boundary and that crosses it by more than 2 bytes will trigger a micro expansion creating two accesses for this load instruction. In consequence, this blocks the IC grant at most 2 cycles, because the expansion is done after the caches but before the FP arbiters. In particular, for the cache bypass worst-case (Figure 7.3) this doubles the IC starvation time, reaching at most 40 cycles. Further on it will require 2 grants from the SAP.

Different types of accesses can trigger a Read/Modify/Write behavior after the SAP arbiter, just before entering the memory bank. They are: any sub size access (less than 32b), any access that is not 8-byte aligned, and any write operation bigger or equal to 8-bytes, even if it is aligned. This R/M/W operation normally happens only if a load operation returns with a checksum error or when performing atomic instructions. This behavior adds 4 cycles in latency and 1 cycle in bandwidth.

7.5.2.2 *Reading versus Writing*

General purpose programs interact in more diverse ways than our proposal of only reading from its own memory bank while writing could be done to any bank. If the memory is in interleaved mode, this assumption is already false.

So if the interleaved mode is used or reading operations are done outside of a task's own memory bank, the interference model must take into account the return path and its arbitration to the target core. There is a RR arbiter receiving the response from different memory banks, as can be seen in Figure A.1 in Appendix A.

7.5.3 *Software Framework*

A generic program could be fitted into this analysis if it is at least divided into distinct tasks that we can analyze and extract their communication patterns and dependencies and deploy to different cores. In other words, the program should be translatable to a DAG format and enriched with communication information. Without this particular organization, we can not compute the program's Worst-Case Response Time (WCRT), only its WCET, which is not sufficient for a multi or many-core architecture.

7.6 CONCLUSION

An in-depth analysis of the MPPA3 architecture has been presented, highlighting the needed components in order to tightly compute the possible interference in all arbitration points of the system. This description was then transformed in mathematical models and implemented in MIA, a part of the bigger framework from Chapter 4. MIA is responsible for estimating the WCRT and provide appropriate release dates for the applications.

Some possible improvements in the presented analysis for time-critical computing on the MPPA3 are:

- Modeling the memory access by taking into account the transaction pipelining and not only the worst-case between the start and end of a transaction, i. e. consider non-blocking accesses;
- Improving the AXI DRR arbiter modeling, as we have considered it as a regular RR due to the burst nature of the protocol.

Part III

EVALUATION

8

TOOL EXTENSIONS

The presentation of the workflow developed throughout this thesis in Chapter 4 put in evidence the interaction and requirement of additional software tools to construct an efficient and safe framework for the generation of safety-critical real-time systems. The context of this work is at the boundary of the hardware and low-level software, which means that according to the platform and the type of code that we want to generate, the tools that will be used need to be extended, improved or adapted.

This chapter presents the improvements and extensions done on tools that existed before this thesis or developments based on standards that were already public. The first one is the Multi-Core Interference Analysis (MIA) tool in Section 8.1, which is used in our framework to estimate the global Worst-Case Response Time (WCRT) of applications and verifiy their schedulability. An improved core algorithm was developed that reduces significantly the complexity and allows better scalability of the tool. An arbitration model of our target platform, the Kalray MPPA3 was implemented in this new version and is used in Chapter 9 for the experiments.

Initialization and communication code needs to be generated to integrate the functional code obtained after the compilation of synchronous data-flow languages. This integration code is strictly platform dependent and Section 8.2 describes the complete code generation process and the changes required for our framework and target.

Hardware abstraction models are extensively used in the academic and industry to speed up the generation of code for a variety of platforms at the same time. With the hardware knowledge of the Kalray MPPA3 obtained during this thesis, in Section 8.3 we develop a Software-Hardware Interface for Multi-Many-Core (SHIM) model of this processor that is composable and reusable, following a established IEEE standard.

8.1 MULTI-CORE INTERFERENCE ANALYSIS (MIA)

This section highlights the improvements made on an existing tool called MIA. First a context on the response time analysis problem and the software is given, what motivated its revision, the new proposed algorithm and the implementation of the Kalray MPPA3 models introduced in Chapter 7.

8.1.1 *Response Time Analysis*

Due to the nature of safety-critical real-time systems, they must be rigorously verified. This verification covers not only the functionality of the program but also if it delivers the correct responses on time. At the same time, the embedded industry is also transitioning from single to multi/many-core processors. Having multiple cores on a system also means that they will be in concurrency when accessing shared resources, such as the memory. This concurrency problem is solved by arbiters implemented in hardware. Their role is to decide the order in which the data traffic requests will go through. The algorithm implemented in each arbiter is what ultimately impacts the predictability and performance of these accesses.

Therefore in order to construct a complete response time analysis there are multiple steps to be followed, described in richer detail in Section 3.3. The first one is to compute or estimate the Worst-Case Execution Time (WCET) in isolation of the program. For multi/many-core platforms, when multiple cores access the memory at the same time, they are arbitrated and slow each other down. This delay is called *interference* and must be taken into account. Once this is considered we talk about WCRT of a program, instead of just WCET. The WCRT defines the time between the release date until the end of the task execution, taking into account the interference delay. The final goal of this analysis is to be part of a time-triggered execution, where each task within a program is deployed at a specific point in time. Thus, the release dates of the tasks must also be computed using the WCRT and the global map and schedule.

8.1.2 *Problem Statement*

Let us now formulate the problem that we aim to solve with the response time analysis technique. As previously stated, we are within a time-triggered schedule that gives us the precise time when each task of a program will be released. This time is strictly respected during the execution, even if all the task's dependencies were already satisfied and the task could eventually start earlier.

The notion of release date is essential for the interference computation. A given task τ_1, running on core C_1, with a release date rel_{τ_1} and a response time R_{τ_1} can only interfere with another task τ_2, running on core C_2, with a release date rel_{τ_2} and a response time R_{τ_2} if there is an overlap in their execution time, i.e. $[\, rel_{\tau_1} + R_{\tau_1} \,] \cap [\, rel_{\tau_2} + R_{\tau_2} \,]$. If they do not overlap, we can guarantee the absence of interference between τ_1 and τ_2.

Our objective is that, given a Directed Acyclic Graph (DAG) of tasks with their intrinsic dependencies and communication, their WCET in isolation, their memory accesses, an initial mapping and scheduling and a bus arbiter model, we are able to compute release dates for each one of the tasks and the total WCRT of the graph. Optionally the tasks may

have minimal release dates that forbid them from being scheduled before that date, even if the tasks that they depend upon have already finished.

The challenge in solving this problem is that the WCRT of each task (the WCET in isolation plus the interference) is directly influenced by the release date of the task itself and of the others that may overlap with it. Therefore, changing the release dates of tasks may also alter the amount of interference and WCRT, which can impact the release dates of tasks that will be scheduled later on. From a global point of view, this creates a circular dependency problem that increases the complexity of the algorithm trying to solve it.

To concretely understand the problem an example is given here, containing a DAG with its initial and final schedule. The final schedule takes interference into account and shows how it changes the WCRT of the program.

EXAMPLE 5 (RESPONSE TIME ANALYSIS OF A DAG): *The task set Γ is composed of 4 tasks: n_0, n_1, n_2, n_3, n_4. Their WCET in isolation is respectively 2, 2, 1, 3 and 2. The tasks also have minimal release dates: $t = 0$ for n_0, n_3 ; $t = 2$ for n_1 ; $t = 4$ for n_2, n_4. The tasks that communicate with each other write only 1 unit of data to their target and they are $n_0 \rightarrow n_1$; $n_0 \rightarrow n_2$; $n_3 \rightarrow n_2$; $n_3 \rightarrow n_4$. These values can be seen on the edges between the nodes in Figure 8.1.*

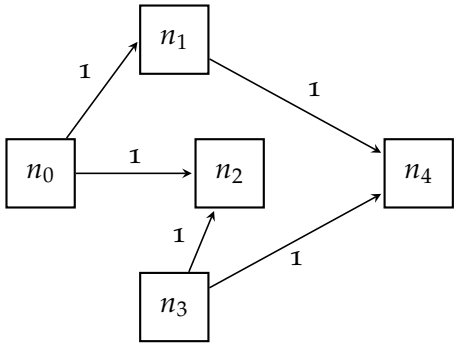


Figure 8.1: DAG under analysis

We suppose the execution of this DAG on a platform with 4 cores. The mapping of these tasks to the available cores follows: $n_0 \mapsto C_0$; $n_1, n_2 \mapsto C_1$; $n_3 \mapsto C_2$; $n_4 \mapsto C_3$. An initial schedule is given in Figure 8.2. Time starts at $t = 0$ and each vertical dashed line represents 1 unit of time.

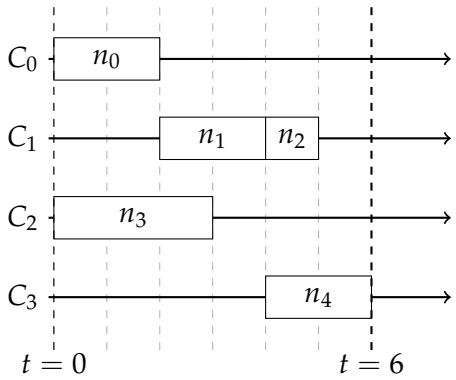


Figure 8.2: Initial schedule for DAG under analysis

This schedule is valid as it respects the minimal release dates and the dependencies from the tasks that are communicating. The response time of the whole application is $t = 6$. However this analysis

does not consider the interference when tasks are writing data to the same shared memory. The final schedule is given in Figure 8.3.

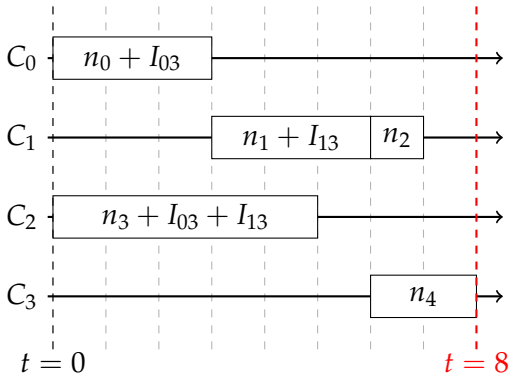


Figure 8.3: Final schedule for DAG under analysis

There is interference between tasks n_0 and n_3 (represented as I_{03} in Figure 8.3) as their response time overlap and they write to the same target n_2. The same applies to tasks n_1 and n_3 (represented as I_{13}) as they both write to n_4. This increases each task response time, and in particular delays the release date of task n_4 to $t = 6$. The WCRT of the application is thus $t = 8$.

8.1.3 *Original Algorithm*

Previous work has solved the analysis problem that was introduced in Section 8.1.2. A general and extensible framework for computing a bound on the delay due to interference is presented in [48] under the name Multi-Core Response Time Analysis (MRTA). It is extensible as, multiple architectures with different arbitration policies to shared resources and distinct memory systems, can be modeled in it. One major contrast with the problem statement in Section 8.1.2 is that this work allows preemption and supposes sporadic tasks with minimum inter-arrival times, without dependencies.

A derived work appears in [118] as an extension of the MRTA framework by modeling the arbitration system of an industrial processor and its memory bank scheme. Another change is the limitation of the application scope to a DAG based model with precedence constraints and communication, allowing only periodic tasks and no preemption, which is aligned with our problem description.

A simplified version of the general algorithm is presented in Algorithm 7. By knowing the minimal release dates and initial response times of the task set, the algorithm uses two fixed-point iterations to solve the response time analysis. As discussed before, these fixed-point iterations are required as altering the release dates may also alter the response time and vice-versa.

The function ComputeResponseTimes (Line 3) uses the shared resource arbiter model to compute the response times of tasks $\{\tau_1, \ldots, \tau_n\}$, with the release date set Θ, while bounding the interference from co-runners. This is the first-fixed point iteration as the function stops once the response times of the tasks are constant.

Function UpdateReleaseDates (Line 4) verifies and updates the release date set based on the current response time set and the tasks dependencies. The second fixed-point iteration occurs at the end of the while loop (Line 6): in order to finish the algorithm the release dates must be constant between two different iterations.

Algorithm 7: Original scheduling algorithm

Input: Set of release dates $\Theta = \{rel_1, \ldots, rel_n\}$,
set of response times $\mathcal{R} = \{R_1, \ldots, R_n\}$,
set of deadlines $D = \{D_1, \ldots, D_n\}$ of tasks $\{\tau_1, \ldots, \tau_n\}$
Output: `schedulable`, Θ, $\mathcal{R}$ **OR** `unschedulable`

1 $l \leftarrow 0$, $\Theta^l \leftarrow$ InitRel(), $\mathcal{R}^l \leftarrow \perp$;
2 **do**
3 $\quad \mathcal{R}^{l+1} \leftarrow$ ComputeResponseTimes(Θ^l);
4 $\quad \Theta^{l+1} \leftarrow$ UpdateReleaseDates($\Theta_{min}, \Theta^l, \mathcal{R}^{l+1}$);
5 $\quad l \leftarrow l + 1$;
6 **while** $\Theta^l \neq \Theta^{l-1}$;
7 **if** $\forall i : (\Theta[i]^l + \mathcal{R}[i]^l \leq D_i)$ **then**
8 $\quad$ **return** "*schedulable*", Θ, $\mathcal{R}$;
9 **else**
10 $\quad$ **return** "*unschedulable*";

The complexity of this algorithm was proved to be $O(n^4)$ [116] where n is the number of tasks to schedule. This raises scalability issues as soon as n starts to go over hundreds of tasks. The complexity mainly comes from the two functions in Lines 3 and 4. ComputeResponseTimes reaches $O(n^3)$: n iterations to converge, n iterations to calculate the response time of all tasks and $n-1$ iterations to compute the interference function. UpdateReleaseDates reaches $O(ne)$: iterates over all tasks and its dependencies, which are the graph edges.

The emergence of modern real-time systems with an ever increasing set of functionality raises the urge for a more efficient algorithm to compute the WCRT. The new algorithm proposed here is inspired by the termination proof in [116], where a time cursor is used to show that at each iteration a new task gets its definitive release date. Instead of aiming for fixed-point computation as in Algorithm 7, the cursor is used to iterate through the finite task set and their dependencies.

8.1.4 *Proposed Algorithm*

This section investigates how Algorithm 7 was reformulated to reduce the complexity, allowing it to be applied to huge real-time systems.

8.1.4.1 *Approximations and Hypotheses*

Before diving into the new algorithm, it is important to put in evidence some of its assumptions. We assume the following constraint: adding a new task to the program can only increase the interference received by other tasks. It can seem as an intuitive statement, but it is required for the proof of correctness on Section 8.1.4.4. In order to be more general we assume that the interference might be *non additive*, i. e. the interference between n tasks is not necessarily the sum of the interferences between all pairs of tasks. If a bus arbiter do have this *additive* property, exploiting it could simplify and even speed up the algorithm, but this is not the general case.

When multiple tasks are mapped to the same core, we assume the hypothesis that they can be treated as a single big task, summing their WCETs and memory accesses. No timing overlap analysis between tasks is done, which may lead to more pessimistic results. This incurs that if any task accesses the memory bank privatized to a certain core it will provoke interference. Figure 8.4 shows two diagrams that are topologically equivalent when considering a task set where n_1 and n_2 interfere with n_0. They are perceived as one big task by n_0 as the diagram on the right shows.

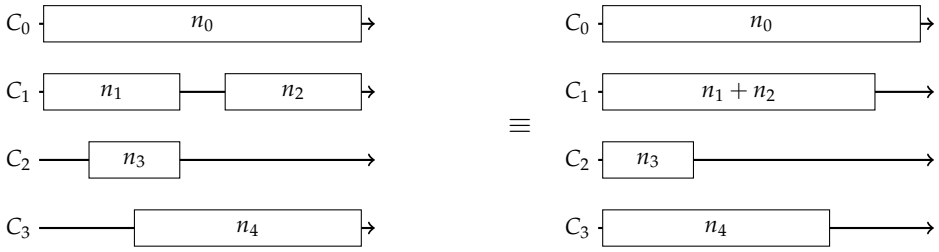


Figure 8.4: Equivalence between diagrams for interference calculation

8.1.4.2 *Core idea*

Given the task set and initial release dates, the proposed algorithm works incrementally, by adding tasks one by one to the schedule. The algorithm works with a time cursor t, starting from $t = 0$ and progressing forward. The tasks are divided into three groups:

- Closed ($\mathcal{C}$): t is after their finish date. These tasks have both their final release date and response times computed.
- Alive ($\mathcal{A}$): t is between their release and finish date. These tasks have their final release date, but their response time may be influenced by tasks not yet added to the schedule.
- Future: t is before their release date, neither the release date nor their response time is computed.

At each iteration, the cursor t jumps to the nearest end date of the current alive tasks or the minimal release of future tasks, whichever is smaller. New available tasks, i.e. with all dependencies satisfied, are then scheduled, and the interferences that they add to and receive from the current alive tasks are computed. They cannot interfere with dead tasks, because they do not overlap, and their interferences with future tasks will be computed later in the algorithm, when those are added to the alive group.

With this approach, when a task is scheduled, its release date is definitively set and, as previously discussed, will not be changed with future tasks.

Figure 8.5 captures a snapshot of the algorithm being executed. The vertical dashed red line represents the current time cursor position. Only the solid boxes are considered alive tasks. The dotted boxes on the left are the dead tasks, and the dashed ones on the right are the future tasks.

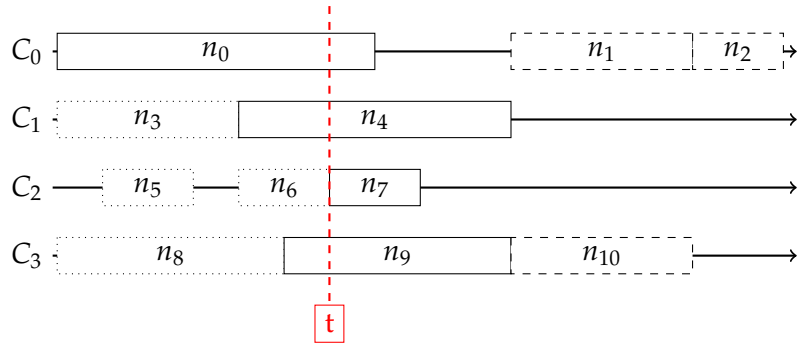


Figure 8.5: Snapshot of the new algorithm cursor mechanism

8.1.4.3 *Detailed Algorithm*

The proposed algorithm is given in Algorithm 8 as pseudo code, and detailed below. The inputs are a task set Γ, an initial set of release dates Θ and response times $\mathcal{R}$, the number of cores c available in the platform, the mapping of tasks to cores and a shared memory, which may have distinct arbitrated banks reserved for each core to minimize interference[1].

In the example from Figure 8.5, we have $\Gamma = \{n_0, \ldots, n_{10}\}$, $c = 4$ and the mapping is as follows: $n_0, n_1, n_2 \mapsto C_0$, $n_3, n_4 \mapsto C_1$, $n_5, n_6, n_7 \mapsto C_2$ and $n_8, n_9, n_{10} \mapsto C_3$.

The time cursor begins at $t = 0$, with $\mathcal{A}$, the set of current alive tasks, initially empty. The following steps are then repeated until all the tasks are scheduled (at each step we give the corresponding state in the example from Figure 8.5 and the lines from the Algorithm 8):

1. $\mathcal{C}$ (closed) is the set of tasks ending at time t. It is simply computed by scanning the current alive tasks, and determining if the end of the task (*rel* + WCRT) is less than or equals to t. These tasks are then removed from their reverse dependencies list, allowing tasks depending on these closed ones to start.

   Algorithm: Line 3 to Line 6, *Example:* $\mathcal{C} = n_6$

2. $\mathcal{A}$ (Alive) $\leftarrow \mathcal{A} - \mathcal{C}$

   Algorithm: Line 7, *Example:* $\mathcal{A} = n_0, n_4, n_9$

3. $\mathcal{O}$ (Opening) is the set of tasks opening at time t. It is computed by scanning the head of the stack of scheduled tasks for each core, and determining whether its dependencies are satisfied and if its minimal release date is smaller than or equal to t.

   Algorithm: Line 9 to Line 15, *Example:* $\mathcal{O} = n_7$

4. $\mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{O}$

   Algorithm: Line 16, *Example:* $\mathcal{A} = n_0, n_4, n_7, n_9$

5. For any *destination* task in $\mathcal{A}$ and for any *source* task in $\mathcal{A}$, which have accesses to the same memory bank, we determine if the source task has already been accounted for

1 If the memory is composed of only one bank (or the banks are interleaved and memory accesses cannot be reliably associated to a bank), the loop iterating over banks on Line 17 to Line 23 of Algorithm 8 iterates over only one element and can be simplified. However, while the analysis is simplified, the worst-case interference is increased as all access from all cores, even to their private code and data, may be delayed by others.

Algorithm 8: Proposed scheduling algorithm

Input: Set of release dates $\Theta = \{rel_1, \ldots, rel_n\}$,
set of response times $\mathcal{R} = \{R_1, \ldots, R_n\}$,
set of deadlines $D = \{D_1, \ldots, D_n\}$ of tasks $\{\tau_1, \ldots, \tau_n\}$
Output: `schedulable`, Θ, $\mathcal{R}$ **OR** `unschedulable`

1 **forall** k, $S_k \leftarrow$ stack of tasks scheduled on core k; $\mathcal{A} \leftarrow \varnothing$; $t \leftarrow 0$;
2 **while** $t < +\infty$ **do**
3 $\quad \mathcal{C} \leftarrow \{\tau \in \mathcal{A} \mid (rel_\tau + \mathcal{R}_\tau) \leq t\}$;
4 $\quad$ **for** $\tau \in \mathcal{C}$ **do**
$\quad\quad$ `// `τ`.rev_deps →` `tasks that depend on` τ
5 $\quad\quad$ **for** τ' *in* τ.rev_deps **do**
6 $\quad\quad\quad$ τ'.deps.`remove`(τ);
7 $\quad \mathcal{A} \leftarrow \mathcal{A} - \mathcal{C}$;
8 $\quad \mathcal{O} \leftarrow \varnothing$;
9 $\quad$ **for** $k \in$ *list of cores* $c = \{0, \ldots, c-1\}$ **do**
10 $\quad\quad$ **if** S_k *is not empty* **then**
$\quad\quad\quad$ `// get top of stack without removing`
11 $\quad\quad\quad$ $\tau\_next \leftarrow S_k$.`peek`$()$;
12 $\quad\quad\quad$ **if** τ_next.deps *is empty* ***AND*** *min_rel of* τ *is* $\leq t$ **then**
13 $\quad\quad\quad\quad$ $\mathcal{O} \leftarrow \mathcal{O} \cup \{\tau\}$;
14 $\quad\quad\quad\quad$ $rel_\tau \leftarrow t$; `// updates release date`
15 $\quad\quad\quad\quad$ S_k.`pop`$()$; `// removes top of stack`
16 $\quad \mathcal{A} \leftarrow \mathcal{A} \cup \mathcal{O}$;
17 $\quad$ **for** $\tau\_dest \in \mathcal{A}$ **do** `// task target of mem access`
18 $\quad\quad$ **for** $\tau\_src \in \mathcal{A}$ **do** `// task source of access`
19 $\quad\quad\quad$ **for** *bank* b *in banks* $\mathcal{B}$ **do**
20 $\quad\quad\quad\quad$ **if** τ_dest *and* τ_src *both access* b **then**
21 $\quad\quad\quad\quad\quad$ **if** τ_src *not in* τ_dest.interfers_with$[b]$ **then**
22 $\quad\quad\quad\quad\quad\quad$ τ_dest.interfers_with$[b]$.`add`(τ_src);
$\quad\quad\quad\quad\quad\quad$ `// updates response time`
23 $\quad\quad\quad\quad\quad\quad$ τ_dest.interferences$[b] \leftarrow I^{\text{BUS}}(\tau_dest, \tau_dest$.interfers_with$[b], b)$;
24 $\quad t\_next \leftarrow +\infty$;
25 $\quad$ **for** $\tau \in \mathcal{A}$ **do**
26 $\quad\quad$ $t\_next \leftarrow \min(t\_next, rel_\tau + \mathcal{R}_\tau)$;
27 $\quad$ **for** *min_rel in minimal release of future tasks* **do**
28 $\quad\quad$ $t\_next \leftarrow \min(t\_next, min\_rel)$;
29 $\quad t \leftarrow t\_next$;
30 **if** $\forall i : (\Theta[i]^l + \mathcal{R}[i]^l \leq D_i)$ **then**
31 $\quad$ **return** "*schedulable*", Θ, $\mathcal{R}$;
32 **else**
33 $\quad$ **return** "*unschedulable*";

in the interferences received by the destination. If not, that interference is recomputed by the specified bus arbiter function (see Section 8.1.5), after adding the source of the list of nodes that the destination interferes with. Notice that at least one of the source and destination task must be in $\mathcal{O}$, otherwise it would already have been accounted for. The interferences are computed separately for each memory bank access from the task τ. The total interference received by the task τ is the sum of those values.

Algorithm: Line 17 to Line 23

6. t is updated to the minimal value between the next smallest release date of future tasks and the next finish time of alive tasks.

   Algorithm: Line 24 to Line 29

8.1.4.4 *Termination and Correctness*

TERMINATION Except the outermost loop iterating over t, all the other loops iterate over finite sets, which by definition ensures that they will end. The main while loop finishes too, as there are at most $2n$ possible values for *t_next*: each jump of t is to the beginning or end of a task, and those, once visited, will never change later.

CORRECTNESS We give here an intuition on the formal proof of the algorithm. Assume the algorithm is correct when scheduling the $n-1$ first tasks. When the n-th task is added, its release date is definitively set and it may still only interferes the alive tasks. Once the interference computation is done, the algorithm finishes and, by recurrence, all previous dependencies and release dates are respected, providing a correct and final schedule for the task set.

EQUIVALENCE According to the results of the tests performed, this algorithm is conjectured to be equivalent to the original one. This remains to be formally proven though.

8.1.4.5 *Complexity Analysis*

The concepts seen in Section 8.1.4.2 and in the Algorithm 8 are important to understand the complexity analysis in this section. In particular, we recall here the definition of the set of alives tasks $\mathcal{A}$: the cursor t used in the algorithm is between the release date and the finish date of any task contained in this set.

The size of the set of alive tasks $\mathcal{A}$ is bounded by the number of cores, as there can be only one alive task per core at a given time. Therefore, we access the linear I^{BUS} function (Line 23 of Algorithm 8) a bounded number of times for each progression of t. This I^{BUS} function contains the implementation of the model that computes the interference in a given bus. The possible values for t are the tasks finish dates and their minimal release dates, making it at most $2n$. The two nested loops then give an overall complexity, with n tasks, b banks and c cores:

$$O(c^2 \cdot b \cdot n^2) \tag{8.1}$$

For a given processor, b and c are constants, so Equation (8.1) may be simplified to $O(n^2)$.

8.1.5 *MPPA3 Arbitration Model Implementation*

The Kalray MPPA3 arbitration model described in Chapter 7 was implemented in the new python version of the MIA tool[2] presented in Section 8.1.4. We describe here the main contributions to the tool when writing this new model.

8.1.5.1 *Intra-Cluster Arbitration*

LEVEL 1 The cache Fixed Priority (FP) arbiter detailed in Section 7.1.1 and the proposed models to incorporate it into the response time analysis step of our workflow are a new addition to MIA. The implemented versions are still experimental and vary according to the model. They also suppose a static code analysis to obtain the amount of uncached access and their frequency. In practice, in our workflow we suppose that the cache arbitration delay is incorporated into the WCET of a task.

LEVEL 2 The arbiter detailed in Section 7.1.2 is essentially a configurable Round-Robin (RR). This configuration applies to the amount of consecutive grant rounds provided to each initiator. The implementation of this model is straightforward and similar to the MPPA2 that was ported to this new MIA version.

8.1.5.2 *Inter-Cluster Arbitration*

The major contribution added to MIA by modeling the MPPA3 is the implementation of the inter-cluster arbitration as a multi-level problem. This is detailed in Section 7.2 and the summary is: an intra-cluster arbitration from the source, a Deficit Round-Robin (DRR) arbitration at the edge of the arrival cluster and another intra-cluster arbitration to access the destination memory bank.

The MPPA2 model had additional tasks T_x and R_x to represent the communication through the NoC. This new model using the AXI greatly simplifies the implementation, only requiring to sum, on top of interference computed by the arbiters, the constant traversal time between clusters (see Table 3.1) multiplied by the number of data blocks exchanged (see Equation (6.4)).

8.2 PARALLEL CODE GENERATION AND ORCHESTRATION

The workflow introduced in Chapter 4 contains a code generator step identified as Multi-Core Code Generator (MCG). This tool is part of the SCADE Suite [40] and will be described in finer details in this section, as well as the modifications done in the integration script to target the MPPA3 processor.

8.2.1 *From sequential to parallel code generation*

The original code generator contained in SCADE version 6 is called Qualified Code Generator (KCG). It is certified at high levels for critical industries [28]: avionics at Level A of DO-178B, automotive at SIL 3 level of the IEC 61508 standard. However the code

2 Its code is available at the following link: `https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/synchrone/mia`

produced is not suitable for multi-core execution. All SCADE programs have intrinsically a root operator that manages inputs and outputs and encompass all other nodes of an application. KCG generates a single sequential function for the root operator.

Some workarounds outside the SCADE model where possible, but lead to problems due to manual code requiring to be written. These workarounds are also hard to automate as the inner task model also requires changes that is out of the scope of a typical SCADE user. MCG in its turn introduces a new `C:task pragma` to the language, that allows a developer to indicate that an instace of an operator should be extracted into a separate task. The code generation process then produces, instead of a single sequential function, a set of tasks that communicate through one-to-one channels.

8.2.2 *Parallel Code Generation overview*

MCG therefore generates the functional tasks code as well as a decoupled communication model through individual channels. One task executes the root operator of the model. Then, for each annotated operator in the input model, one task is generated. This task receives data on an input channel, calls the operator and then sends the result on an output channel. The implementation of isolated tasks that communicate through message passing is a common technique for multi-core execution, in particular for safety-critical systems.

Each task has a context that contains: memory structures, outputs, several methods and probes to inspect internal state. These methods are `reset` and `init` as well as one or several `cycle` methods. The `cycle` method in itself can be split into several other methods, which is not possible with KCG. In order to communicate with other tasks each `cycle` method has input and output channels. Methods of the same task can communicate directly through its context element.

For each channel used for communication a structure type is generated, containing the data to be transmitted in this channel. The channel is basically a FIFO of size one, connected to one sender and one receiver. The sender writes to and the receiver read from a field in the task's context, which is a pointer to the structure type of the channel. The actual allocation and implementation of the channel is part of the integration code.

EXAMPLE 6 (SCADE MODEL AND GENERATED CODE): *To exemplify the code generation process Listing 8.1 SCADE Model is used, containing only two tasks:*

```
function F1 ( i1 : int32 ) returns ( o1 : int32 )
  o1 = i1 * i1 ;

function F2 ( i2 : int32 ) returns ( o2 : int32 )
  o2 = ( i2 -1)*( i2 +1);

function root ( i : int32 ) returns ( o : int32 )
  o = #pragma kcg C : task F1_task #end F1 ( i )
    - #pragma kcg C : task F2_task #end F2 ( i );
```

Listing 8.1: Simple SCADE Model

Figure 8.6 illustrates this model graphically:

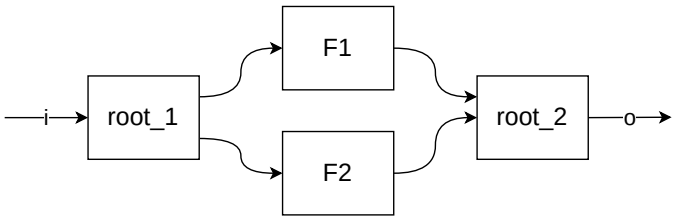


Figure 8.6: Simple SCADE Model diagram

Listings 8.2 and 8.3 show, respectively, the channel structure and task context code that is generated for the `F1_task`:

```
/* channel types */
typedef struct kcg_tag_F1_task_in_ch_type {
  kcg_int32 i1 ;
} F1_task_in_ch_type ;
typedef struct kcg_tag_F1_task_out_ch_type {
  kcg_int32 o1 ;
} F1_task_out_ch_type ;
```

Listing 8.2: `F1_task` channels

```
  /* context type */
typedef struct {
/* ---------------- no memorised outputs ---------------- */
/* ------------------ no local probes ------------------ */
/* ------------------ no local memory ------------------ */
/* --------------------- channels ----------------------- */
F1_task_in_ch_type * /* o=(F1)/ */ F1_task_in_ch;
F1_task_out_ch_type * /* o=(F1)/ */ F1_task_out_ch;
/* --------------- no sub nodes' contexts --------------- */
/* ------------ no clocks of observable data ------------ */
} outC_F1_task;
```

Listing 8.3: `F1_task` context

Listing 8.4 shows the body of the cycle function of the `F1_task`. It is possible to identify that it calls the `F1` method using the contents of the input channel `F1_task_in_ch` and writes the results to the output channel `F1_task_out_ch`, by using the associated context fields.

The code generated for the `root` operator is similar, but due to the `#pragma kcg C:task F1_task #end` and `#pragma kcg C:task F2_task #end` annotations, this operator is split in two: `root_1` and `root_2`.

```
  /* cycle function */
void F1_task ( outC_F1_task * outC )
{
(* outC->F1_task_out_ch ).o1 = /* o =( F1 )/ */
  F1 ((* outC - > F1_task_in_ch ). i1 );
```

Listing 8.4: `F1_task` cycle function

```
void root_1 ( inC_root * inC , outC_root * outC )
{
  (* outC - > F2_task_in_ch ). i2 = inC - > i ;
}

void root_2 ( outC_root * outC )
{
  outC - > o = (* outC - > F1_task_out_ch ). o1 -
    (* outC - > F2_task_out_ch ). o2 ;
}
```

Listing 8.5: `root` task cycle function

8.2.3 *Integration*

In itself the generated code for the tasks is simply a high level structure that needs to completed so that it can be properly executed in a given hardware architecture. Therefore there is an integration step that must be done to run the program on a concrete platform.

8.2.3.1 *Platform-agnostic part*

This section presents the code generation that must be performed independetly of the target platform and that is thus mostly generic and portable following the `C` standards. The generation structure described here was extracted from the examples supplied by the SCADE library and this structure was simply replicated for our platform. Section 8.2.3.2 describes the specificities required by the Kalray MPPA3 hardware.

TASK ALLOCATION The assignment of the generated tasks to the available computing resources constitutes the allocation process. The same core that runs a set of tasks sequentially is called *worker*. The order in which the methods are called in a given worker, i. e. initial scheduling, is determined by the data-flow dependencies coming from channels and additional dependencies between methods if they belong to the same task. This problem is not specific to MCG and was explored in details in Chapter 6. At the end of the code generation, MCG supplies a mapping file that is used to help understand the dependencies in order to provide a valid allocation.

PROGRAM STRUCTURE The final integration code (typically the `main` in a `C` program) must perform several tasks for the uppermost (root) level:

1. Allocate the context of the root operator
2. Allocate the input and output structure of the root operator

3. Initialize the context by invoking the `init` method
4. For each step of the operator:
   a) Set the inputs
   b) Call the `cycle` function
   c) Read the outputs

Additionally as the code for the workers is generated independently, MCG integration code must perform more tasks:

1. For each worker, allocate the context of its tasks
2. For each worker, initialize the context by calling the `init` method
3. For each step:
   a) Set the inputs
   b) For each worker, call the methods in a valid order according to the dependencies. For each method:
      i. Receive the data on input channels
      ii. Call the method
      iii. Send the data on output channel
   c) Read the outputs

From the steps described here, the Acquisition Execution Restitution (AER) or Read Execute Write (REW) phased patterns introduced in Section 2.4 and explored in Chapter 5 can be easily retrieved. In particular when dealing with the inputs and outputs from each operator and the data receival and sending process in channels.

8.2.3.2 *Platform dependent part*

SCADE Suite provides a helper script and a Python library to ease the integration with MCG generated code. The script uses the Mako template library[3] for code generation. It provides an easy way to access the information contained on tasks, methods and channels in order to output `C` code.

All the implementation details described in this section concern the code geeneration of our target hardware, the Kalray MPPA3. For a different platform these need to be reworked. This platform dependent implementation is required to have an executable code, integrated with the platform agnostic generation part presented in Section 8.2.3.1. For the Kalray MPPA3 we have defined two execution modes targetting general purpose and hard real-time use, respectively. They are the *event-triggered* and *time-triggered* modes.

The *event-triggered* mode uses the hardware synchronization mechanisms described in Section 3.2.5: Advanced Programmable Interrupt Controller (APIC) and mailboxes. They are configured so each Processing Engine (PE) to PE connection has a dedicated mailbox configured and when a synchronization event must occur, the target PE receives an interruption. As a given PE may receive multiple interrupts, sometimes out of order,

3 Documentation available at https://www.makotemplates.org/

tokens are used to verify that the data corresponding to a certain communication channel has indeed arrived. There is no response time analysis made on this code, but it is used as a functional reference of the application. An example of the *event-triggered* code for a task is given in Listing 8.6.

```
char stack_PE1[PE_STACK_SIZE] __attribute__((section(".data_bank1"), aligned(8)));
void *thread_PE1(void *arg)
{
  size_t i;
  outC_N1_task outCtx_N1_task;

  /* initialize context fields for channels */
  (&outCtx_N1_task)->N1_task_out_ch = &N1_task_out_ch;
  (&outCtx_N1_task)->N1_task_in_ch = &N1_task_in_ch;

  N1_task_reset((&outCtx_N1_task));

  for(i=0; i < NB_STEPS; i++) {
    /* call N1_task */
    while (!N1_task_in_ch_ready) {
      wait_apic_it_and_clear(PES_IT_NUM);
    }
    N1_task_in_ch_ready = 0;
    __kvx_builtin_fence();
    N1_task((&outCtx_N1_task));
    N1_task_out_ch_ready = 1;
    __kvx_builtin_fence();
    mailbox_notify(SYNC_CL0_CO0_RX_ID, __kvx_get_cpu_id());
  }
  eot_exit_pe();
  return NULL;
}
```

Listing 8.6: *Event-triggered* code example

The *time-triggered* mode uses the hardware timers described in Section 3.2.6 to precisely start each task of the application at their computed release date. These release dates and the general schedulability are the responsibility of MIA. Typical synchronization mechanisms are just used at the initialization to ensure that all PEs and Clusters have the same start of time reference. No additional data tokens are required in this mode. An example of the *time-triggered* code for a task is given in Listing 8.6.

```
char stack_PE1[PE_STACK_SIZE] __attribute__((section(".data_bank1"), aligned(8)));
void *thread_PE1(void *arg)
{
  size_t i;
  outC_N1_task outCtx_N1_task;
  uint64_t origin_of_time_PE1;

  /* initialize context fields for channels */
  (&outCtx_N1_task)->N1_task_out_ch = &N1_task_out_ch;
  (&outCtx_N1_task)->N1_task_in_ch = &N1_task_in_ch;
```

```
  N1_task_reset((&outCtx_N1_task));

  __kvx_timer64_setup(UINT64_MAX, UINT64_MAX, 0, TIMER_0);
  wait_apic_it_and_clear(RM_PES_IT_NUM);

  origin_of_time_PE1 = __kvx_timer64_get_value(TIMER_0);

  for(i=0; i < NB_STEPS; i++) {
    /* call N1_task */
    while(origin_of_time_PE1 - __kvx_timer64_get_value(TIMER_0) <= i *
        reaction_period + rel_N1_task)
      ;
    N1_task((&outCtx_N1_task));
  }
  eot_exit_pe();
  return NULL;
}
```

Listing 8.7: *Time-triggered* code example

INITIALIZATION CODE Each target platform and execution model may have a different initialization sequence. In particular for the MPPA3 this code is executed in the Resource Manager (RM) core and it changes if we are in a *event-triggered* or *time-triggered* model, as well as if we exceed the limits of a single-cluster.

Generally we initialize the channel tokens, if any, configure the RM to wait until all PEs have finished their execution and then we configure the mailbox event system and barriers, either for synchronization (in the case of *event-triggered*) or for other clusters startup. Then finally we started each PE (worker) that has a set of tasks assigned to it and then wait for the program to finish before exiting.

An example of the initialization template and the final code can be found in Appendix B, Listings B.1 and B.2.

COMMUNICATION IMPLEMENTATION The implementation of the communication between channels of size one is a well known multicore programming problem called the bounded producer-consumer. For the MPPA3, the code that must be generated changes in the same way as for the initialization.

If we are in a *event-triggered* model, a channel reader must block itself and wait for the arrival of an event, to then verify that the data has arrived using a token. A channel writer must perform the store operation and then send this event signal to the corresponding worker.

If we are in a *time-triggered* model, this process is simplified as we are sure, through the static analysis made before execution, that once the release date for that has arrived, the data will be ready. So we can simply read or write data directly without having to worry about synchronization.

An example of the communication template and the final code within a task following the *time-triggered* model can be found in Appendix B, Listings B.3 to B.5.

8.3 SOFTWARE-HARDWARE INTERFACE FOR MULTI/MANY-CORE (SHIM)

This section introduces the SHIM model, what motivated its creation, the main hardware features that it covers and an overview of the model created for the MPPA3 architecture.

8.3.1 *SHIM main characteristics*

SHIM is an IEEE standard [84] that appeared to tighten the gap between hardware manufacturers and software developers that want to effectively use these platforms. Instead of relying exclusively on libraries or proprietary tools supplied by the manufacturers, with a SHIM model a developer could properly understand the architecture and write efficient code for it. Indeed, the main target of SHIM models are tools that automatically generate code for multiple platforms, such as the Model-Based Parallelizer [83].

The SHIM interface uses Extensible Markup Language (XML), more specifically the XML Schema (XSD), to define its structure. XSD is similar to an Unified Modeling Language (UML) class diagram but in textual format. Each architecture is defined in at least one SHIM XML file but could be extend to separate files that can contain instruction set information, power configuration or vendor extensions.

The SHIM description is feature rich, providing a variety of concepts to describe the core, cluster, memory and communication mechanisms of an architecture. In this section the main elements of the SHIM standard are introduced to provide a background knowledge for understanding how they were used to model the Kalray MPPA3 architecture in Section 8.3.2.

8.3.1.1 *ComponentSet*

The combination of processors and memory devices in a system is called *topology* in SHIM language. For many-core architectures it can also represent the cluster concept, which is a particular group of cores and memory banks that are close and due to this proximity have a stronger performance affinity. A cluster can be any combination of another cluster, processor cores and memory components. The root component of a model is always a cluster. These hardware terms have different names in SHIM terminology:

- *ComponentSet*: cluster of any level. The outermost cluster is the system boundary itself
- *MasterComponent*: Processor core, accelerator or other master devices
- *SlaveComponent*: Any type of memory

Typically a memory is divided into multiple address spaces that may be used to partition the memory system or to enforce access control privileges. An address space can also be further subdivided into multiple subspaces or blocks. These spaces are used to identify the location of a memory access (a load or a store instruction) and predict the typical performance that can be expected from this operation. The SHIM terminology defines:

- *AddressSpaceSet*: group of address spaces
- *AddressSpace*: address space contained in a given set

- *SubSpace*: a subspace of a given *AddressSpace*

The innermost component *SubSpace* is tied to a memory device, i. e. *SlaveComponent* through an object called *MasterSlaveBinding*. This object also indicates which *MasterComponent* has access to the memory. Using this scheme it is possible to compartmentalize the spaces to particular cores or accelerators.

8.3.1.2 *CommunicationSet*

In order to have an useful software running on multiple cores, it must share data between these cores through some interface and also use a synchronization or trigger mechanism to coordinate the parallel execution. The SHIM interface provides a class of objects to represent this called *CommunicationSet*. Each child in this set contains a *ConnectionSet* that includes one or more *Connection* describing the source and target *MasterComponents* that are involved in this type of communication. The SHIM terminology provides multiple classes of connection:

- *SharedRegisterCommunication*: Set of registers that can be accessed by multiple cores
- *SharedMemoryCommunication*: Memory space that can be accessed by multiple cores but with particular operations (Test and Set, Compare and Exchange, ...)
- *EventCommunication*: Register used to raise event signals to other cores based on specific bitmap values
- *FIFOCommunication*: Buffers of different sizes that are used for inter processor communication
- *InterruptCommunication*: interrupt mechanism between processors to invoke an interrupt handler

8.3.1.3 *ContentionGroupSet*

The advantage of using a multi or many-core platform relies on the fact that the computation of a given program can be distributed throughout different cores. After the computation has finished on each core, they usually must communicate to exchange results or receive new inputs. These data transfers are possible through several resources such as busses, crossbars, Network-On-Chips (NoCs), Direct Memory Access (DMA), ...

A communication mechanism has a bandwidth that is limited and shared between all the initiators. Therefore if multiple components use a communication resource at the same time, and if they reach the maximal bandwidth available, they will create a contention and mutually slow them down.

The *ContentionGroupSet* provides a way to define the shared communication resources in a platform through *ContentionGroup* objects. These groups only model the resource utilization and the maximum bandwidth. They do not provide a way to specify more complex aspects such as buffers, protocols or arbiters. The SHIM model in this sense is targeted for general performance analysis rather than fine grain timing estimation, in contrast to the modeling presented in Chapter 7.

The SHIM terminology involved inside a *ContentionGroup* follows:

- *PerformanceSetRef*: list all the accesses that make use of the resource in contention

- *PerformanceSet*: describes the performance expected by each access in the defined contention scenario
- *DataRate*: defines the maximum bandwidth in data per second
- *Throughput*: defines the maximum bandwidth in data per cycle (in a given frequency)

8.3.1.4 *Other sets and terminology*

Some other concepts that are modeled by SHIM but are less relevant for the scope of this thesis are contained in this section. The *FrequencyAndVoltageSet* contains the specification on clock frequencies, voltage domains and operating points which are used for performance estimation. The complete instruction set of a core can be defined in SHIM through a LLVM IR with the number of cycles taken in average by an instruction. *PowerConfiguration* is used to define the expected power consumption within a given *OperatingPoint*.

Finally, if the SHIM standard does not provide a way to properly represent details about a given architecture, it can be extended. These vendor extensions should be defined in separate files and provide a way for the standard to remain concise while still allowing expansions if required.

8.3.2 *MPPA3 SHIM Model*

A SHIM model of the MPPA3 processor is available at Github [4]. Due to the file size (more than 45 thousand lines) it is not included in its totality here. This model can be used for automatic code generation through software such as eSOL eMBP [83] or faithfully represent the target hardware in SCADE Architect [95]. The advantage of having a comprehensive hardware model for code generation relies on being able to understand the architecture characteristics in order to produce code that will actually benefit from them, instead of a generic software that will not fully explore its capabilities. This section describes how the elements that were presented until now were used to represent the concrete platform, first at Cluster level and then at System On Chip (SoC) level.

8.3.2.1 *Cluster Model*

At the cluster level, the *ComponentSet* was used to delimitate each one of the clusters of the architecture. Inside of it, each RM or PE is a *MasterComponent*, attached to several references: *Cache*, *frequencyDomain*, *voltageDomain* and *operatingPoint*. For each of these components we also define the architecture they belong to, and their endianness.

The *MasterComponent* also hosts the information about the types of possible memory accesses and the *CommonInstructionSet* where the individual processor instructions can be benchmarked through the LLVM compiler. After all the cores have been defined the cluster *ComponentSet* contains *SlaveComponents*, in our case the local scratchpad memory. And then finally all the caches (data and instruction) that have been assigned to the cores, with their respective set of characteristics: size, number of ways, line size, replacement policy, $\cdots$.

4 Kalray MPPA3 SHIM model repository: `https://github.com/kalray/shim-model/`

Listing 8.8 shows a portion of the elements contained in the model of Cluster 0 of the MPPA3 with some information omitted or simplified to facilitate the reading of the code snippet.

```
<ComponentSet name="Cluster_00" id="cc_00">
  <MasterComponent name="RM_C00" id="rm_cc_00" frequencyDomainRef="fqr" voltageDomainRef="vdr" operatingPointRef="opr"
      arch="KV3" archOption="V1" endian="LITTLE" masterType="PU" nThread="1" pid="3.1">
    <CacheRef>DCache_RM_C00</CacheRef>
    <CacheRef>ICache_RM_C00</CacheRef>
    <AccessTypeSet>
      <AccessType name="AT_B" id="AT_b_id" rwType="RWX" accessByteSize="1" alignmentByteSize="1"/>
      <AccessType name="AT_H" id="AT_h_id" rwType="RWX" accessByteSize="2" alignmentByteSize="1"/>
      <AccessType name="AT_W" id="AT_w_id" rwType="RWX" accessByteSize="4" alignmentByteSize="1"/>
      <AccessType name="AT_D" id="AT_d_id" rwType="RWX" accessByteSize="8" alignmentByteSize="1"/>
      <AccessType name="AT_Q" id="AT_q_id" rwType="RWX" accessByteSize="16" alignmentByteSize="1"/>
      <AccessType name="AT_O" id="AT_o_id" rwType="RWX" accessByteSize="32" alignmentByteSize="1"/>
    </AccessTypeSet>
    <CommonInstructionSet name="LLVM Instructions">
    ...
    </CommonInstructionSet>
  </MasterComponent>
  <MasterComponent name="PE00_C00" id="pe00_cc_00" frequencyDomainRef="fqr" voltageDomainRef="vdr" operatingPointRef="
      opr" arch="KV3" archOption="V1" endian="LITTLE" masterType="PU" nThread="1" pid="3.2">
  ...
  </MasterComponent>
  ...
  <SlaveComponent name="SMEM_00" id="smem_00" size="4" sizeUnit="MiB" rwType="RWX"/>
  <Cache name="DCache_RM_C00" id="DCache_RM_C00" cacheType="DATA" cacheCoherency="HARDWARE" size="16" sizeUnit="KiB"
      nWay="4" lineSize="64" lockDownType="LINE" prefetch="ONMISS" replacement="LRU" writeAllocate="NEVER" writeBack="
      NEVER"/>
  <Cache name="ICache_RM_C00" id="ICache_RM_C00" cacheType="INSTRUCTION" cacheCoherency="HARDWARE" size="16" sizeUnit="
      KiB" nWay="4" lineSize="64" lockDownType="LINE" prefetch="ALWAYS" prefetchDistance="128" replacement="LRU"
      writeAllocate="NEVER" writeBack="NEVER"/>
  ...
  <Cache name="DCache_PE15_C00" id="DCache_PE15_C00" cacheType="DATA" cacheCoherency="HARDWARE" size="16" sizeUnit="KiB"
      nWay="4" lineSize="64" lockDownType="LINE" prefetch="ONMISS" replacement="LRU" writeAllocate="NEVER" writeBack=
      "NEVER"/>
  <Cache name="ICache_PE15_C00" id="ICache_PE15_C00" cacheType="INSTRUCTION" cacheCoherency="HARDWARE" size="16"
      sizeUnit="KiB" nWay="4" lineSize="64" lockDownType="LINE" prefetch="ALWAYS" prefetchDistance="128" replacement="
      LRU" writeAllocate="NEVER" writeBack="NEVER"/>
</ComponentSet>
```

Listing 8.8: Portion of the MPPA3 SHIM Model of Cluster 0

8.3.2.2 *SoC Model*

At the SoC level, we have the regroupment of all the clusters represented in their *ComponentSets*, as well as the global memory in the form of a *SlaveComponent*. Then the *AddressSpaceSet* defines the possible address spaces for the different *ComponentSets* or *MasterComponents*. A possible connection is established through a *MasterSlaveBinding* with a respective performance associated, providing the memory latency for that access.

Some of the possible *CommunicationSets* in the MPPA3 are represented, through shared memory regions and mailboxes, but this was not modeled exhaustively, as it was out of the scope of this SHIM model goal. The *ContentionGroupSet* is what gets the closest to the interference models found in MIA, though it is merely a throughput and data rate indication, without any worst-case information. Finally the SoC model ends with non-functional information through the *FrequencyVoltageSet* and *PowerConfiguration*.

Listing 8.9 shows a portion of the elements contained in the model of the complete MPPA3 SoC with, once again, some information omitted or simplified to facilitate the reading of the code snippet.

```
<?xml version="1.0" encoding="UTF-8"?>
<shim:Shim xmlns:shim="http://www.multicore-association.org/2017/SHIM2.0/" name="KV3 (MPPA3, Coolidge) SHIM Architecture
    Description" shimVersion="2.0">
  <SystemConfiguration>
    <ComponentSet name="Clusters" id="clusters">
      <ComponentSet name="Cluster_00" id="cc_00">
      ...
```

```
    </ComponentSet>
    <ComponentSet name="Cluster_01" id="cc_01">
    ...
    </ComponentSet>
    <ComponentSet name="Cluster_02" id="cc_02">
    ...
    </ComponentSet>
    <ComponentSet name="Cluster_03" id="cc_03">
    ...
    </ComponentSet>
    <ComponentSet name="Cluster_04" id="cc_04">
    ...
    </ComponentSet>
    <SlaveComponent name="DDR" id="ddr" size="4" sizeUnit="GiB" rwType="RWX"/>
  </ComponentSet>
  <AddressSpaceSet>
    <AddressSpace name="AS_Clusters" id="AS_Clusters">
      <SubSpace name="SS_INTERNAL_MEM" id="SS_INTERNAL_MEM" start="0" end="4194303" endian="LITTLE">
        <MasterSlaveBindingSet>
          <MasterSlaveBinding slaveComponentRef="smem00"/>
          <MasterSlaveBinding slaveComponentRef="smem01"/>
          <MasterSlaveBinding slaveComponentRef="smem02"/>
          <MasterSlaveBinding slaveComponentRef="smem03"/>
          <MasterSlaveBinding slaveComponentRef="smem04"/>
        </MasterSlaveBindingSet>
      </SubSpace>
      <SubSpace name="SS_SMEM_0" id="ss_smem00" start="16777216" end="20971519" endian="LITTLE">
        <MasterSlaveBindingSet>
          <MasterSlaveBinding slaveComponentRef="smem00">
            <Accessor masterComponentRef="rm_c00">
              <PerformanceSet id="perf_rm_c00_smem00">
                <Performance accessTypeRef="AT_b_id">
                  <Pitch best="3.0" typical="23.0" worst="368.0"/>
                  <Latency best="3.0" typical="23.0" worst="368.0"/>
                </Performance>
                ...
              </PerformanceSet>
            </Accessor>
          </MasterSlaveBinding>
        </MasterSlaveBindingSet>
      </SubSpace>
      <SubSpace name="SS_SMEM_1" id="ss_smem01" start="33554432" end="37748735" endian="LITTLE">
      ...
      </SubSpace>
      <SubSpace name="SS_SMEM_2" id="ss_smem02" start="50331648" end="54525951" endian="LITTLE">
      ...
      </SubSpace>
      <SubSpace name="SS_SMEM_3" id="ss_smem03" start="67108864" end="71303167" endian="LITTLE">
      ...
      </SubSpace>
      <SubSpace name="SS_SMEM_4" id="ss_smem04" start="83886080" end="88080383" endian="LITTLE">
      ...
      </SubSpace>
    </AddressSpace>
    <AddressSpace name="AS_Global" id="AS_Global">
      <SubSpace name="SS_DDR" id="ss_ddr" start="4294967296" end="8589934591" endian="LITTLE">
      ...
      </SubSpace>
    </AddressSpace>
  </AddressSpaceSet>
  <CommunicationSet>
    <SharedMemoryCommunication name="SHMEM_C00" operationType="OTHER" dataSize="4" dataSizeUnit="MiB" addressSpaceRef=
        "as_clusters" subSpaceRef="ss_smem_00">
      <ConnectionSet>
        <Connection from="rm_c00" to="pe00_cc_00"/>
      </ConnectionSet>
    </SharedMemoryCommunication>
    ...
    <EventCommunication name="Event_APIC_Mailboxes">
      <ConnectionSet>
        <Connection from="rm_c02" to="pe02_c03"/>
      </ConnectionSet>
    </EventCommunication>
    <InterruptCommunication name="Interrupt_IPI_C00">
      <ConnectionSet>
        <Connection from="rm_c00" to="pe00_cc_00"/>
      </ConnectionSet>
    </InterruptCommunication>
  </CommunicationSet>
  <FrequencyVoltageSet>
    <FrequencyDomain name="FrequencyDomain_Clusters" id="fdr"/>
    <VoltageDomain name="VoltageDomain_Clusters" id="vdc"/>
    <OperatingPointSet name="OperatingPointSet0" id="op0">
      <OperatingPoint name="Normal" id="op0" frequency="800" frequencyUnit="MHz" voltage="800" voltageUnit="mV"/>
      <OperatingPoint name="Turbo" id="op1" frequency="1000" frequencyUnit="MHz" voltage="880" voltageUnit="mV"/>
    </OperatingPointSet>
  </FrequencyVoltageSet>
  <ContentionGroupSet>
    <ContentionGroup name="ContentionGroup_SMEM00" id="cg_smem00">
      <Throughput frequencyDomainRef="fdr" value="32" unit="B/cycle"/>
      <DataRate value="32" unit="GiB/s"/>
```

```
        <PerformanceSetRef>perf_rm_coo_smemoo</PerformanceSetRef>
        ...
      </ContentionGroup>
    </ContentionGroupSet>
  </SystemConfiguration>
  <PowerConfiguration>
    <PowerConsumptionSet name="PowerConsumptionSet_1" id="pcs">
      <PowerConsumption operatingPointRef="op0" power="10" powerUnit="W"/>
      <PowerConsumption operatingPointRef="op1" power="20" powerUnit="W"/>
    </PowerConsumptionSet>
  </PowerConfiguration>
  <FunctionalUnitSet>
    <FunctionalUnit name="BCU"/>
    <FunctionalUnit name="LSU"/>
    <FunctionalUnit name="ALU"/>
    <FunctionalUnit name="MAU"/>
    <FunctionalUnit name="TCA"/>
  </FunctionalUnitSet>
</shim:Shim>
```

Listing 8.9: Portion of the MPPA3 SHIM Model of the complete SoC

8.3.2.3 *Limitations*

The model presented here does not aim to be exhaustive in terms of representing all the components contained in the Kalray MPPA3 SoC. Even at the cluster level, we chose not to include the Cryptographic Accelerators, DSU and DMA components (see Figure 3.3), for example, as they are not relevant for the scope of this work. On top of this, even if the SHIM standard aims to be generic, it does not provide a lot of features to represent such components that distance themselves from regular cores.

As seen in Section 8.3.2.2, the only feature provided by the SHIM standard to represent multiple *MasterComponents* accessing the same shared resource is through *ContentionGroups*. However, it can only be specified the throughput in terms of Bytes per cycle and the data rate in terms if Gigabytes per second. With this limitation and without any vendor extension, the base SHIM standard is not adapted for hard real-time systems, were precise information about the expected delays at runtime should be obtainable statically prior to execution.

8.4 CONCLUSION

Contributions made to improve, extend or develop new models for existing tools and standards have been presented in this chapter.

The Multi-Core Interference Analysis (MIA) tool has been rewritten with a new algorithm that simplifies the response time analysis of tasks and avoids any fixed point iteration, reducing the complexity from $O(n^4)$, in the old version, to $O(n^2)$. This is possible with the simple idea of having a time cursor that only consider the interference delay happening between tasks that are being executed at a given instant. Further improvement could be obtained by also taking into account the overlap time among the task that are alive. This will certainly lead to less pessimistic results than including the whole interval until the next release or finished date. A better deadline management system could also be added to return individual node schedulability information.

The SCADE Multi-Core Code Generator (MCG) has been analyzed in depth in this chapter. In particular it was detailed the platform dependent parts that needs to be adapted in order to target a new platform after generating the functional code from SCADE. The implementation choices for the different execution modes targeting the Kalray MPPA3 were described. Future work can include additional execution modes or trying to improve

the existing ones. For example, instead of an active timer wait in the time-triggered mode, the core could be put to sleep and be woken up by a timer interrupt.

The Software-Hardware Interface for Multi-Many-Core (SHIM) is an IEEE standard that tries to close the gap between hardware models and software that require such models to automatically produce code or perform analyses. The MPPA3 SHIM model that was presented in this chapter contains the essential information to understand and integrate the platform in any software that requires SHIM. Improvements can be done to include more components from the platform and also complete the *CommunicationSet* and *ContentionGroupSet* elements.

9

EXPERIMENTS

This chapter hosts the experiments that put in practice the concepts developed throughout the contributions seen in Part II, in particular the phased execution models from Chapter 5, the mapping and scheduling algorithm from Chapter 6 and the Multi-Core Interference Analysis (MIA) algorithm rework from Section 8.1.

The applications used to evaluate the phased execution models are presented at the beginning through a brief description, graphical visualization and profiling information. For this evaluation we stayed within the cluster boundaries, so all of them fit the local memory limits. Nonetheless, some of them are case-studies or industrial applications and it is important to establish their characteristics, in order to discuss the results further into the chapter.

The first experiments are then performed by applying the memory partitioning and interference models presented before. A quick recap is given before diving into how the evaluation was performed in two target platforms: the Kalray MPPA2 and MPPA3. The results are presented raising a broader discussion targeted for the real-time systems community. This is followed by the experiments used to evaluate the solutions to the Directed Acyclic Graph (DAG) mapping and scheduling problem. As we are exploring the multi-cluster feature of the MPPA3, instead of using handmade applications, we generate random DAGs with tens to hundreds of nodes. This allowed a proper exploration of the proposed algorithm and the platform. The last evaluation is done on the new algorithm for response time analysis implemented in MIA. Randomly generated DAGs with the same arbitration model were used to provide a fair point of comparison with the old version.

From the contributions presented in this thesis, Chapter 7 is the only one that is not evaluated here. This is justified by the fact that the arbitration model implementation

is a direct translation of the mathematical models. However, this implementation is the foundation for all the experiments performed in this chapter where a MPPA3 model is required.

9.1 APPLICATIONS PRESENTATION

The applications used for the experiments in Section 9.2 are presented here, giving emphasis on their structure, resource usage and access availability. An important common point between these applications is that all the code and data fit in a cluster local memory, avoiding the need for global memory accesses during the execution. An initial mapping of application tasks to cores is given for each example as the goal is to evaluate only the phased execution models, scheduling and memory interference impact.

9.1.1 *Simple Data Flow*

This example application[1] is the one used in Chapter 5, more specifically in Example 1 to illustrate the problems with single-phased execution and how having multiple phases can help to solve them. During the implementation this application was used to validate our methodology in terms of scheduling and implementation. As shown in Figure 9.1 the 5 task nodes are initially mapped to 2 cores, with N_0 and N_3 having no initial dependencies, while the other nodes are connected to the data-flow that derives from them. The execute tasks perform basic arithmetic operations.

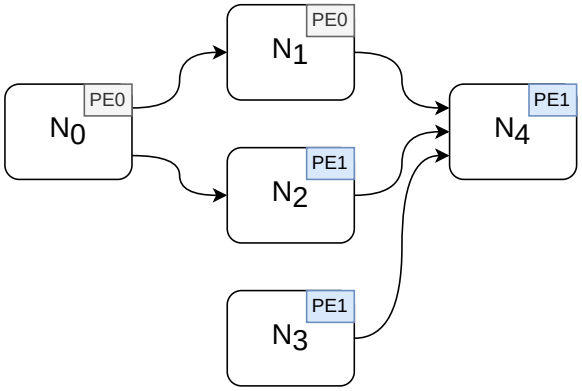


Figure 9.1: Simple data-flow graph and preliminary mapping

Profiling information of this application for the 3-Phased partitioning model is:

- Compute phases — WCET: from 260 to 326 cycles, WCA: from 2 to 7 accesses;
- Read phases — WCET: from 183 to 198 cycles, WCA: from 0 to 3 accesses;
- Write phases — WCET: from 183 to 202 cycles, WCA: from 0 to 3 accesses;
- Computation-Communication Ratio (CCR) — 1 732 compute cycles to 3 041 communication cycles.

1 Available at `https://gricad-gitlab.univ-grenoble-alpes.fr/verimag/reproducible-research/simple-dflow-rtss-2020/`

9.1.2 *Avionics Case Study*

ROSACE is an avionics open-source case-study developed by ONERA [106][2]. It contains a multi-periodic flight controller program that aims to be easily executed on a multi/many-core processor. The original code has been expanded into a hyper-period that normalizes the multi-periodic nature of the program. The application contains 10 nodes (execute tasks) and is initially mapped to 8 cores. The corresponding data-flow graph is given in Figure 9.2.

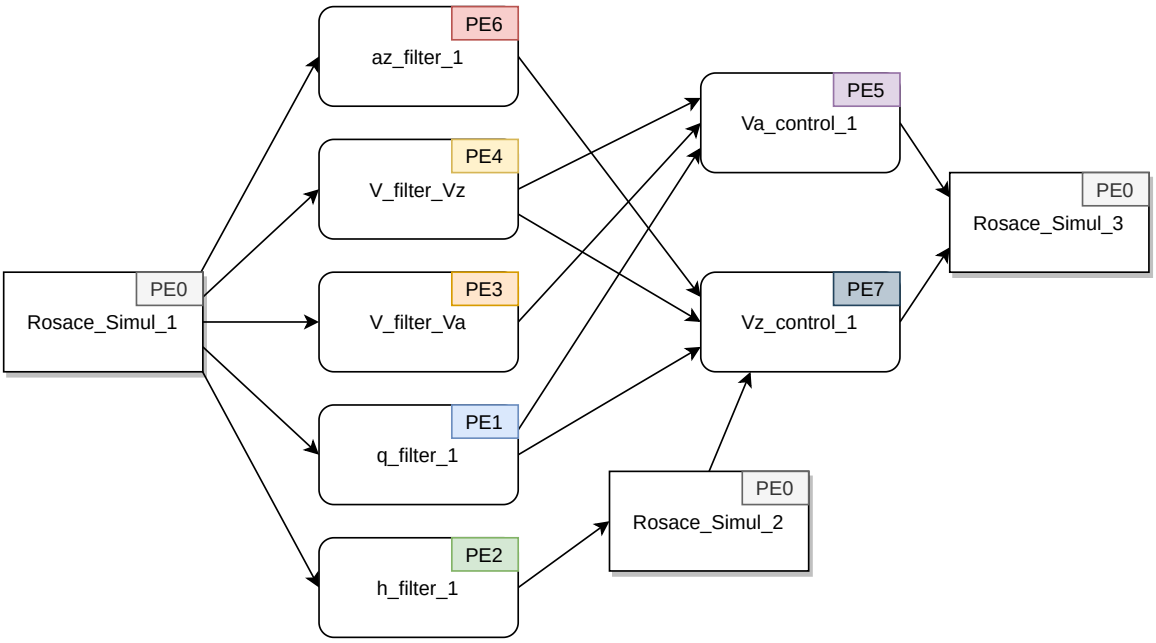


Figure 9.2: ROSACE data-flow graph and preliminary mapping

Profiling information of this application for the 3-Phased partitioning model is:

- Compute phases — WCET: from 624 to 74 343 741 cycles, WCA: from 7 to 195 accesses;
- Read phases — WCET: from 170 to 185 cycles, WCA: from 1 to 2 accesses;
- Write phases — WCET: from 170 to 187 cycles, WCA: from 1 to 3 accesses;
- CCR — 260 214 753 compute cycles to 5 972 communication cycles.

The ROSACE case-study, in comparison with the simple data flow application, has bigger compute phases in terms of WCET, as well as a CCR revealing that the application is more compute intensive.

9.1.3 *Automotive Industrial Program*

The third case-study is an industrial automotive program with 4 765 lines of functional code (not including the orchestration code). The 9 nodes are initially mapped to 6 cores. There is one initial task and one final task, the other 7 tasks depend only on the data

2 Available at `https://forge.onera.fr/projects/rosace-case-study`

produced by the initial task. This is a common structure found in highly parallel periodic applications.

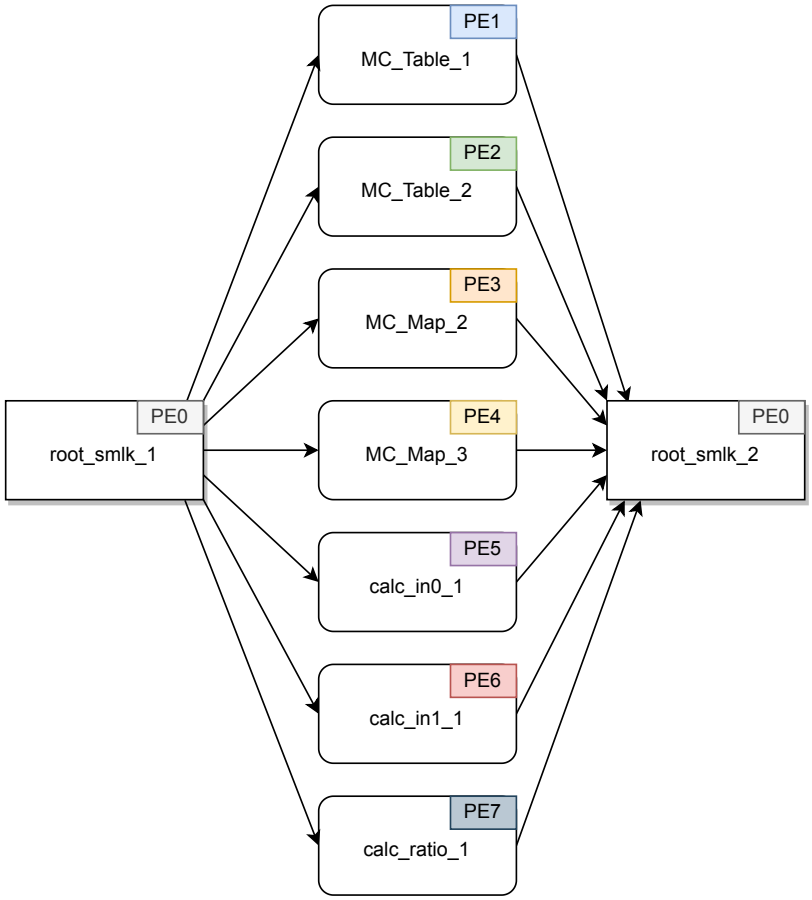


Figure 9.3: Automotive industrial data-flow graph and preliminary mapping

Profiling information of this application for the 3-Phased partitioning model is:

- Compute phases — WCET: from 465 to 2 653 cycles, WCA: from 10 to 68 accesses;
- Read phases — WCET: from 187 to 202 cycles, WCA: from 1 to 3 accesses;
- Write phases — WCET: from 187 to 202 cycles, WCA: from 1 to 3 accesses;
- CCR — 12 548 compute cycles to 5 416 communication cycles.

This automotive application sits in between the simple data-flow and the ROSACE case study in terms of WCET and CCR, being a good candidate to evaluate balanced programs, more evenly distributed between computation and communication.

9.2 PHASED EXECUTION MODELS EXPERIMENTS

Chapter 5 introduced the different phased execution models that were implemented and applied to the programs seen in Section 9.1. A brief recall of the different models:

- 2-Phased model with Execute and Write phases. A local partition for each core allowing write from other tasks in this private partition

- 3-Phased model with Read, Execute and Write phases. A local partition for each core and one global shared partition
- Memory-Centric 3-Phased model with Read, Execute and Write phases but with a dedicated core orchestrating the memory phases

Moreover, regarding the memory interference model, we either enforce isolation during memory phases, or we allow interference and estimate the delay values.

9.2.1 *Evaluation context*

We evaluate our implementation of the execution models proposed in Chapter 5 with the applications presented in Sections 9.1.1 to 9.1.3:

1. The simple data-flow example;
2. The ROSACE avionics case-study;
3. An industrial automotive Electronic Control Unit (ECU) program.

The starting programming language of our implementation is SCADE [40], which is widely used in the industrial context. Our goal with these applications is to evaluate the final schedule response time with real world scenarios. In particular, we are interested in comparing the possible timing improvements when taking interference into account against safe execution models that provide temporal isolation.

The implementation of the code generation steps of the workflow described in Chapter 4 was done in Python with the Mako template library[3] for the platform dependent parts, as explained in Section 8.2.3.2. A bash script then ties together all the necessary steps of the workflow. The Worst-Case Execution Time (WCET) and Worst-Case Number of Accesses (WCA) of the execute and memory phases were measured after multiple runs on the processor, due to the absence of access to a formal analysis of the Kalray MPPA3. Timing analyzers such as OTAWA [17], Heptane [78] or aiT [56] must be used, instead of measurement techniques, if strong timing guarantee is required.

For the interference estimation, it is important to know that we target and compare two platforms in these experiments: the Kalray MPPA2 and MPPA3 processors. The goal of this comparison is to see if the same result tendencies are seen in both platforms. We do not aim to compare the performance or global response time of the applications in the different processors.

An extensive analysis of the arbitration system of the MPPA2 is provided in [118]. A quick summary follows: a blocking memory access requires 10 cycles to be completed and the multi-level arbitration system makes the maximal cost to be:

$$\sum_{1 \leq i \leq 15} (\min(I(P_0), I(P_i))) + \min((Y), (X)) + (R_x) \tag{9.1}$$

where I gives the interference between specific cores, Y is the interference generated by any core, X is the interference between other initiators (T_x, RM, DSU) and R_x is the high priority initiator at the third and final level.

3 Documentation available at `https://docs.makotemplates.org/`

The complete analysis of the MPPA3 arbitration system is provided in Chapter 7. To summarize, a blocking memory access on the MPPA3 requires 22 cycles to be completed and the intra-cluster arbitration level from Equation (7.6) makes the maximal cost to be:

$$\sum_{1 \leq i \leq 15} (\min(I(P_0), I(P_i))) + \min(AXI_{write}, Y) + \min(AXI_{read}, Y) \tag{9.2}$$

where I gives the interference between specific cores, Y is the interference generated by the core in analysis own accesses and AXI_{write}/AXI_{read} is the interference generated by accesses through the AXI communication bus.

Section 8.1 contains more details about how these interference cost functions are used inside the complete algorithm to obtain the Worst-Case Response Time (WCRT) of the complete program.

The offline scheduling algorithms presented in Section 5.3 have no significant runtime overhead in comparison with the time spent executing the whole workflow in the experiments conducted. For reference, the runtime ranges from 239 ms to 525 ms in the explored programs, while the complete workflow execution can take from 46 s to 1 m 17 s. In this section we will focus exclusively on the global WCRT computed by MIA and measured on the MPPA2/MPPA3, for each proposed execution model and application. The algorithms' runtimes show that their computational cost is reasonable, appropriate for our scenario and does not significantly impact the performance of the whole workflow.

9.2.2 *Results*

Experimental results from the MPPA2 arbitration models are displayed in Figures 9.4, 9.6 and 9.8 and from the MPPA3 in Figures 9.5, 9.7 and 9.9. For each case-study and each implementation we give the estimated *WCRT* computed by MIA for the code generated by our workflow and the corresponding *Measured* execution time (both in nanoseconds as we compare two processors with different clock frequencies). For the *2-Phased* and *3-Phased* implementation we give the results with *Interference* and in all cases the results for the implementation in *Isolation* are given. Remember that for the *Memory-Centric* implementation there is no interference between memory phases as they are all scheduled on the same core. Furthermore, we give the results using both isolated 3-Phased scheduling algorithms: *Cont* for the contiguous implementation of the three phases (see Algorithm 1) and *Opt* for the optimized version with potential idle time slots (see Algorithm 2).

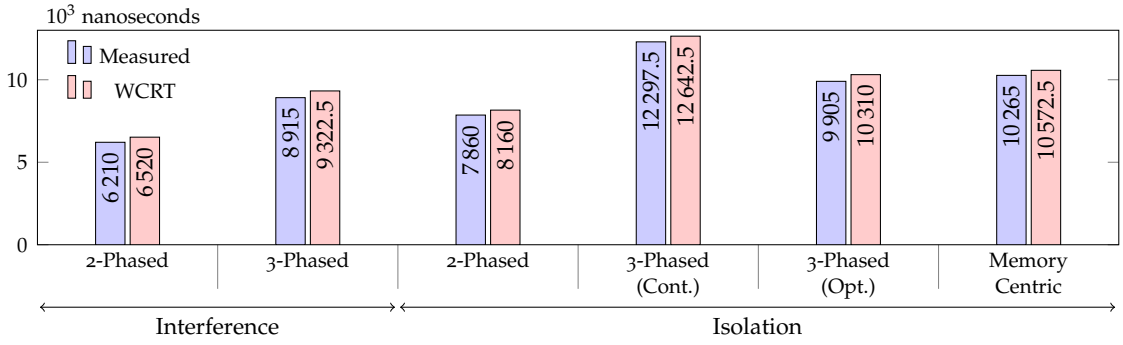


Figure 9.4: Simple Data-Flow **MPPA2**

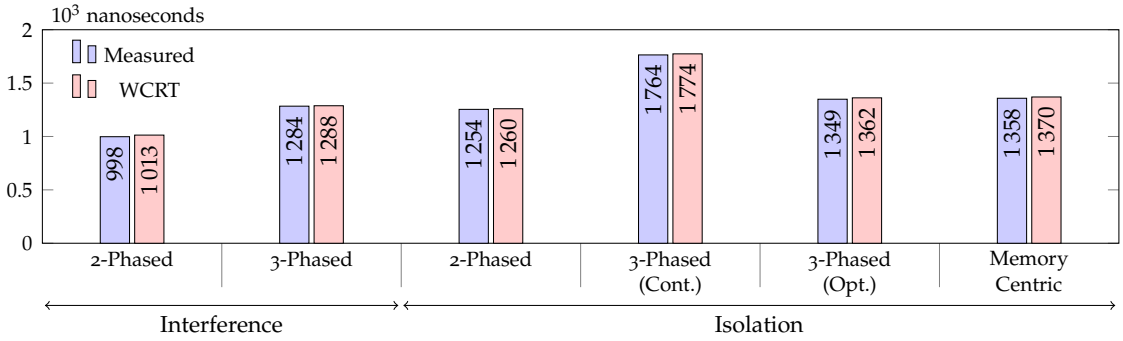


Figure 9.5: Simple Data-Flow **MPPA3**

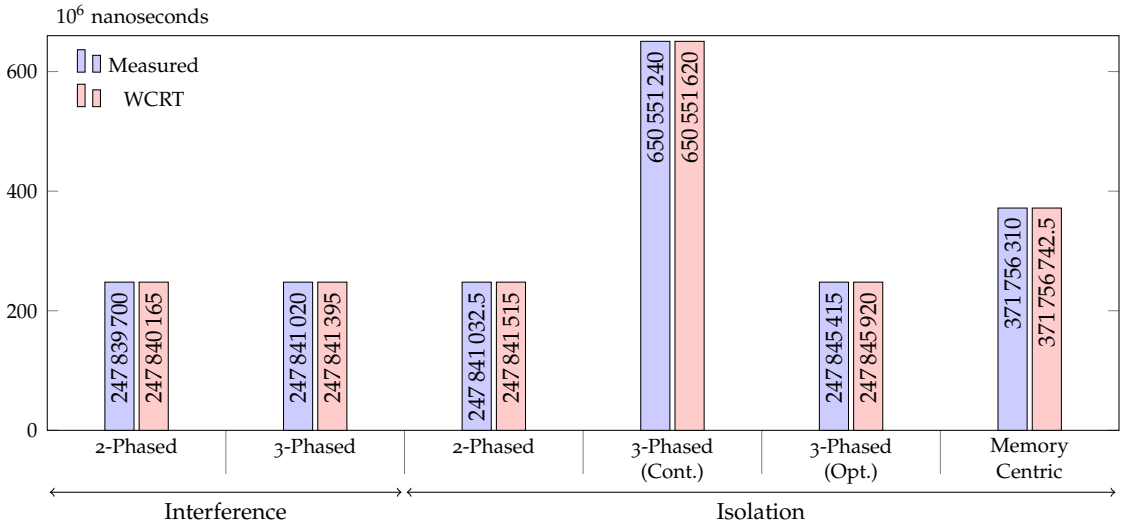


Figure 9.6: ROSACE **MPPA2**

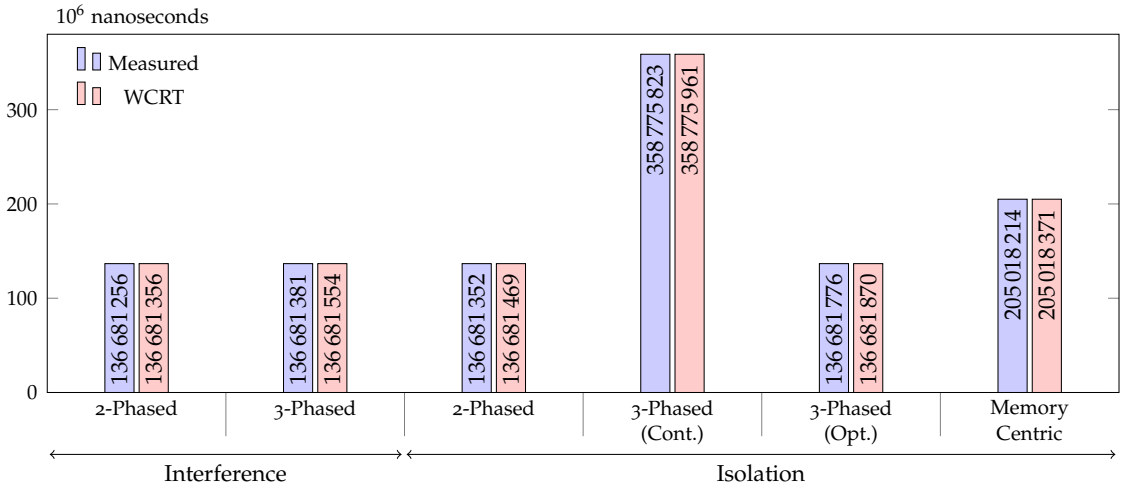


Figure 9.7: ROSACE **MPPA3**

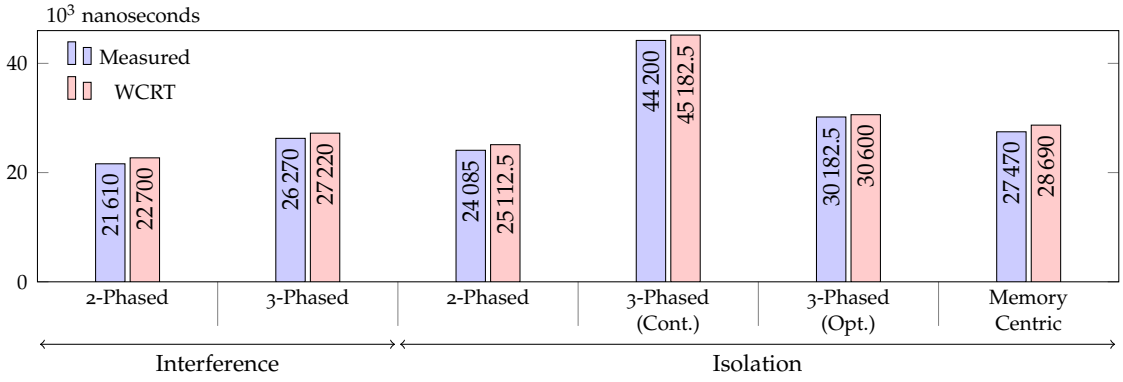


Figure 9.8: Automotive **MPPA2**

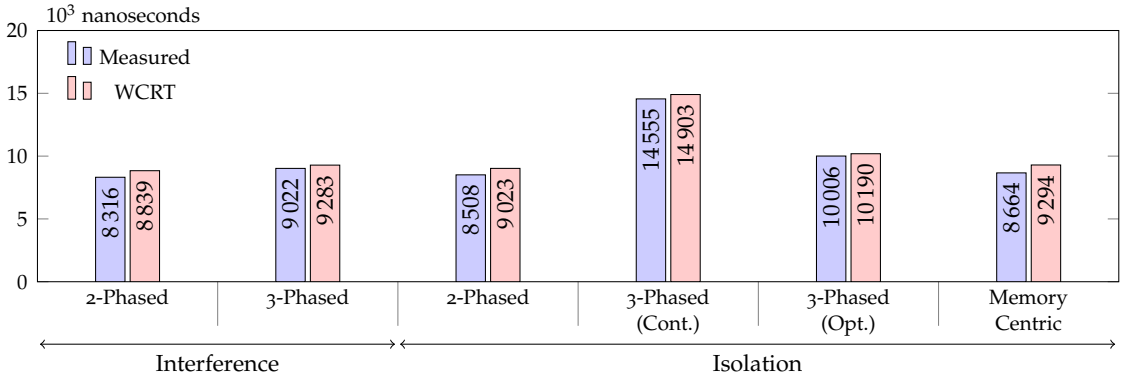


Figure 9.9: Automotive **MPPA3**

We observe that the WCRT is always close to the measured response time, with a maximum difference of 1220 ns in the automotive MPPA2 Memory Centric experiment. This is due to the fact that we use a time-triggered implementation: the difference may only come from the divergence between the WCET and the actual execution time of the last task, as the release date of the last task is identical to the one computed by MIA. Note that in case of interference, the release date of the last task is not the same as in the case of isolation. Furthermore, the difference between the measured and the bound on the WCRT may be larger due to potential interference taken into account during the execution of the last task.

From the *Interference* and *Isolation* WCRT values in all 6 figures we observe that **taking into account delays due to interference leads to shorter WCRT than isolating the memory phases.** This is attributable to the memory partitioning model limiting the interference during memory phases and also to the structure of the MPPA2/MPPA3 processor bus arbiter. The size of shared data is also small for all case-studies, usually less than the bus size: in this case a potential interference between two cores is accounted as only one additional cycle as it is simply a one-cycle wait for the round-robin arbiter. For the simple data-flow case-study, the additional cost of interference is of 2 cycles in the 2-Phased case and 3 cycles in the 3-Phased case. This happens because there are few potential interference points and only 2 cores are used. For ROSACE, in the 2-Phased implementation there is no additional cost due to interference. Note that ROSACE case-study has long execute phases and very small memory phases: this limits the probability of interference. For the 3-Phased implementation, the additional cost for ROSACE is of 2 cycles: here at least one of the additional read phases causes an interference. For the automotive case-study,

the additional cost is of 2 cycles for the 2-Phased implementation and 10 cycles for the 3-Phased one. These low interference costs explains why the WCRT with interference is shorter: the cost of isolation is much heavier than the cost of interference.

Similarly, reducing the number of phases, limits the number of interference and the number of tasks to isolate with the global scheduling algorithms: a **2-Phased implementation is always more efficient than a 3-Phased one.** Note that this is also a specificity of the multi-banked memory that allows to execute locally with a predictable shared data memory (in contrast to a shared cache memory). The difference between the 2-Phased and 3-Phased is not only due to the number of phases, but also due to the distributed shared memory among the banks vs. one shared memory bank. The shared memory bank seems to be a good idea for better isolation of shared memory access. Nevertheless, our experiments show that a distribution of the shared memory on all banks is more efficient for the applications and target processor we use. Therefore, a 2-phase model with execute-write operations is preferable when it can be applied to the program, the architecture contains a multi-banked shared memory and the number of generated memory phases can be controlled.

The Memory-Centric implementation behaves most of the time worse than the 3-Phased *Opt*. The reason for this is the mapping of all memory phases to the same core, which forbids any concurrency between the memory phases. Note that the Memory-Centric implementation of these phased execution models has been introduced mainly for external shared memory, where the memory access time is significantly longer.

Our optimized algorithm that introduces idle intervals instead of enforcing contiguous REW phases, yields the best results among the 3-Phased models, except for the automotive use case. This is a result of read phases being scheduled earlier by this algorithm and as consequence the execute phase may also start sooner. This is at a price of inserting idle times between memory transactions, but it is important to reinforce that the data depencies are still preserved. The exception on the automotive use case is a small difference (less than 1 000 cycles) and probably comes from its CCR profile. Another possible reason is a good timing in the scheduling between memory and execution phases that does not degrade the global execution time, even if the memory transaction are serialized on one core.

Comparing the MPPA2 and MPPA3 arbitration models and overall response time, the first aspect to highlight is that, independently of the target platform, the same tendencies are found for the multiple phased execution and memory interference models. This strengthens all the observations made until this point. The results were presented in nanoseconds to provide a fair comparison, as the clock frequency and memory delays changed from one generation to the other.

If the raw global response time is now analyzed, we perceive an improvement in the MPPA3, having in average 70% smaller execution time across all the variations of execution and interference models studied. The increase in clock frequency (from 400MHz to 1GHz) is responsible for this massive improvement. However, in terms of raw number of cycles, the MPPA3 shows worst results for applications that perform a lot of memory accesses, such as the ROSACE case-study. The increase in Shared MEMory (SMEM) latency (from 10 to 23 cycles) is responsible for this effect, which is hindered by the higher clock frequency.

9.2.3 *Discussion*

In this section we present a broader discussion directed to the real-time systems domain community, commenting on the results obtained during the experiment.

9.2.3.1 *Number of phases*

Among our results, the first point we highlight is the interest of the 2-Phased model. This solution is easy to implement on any processor that provides multi-banked shared memory. The question raised by this is why the use of these 2-Phased method is so scarce, even if it appears to be efficient in providing good execution time. We see here two main reasons. The code may be intrinsically 3-Phased and the 2-Phased implementation would require changes in the way the code is generated. Similarly, in the real-time community, the theory usually works with 3 phases independently of the target processor. In case of the MPPA processor, our study shows that even for SCADE code, **a 2-Phased implementation is more efficient.**

9.2.3.2 *Interference versurs Isolation*

A second point is about the interferences. We observed that **the potential of interference is quite low** in our study, so why are they generally avoided instead of analyzed? Likely, the answer is the hypothesis of a timing compositional processor[4]. With this hypothesis, the additional cost due to interference may be added with a guaranteed final estimated bound. The MPPA processor is assumed to be timing compositional [122]. Unfortunately, for this processor and all other industrial ones, there exists yet no proof of such compositionality. This leads to an important open question: is it possible to write such a proof and guarantee that the compositionality is ensured? A formal proof would be largely beneficial for certification and thus industrial use of the interference analysis.

What we observed about SCADE intrinsically phased execution is that enforcing a sequence of REW for each task may have a significant cost and it is better to introduce idle slots between phases or memory transactions to improve the global response time. Furthermore, the initial mapping and scheduling may not be optimal in some cases, as we observed for our simple program (see Figure 5.1) where node N_3 could be scheduled before node N_2 without losing any precedence constraints nor functional property.

The Memory-Centric implementation is beneficial mainly when an external memory with longer memory transactions is used. However, we included it in our study as it appears as a good solution to isolate the execution of memory phases and execute phases. Even though this seems to be a reasonable argument for predictability, our experiments shows that a good mapping of a 2-Phased or 3-Phased implementation delivers better efficiency even in isolation.

9.3 MAPPING AND SCHEDULING EXPERIMENTS

The mapping and scheduling of a task set represented by a DAG onto a multi/many-core architecture is the problem studied in Chapter 6. The evaluation performed in this section

4 No timing anomaly or with bounded effects such that any delay may be added without any loss of a guaranteed global bound.

aims to give an initial perspective on the proposed algorithm and, when possible, compare it to the classical list-scheduling from SCADE Multi-Core Code Generator (MCG).

9.3.1 *DAG generation*

In order to provide a fair base of comparison between the mapping and scheduling algorithms, instead of using the same applications described in Section 9.1, we generated random DAGs. These DAGs are issued from a method proposed in [138] called layer-by-layer DAG generation. It aims to output a task graph set for fair evaluation of multiprocessor scheduling algorithms. Figure 9.10 illustrates how a generic application may be created by varying the number of layers (NL) and the layer size (LS).

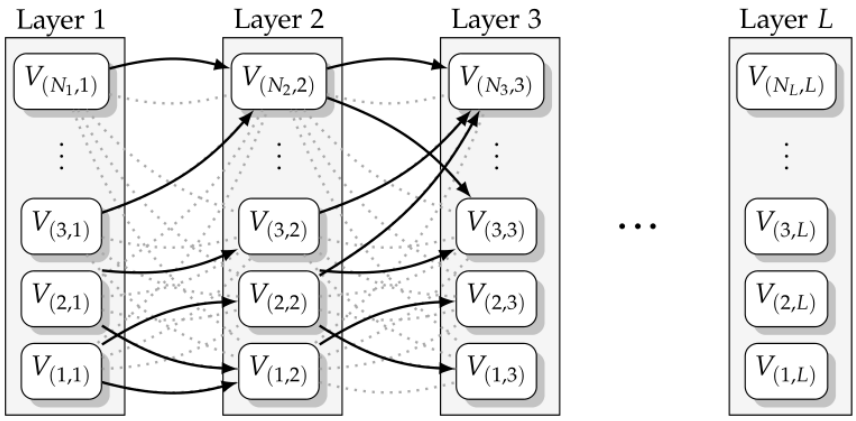


Figure 9.10: Random DAG generation method from Tobita and Kashara [138]

Tasks that are on the same layer do not depend on each other as they do not communicate. Tasks on different layers have a certain probability of communicating, defined as an input parameter. Other parameters include minimum and maximum limits for WCET, WCA, amount of accesses to successor tasks and memory use. Within these limits the final values for each task are generated randomly. To reproduce more faithfully most of the parallel programs structure, an initial and final node are added to serve as branching and sinking points.

A new random DAG generator was developed called *Tagada*. It mimics a fork-join structure and has fewer input parameters: the number of tasks, the expected WCET summing all the tasks in the graph's critical path and the maximal memory use. The other profiling information (WCA and successor accesses) are derived from these supplied parameters to reproduce real use-cases.

9.3.2 *Comparative methodology*

As explained in Section 6.4, our mapping and scheduling baseline is the list scheduling algorithm from SCADE MCG. This algorithm takes into account the WCET of the different tasks and maps and order the tasks into the core that provides the earliest start date. However, it does not take into account the memory use of the platform (256KB per memory bank in the Kalray MPPA3) and can therefore produce unfeasible results.

Our goal with the generated DAGs is to provide applications that must spawn over multiple clusters to fully exploit the MPPA3 platform and observe the inter-cluster interference and AXI communication cost. If these applications were grouped together into a single cluster, they would violate the local memory limits.

Therefore in the next section we provide smaller programs, that fit into one cluster, and bigger programs that cannot be properly allocated by the list scheduling algorithm. For these situations, we only evaluate the final response time given by our proposed algorithm and discuss the obtained mapping and scheduling leveraging possible improvement points.

9.3.3 *Results*

Four applications were generated to evaluate the list scheduling and our proof of concept algorithm. Two of them were generated with the layer-by-layer method using the following parameters:

- **s35 application**: $LS = 3$, $NL = 5$, connection probability of 20%, WCET in $[300, 2000]$, WCA in $[10, 100]$, write accesses to successors in $[1, 30]$ and memory use per task in $[10KB, 50KB]$
- **s10x10 application**: $LS = 10$, $NL = 10$, connection probability of 30%, WCET in $[300, 2000]$, WCA in $[10, 100]$, write accesses to successors in $[1, 30]$, memory use per task in $[10KB, 50KB]$

A visual representation of the s35 application is given in Figure 9.11. The amount of write accesses (communication size) can be seen on the edges connecting the nodes of the graph. The s10x10 application is not represented in a figure due to the hundreds of nodes in the program.

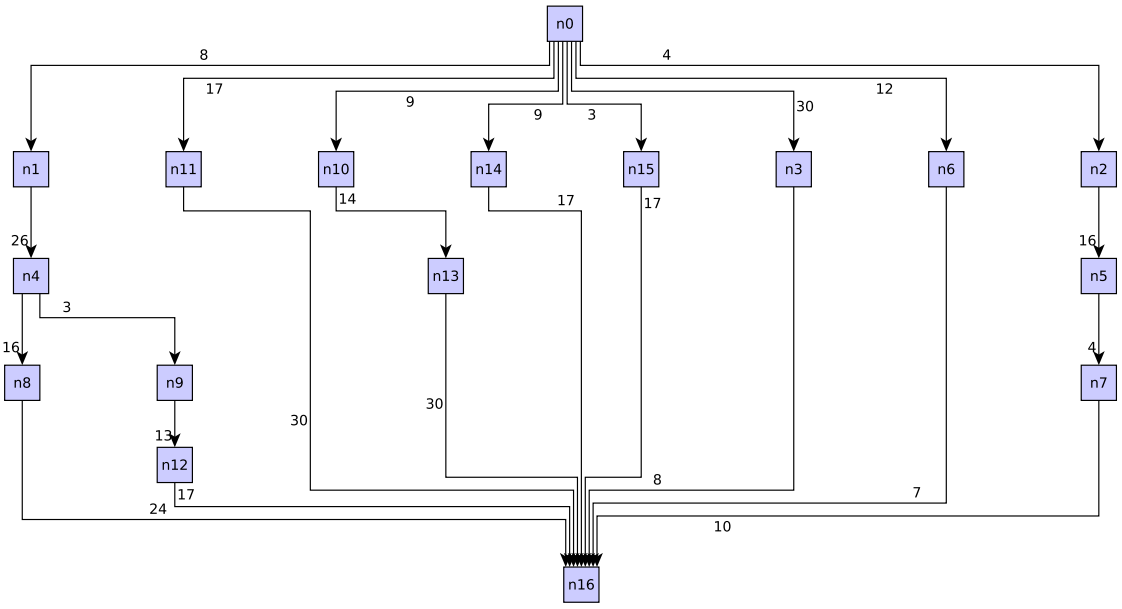


Figure 9.11: s35 application graph

The other two programs were generated with *Tagada* using these parameters:

- **tg18 application**: number of tasks = 18, critical-path WCET $\simeq$ 10 000, memory use per task $\simeq 80KB$
- **tg100 application**: number of tasks = 100, critical-path WCET $\simeq$ 100 000, memory use per task $\simeq 80KB$

A visual representation of the tg18 application is given in Figure 9.12. The tg100 application is not represented in a figure due to the high number of nodes. On top of the amount of write accesses on the edges, we can see inside each node, under its name, its WCET and memory use in KB.

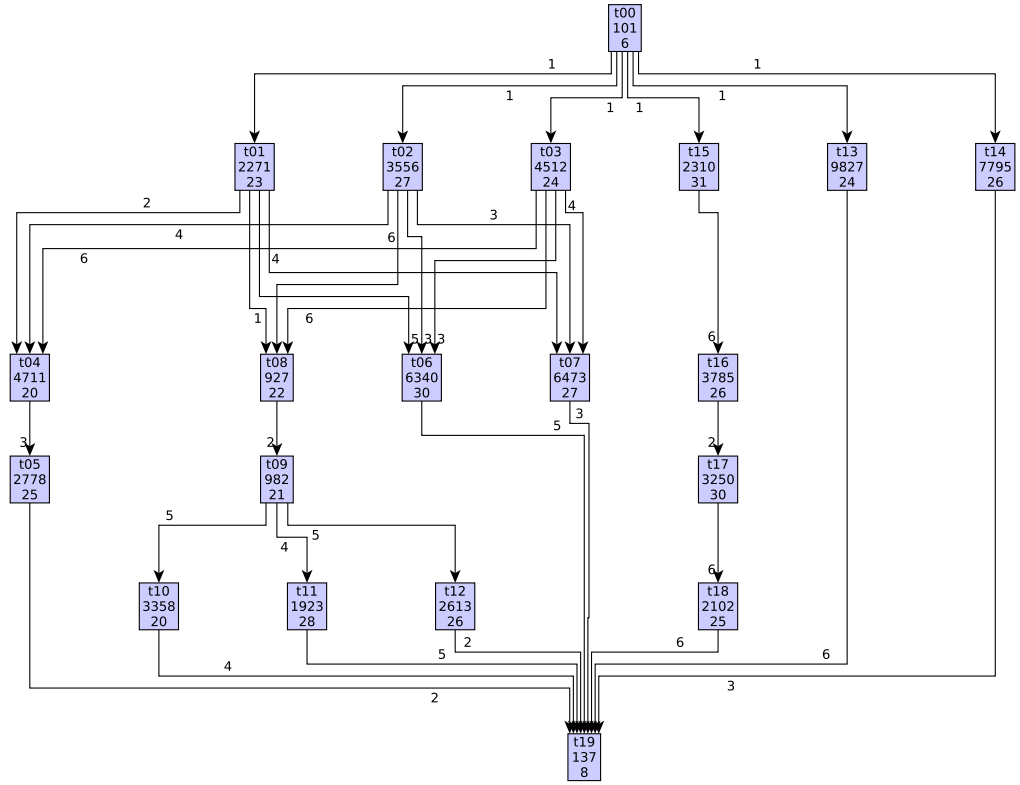


Figure 9.12: tg18 application graph

All these applications were executed on the list scheduling and our algorithm. For the list scheduling, only s35 and tg18 did not violate the memory constraints with the allocation proposed. Therefore we only compare these applications to our algorithm. Table 9.1 presents the results of the response time analysis run by MIA on the solutions produced by these algorithms.

	s35	s10x10	tg18	tg100
List Scheduling	8 361	-	12 423	-
Our Algorithm	7 835	71 147	12 311	227 870

Table 9.1: Mapping and Scheduling results (in cycles)

From these results we can see that our algorithm performs better than the list scheduling for the applications that can be compared. This evidences that the first phase of our solution has interesting heuristic ideas, trying to group together a node with its successors. In comparison with the list scheduling we tend to use more cores: 8 cores versus 7 for the s35 application and 10 versus 8 for the tg18 application.

For the bigger applications s10x10 and tg100, for now we only highlight the relevant impact that spawning more than one cluster has on the execution time. Looking at the final allocation we have used 3 clusters for the tg100 and 4 clusters for the s10x10. A lot of the cores contain a single node (that probably has no successors). This should be an improvement point to reduce the number of clusters and cores used.

9.4 PERFORMANCE IMPROVEMENT ON MIA

The MIA tool, responsible for the response time analysis in this work has been ameliorated with an improved algorithm version, as detailed in Section 8.1. Here we are interested in evaluating this new version by running it through randomly generated DAG scenarios. In order for MIA to perform a response time analysis on a DAG it needs to target a specific arbiter model. We start by detailing the bus arbiter that will be used in all tests.

9.4.1 *Bus Arbiter Function*

The Algorithm 8 discussed in Chapter 8 is designed in a modular way, in order to support different interference models. This way, the bus arbiter function $I^{\text{BUS}}(\tau, \mathcal{S}, b)$, called in Line 23, is completely independent from the main algorithm method. The inputs of this function are a task τ, a set $\mathcal{S}$ of tasks it interferes with and the bank b where those interferences happen. The output of this function is the interference perceived by the task τ, i. e. the number of cycles that the task τ is delayed by elements of the set $\mathcal{S}$.

The function I^{BUS} abstracts the arbiters found in processors when accessing the memory or a communication bus. The goal of this section is to compare the versions of the algorithm seen in Section 8.1: the original one in `C++` (Section 8.1.3) and the improved one in `Python` (Section 8.1.4). The Round-Robin (RR) arbitration function is implemented in both versions and used in the performance evaluation to provide a fair comparison. This arbitration model reflects more faithfully the MPPA2 system, implemented in the old and new MIA versions. The MPPA3 model was only implemented in the new version, once its improvements were verified through the experiments.

The RR arbitration was modeled with more details in Section 7.4.3.2. It is a simple and predictable policy, giving a fair quota of access for each initiator. In opposition to Equation (7.6), in this formulation we are already at a point in the algorithm were we are sure that the tasks interfere, thus there is no need to take release dates or response times as inputs. Here we denote by $a_b(\tau)$ the number of memory accesses (read or write) of t ask τ on bank b. The interference received by a task τ is then at most:

$$I^{\text{BUS}}(\tau, \mathcal{S}, b) = \sum_{\tau' \in \mathcal{S}} \min\left(a_b(\tau'), a_b(\tau)\right) \tag{9.3}$$

In Equation (9.3), the interference is given by a sum of minimums. The left term of the minimum comes from the fact that τ cannot be halted more than the number of accesses

from the other tasks. The right term of the minimum is justified because τ cannot be slowed down more than the number of times it accesses the memory.

Refinements can be added to the modeling of the RR based on assumptions from Section 8.1.4.1: from the point of view of task τ, tasks running on other cores can be seen as one long task. Thus, they can grouped together in the minimum computation. With n cores, and Λ the function that maps tasks to their core:

$$I^{\text{BUS}}(\tau, \mathcal{S}, b) = \sum_{i=1}^{n} \min \left(\sum_{\substack{\tau' \in \mathcal{S} \\ \Lambda(\tau')=i}} a_b(\tau'), a_b(\tau) \right) \tag{9.4}$$

Equation (9.4) presents the RR model implemented in the original and proposed algorithms and is used for a fair comparison between these versions, which is realized in the next section.

9.4.2 *Results*

Again, we chose to generate random DAGs, using the same method proposed in [138] and explained in Section 9.3.1. However, instead of relying on external mapping and scheduling algorithm, we use the layer structure the allocation. Tasks on the same layer are assigned to the cores in a cyclic way: the n-th task of a layer is assigned to the core $c = (n \mod \textit{number of cores})$. Tasks have randomly generated WCET, memory accesses and write operations on tasks of the next layer:

- WCET in $[550, 650]$;
- WCA in $[250, 550]$;
- Write operations in $[0, 100]$.

Two approaches are used to generate the inputs of the benchmark: fixed NL, in which the number of layers is constant and the layer size increases, and fixed LS, in which the layer size stays the same and it is the number of layers that gets enlarged.

The implementation of the original algorithm is done in `C++`, while the proposed algorithm is written in `Python`. This means that there is an interpreter overhead that negatively impact our results mainly for a small number of tasks.

	LS4	LS16	LS64	NL4	NL16	NL64
`Python` (new)	1.03	1.02	1.10	1.75	1.89	1.91
`C++` (original)	3.71	4.39	5.09	4.52	4.64	4.94

Table 9.2: n^x complexity comparison

A linear regression computation on a $\log \times \log$ scale from the benchmark values was done to see if the theoretical complexity goes in hand with the practical outcome. Table 9.2 shows the results, where NL4 represents a fixed number of layers of 4, and LS4 a fixed layer size of 4. The bus arbiter function used is the RR depicted in Equation (9.4) with a maximum allocation of the tasks to 16 cores.

The complexity of the proposed algorithm always stay under $O(n^2)$, contrary to the original one which exceeds $O(n^4)$ and even seems to reach $O(n^5)$ in the NL64 and LS64 cases.

These results are plotted in Figure 9.13, revealing the drastic improvement in computation time provided by the `Python` version. The benchmark has an execution timeout that the `C++` version easily reaches for more than 256 tasks.

In particular, as seen in Table 9.2, LS64 and NL64 are the random DAGs configuration values that showcase the biggest difference between the two versions. For LS64 and 256 tasks, the `C++` version took 1121.79*s* and the `Python` one took mere 4.13*s*, or 270 times faster. For N64 and 384 tasks, the `C++` implementation executed for 535.24*s* and the `Python` for only 0.9*s*, or 593 times faster.

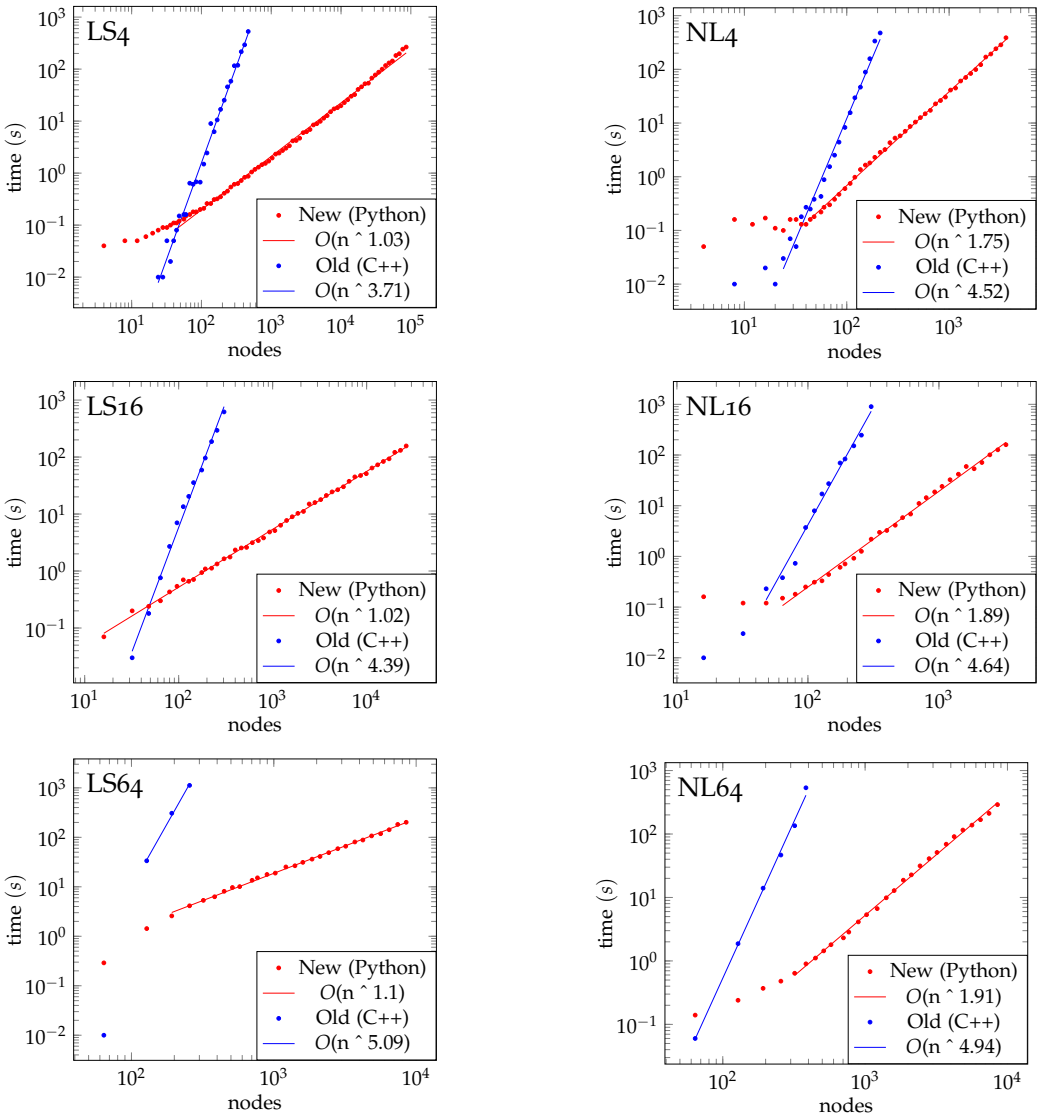


Figure 9.13: MIA old and new version benchmark results

9.4.3 *Discussion*

The revisited version of the algorithm shows a significant complexity improvement to $O(n^2)$, which translates to 593 times faster runtime in our benchmark, in comparison with the original version from [118]. This allows to accomplish the requirements of modern safety-critical real-time systems, scaling to more than 8 000 tasks in our experiments while maintaining a reasonable execution time.

9.5 CONCLUSION

This section has reunited all the experiments and results from our contributions introduced in Part II and also the MIA improvement in Part III. Starting with an overview of the applications used to evaluate the multiple phased execution models and the memory interference schemes proposed. Then we showed the generation process of random DAGs to later on assess on two mapping and scheduling algorithms. The chapter finishes with the complexity analysis and comparison of the new MIA response time analysis algorithm against its old version.

The Phased Execution Models experiments showed that a 2-Phased model gives a better result in terms of WCRT than any 3-Phased model. To apply this however, it is desirable that the platform has a banked memory that allows the Read/Execute phase to be run in isolation. Moreover, we showed that analyzing the interference instead of enforcing memory phases isolation provides shorter execution times. The evaluation done here could be extended to larger applications that require multiple clusters. The impact of intra-cluster communication with regards to the phased execution model can then be also measured.

The proof of concept algorithm developed for the multi-cluster mapping and scheduling of DAGs performed well against a classic list-scheduling when they are comparable. This is a preliminary work that aimed to spark new ideas in the community to better exploit architectures that present a cluster-based topology. Multiple future work directions can be investigated here. The first is to evaluate our proposed algorithm against others, outside the SCADE environment, that take into account the memory use of the platform. As seen in the results, our solution tends to spread nodes into several cores and clusters. A clear improvement path is to regroup these nodes more tightly, in order to attain a better overall platform utilization.

Finally, the improvement in the MIA tool is massive, reducing the complexity from $O(n^4)$ to $O(n^2)$, as proved experimentally, and allowing the scalability to complex systems with hundreds of tasks and cores. Future work include overlap management to provide a finer upper bound on the interference and thus less pessimistic results. It is based upon the idea that tasks will only be able to access the memory at the same time and interfere during their overlap period. Another work path is to develop a method to dynamically schedule the tasks accounting for interferences. This combines the scheduling, ordering and response time analyzer steps in the workflow presented in Chapter 4. The initial idea comes from[120] where it is stated that the lower complexity of the response time analyzer algorithm might allow it to be merged with a dynamic scheduler to implement predictable and optimized programs on many-core architectures.

10

CONCLUSION AND PROSPECTS

This thesis focused on providing an integrated approach to confidently use multi/many-core architectures for hard real-time systems. This integrated workflow covers the choice of an execution model, a strategy to map and schedule tasks and a hardware model to provide safe bounds on the response time.

The initial application is represented in the form of a Directed Acyclic Graph (DAG), with precedence constraints between nodes and explicit communication. This application can be issued from synchronous data flow languages or any other language or model-based development environment providing a DAG at the end of the compilation process.

The target architecture, the Kalray MPPA3 is a Commercial Off-The-Shelf (COTS) processor but with interesting characteristics that make it a good candidate for real-time systems. At the core level, it has in-order pipeline and private caches with a predictable replacement policy. At the cluster level, it provides low latency scratchpad memory and predictable arbitration policies. At the SoC level, it provides an AXI bus relying the different clusters with constant traversal time. This architecture is also said to be timing compositional, which allows response time analysis tools to be applied to it.

10.1 CONTRIBUTIONS

This thesis presents four major contributions: (i) the study of several execution models for real-time systems, (ii) algorithms for the mapping and scheduling of DAGs on multi/many-cores, (iii) the hardware arbitration model of the Kalray MPPA3 and (iv) the extension and improvement of existing software tools.

10.1.1 *Execution Models for Real-Time Systems*

Traditional software is not suited for real-time use. It shows a high degree of uncertainty regarding the moments when access to shared resources may occur. This justifies the decoupling of memory access phases from regular execution phases in order to have precise computation boundaries. These memory phases can then be executed in isolation on platforms with multiple cores to further increase the predictability.

Our contribution focused on the study of different phased execution models: 2-phased (with Execute and Write), 3-phased (with Read, Execute and Write) and 3-phased with a master core coordinating the memory phases. Moreover, the impact on the response

time is analyzed with two variants: running the memory phases in complete isolation, or allowing concurrent memory accesses and estimating the interference delay. Three scheduling algorithms are proposed for the 3-phased and 3-phased with master models.

We concluded through this study with multiple applications, including the ROSACE avionics case-study, that a 2-phased execution model yields the best results for the MPPA2 and MPPA3 platforms. With the banked memory and privatization features provided by these architecture, a reduced number of phases allows for less contention and easier scheduling. We also showed that analyzing the interference delay and allowing concurrent accesses to happen in a system gives better execution times than enforcing isolation between memory phases.

10.1.2 *DAG Mapping and Scheduling*

The mapping and scheduling of nodes from a DAG task model is a research topic that has been well explored in the last decades. We present here an algorithm that takes into account recent concerns raised by many-core SoCs with heterogeneous memory systems.

This heuristic-based algorithm has 3 phases: (i) grouping and ordering of nodes into a packet of virtual cores, (ii) clustering of these virtual cores into virtual clusters trying to minimize the communication cost and (iii) exhaustive exploration of virtual clusters to physical clusters assignment on the Kalray MPPA3 taking into account the AXI traversal cost. Our solution is compared against a classical list-scheduling method.

The evaluation has showed that, in the cases were we can compare ourselves with the classical list-scheduling, our algorithm gives slightly better results. For some of the randomly generated DAGs, the list-scheduling mapped tasks to cores that already had their memory full, revealing that this algorithm is not suited for systems with restricted local memory. We also observed a significant increase in the overall execution time when spawning the nodes into multiple clusters for larger applications.

10.1.3 *Timing Model of an Industrial Many-Core Architecture*

An industrial many-core architecture is a complex hardware circuit containing multiple cores, clusters, memories and the buses relying them. When accessing shared resources, cores must go through an arbiter to decide which one will go first. Providing an accurate model to estimate the delay introduced when concurrent accesses are performed is an important part of a framework for real-time systems.

The multiple Kalray MPPA3 arbitration points were modeled in this contribution. It starts at the arbiter between data and instruction caches, then moving on to the arbiter to access the cluster local memory, and then the multi-level arbitration system to access the memory belonging to another cluster. These models are implemented in the Multi-Core Interference Analysis (MIA) tool and used for all the evaluation involving this processor as a target.

10.1.4 *Tool Extensions*

The MIA tool is a software used for response time analysis in hard real-time systems. It receives as input a DAG, its mapping, scheduling and an architecture arbitration model.

Then using an algorithm it decides if, for a given deadline, this task set is schedulable or not. The timing analysis algorithm was rewritten using a simple approach: a time cursor that only takes into account the interference delay between tasks that are alive at a given moment. In contrast to the original version, it has no fixed point iteration, which reduces significantly the final complexity, from $O(n^4)$ to $O(n^2)$. This results in a reduction from 535 seconds to 0.90 seconds on a benchmark with 384 tasks, i. e. 593 times faster. The MIA tool becomes a good candidate for response time analysis in huge and complex systems as it now easily scales for thousands of tasks and cores.

SCADE is a synchronous data-flow language that is used as input in our framework and in particular for the execution models evaluation. It has a code generator for multi-core targets that needs to be extended in order to integrate with a specific architecture. We describe the changes in the integration script to properly initialize the MPPA3, then the synchronization using the mailbox and interrupt system, the time-triggered method using timers and the communication implementation using local memory and AXI bus.

Software-Hardware Interface for Multi-Many-Core (SHIM) is an IEEE standard designed to describe multi/many-core platforms in a way that can be useful for general software. The goal is to provide a description that can be used to automatically generate code using the characteristic of the platform or to analyze it regarding instruction and communication profiling. A Kalray MPPA3 model was conceived, with enough detail to reach such goals. However, for real-time purposes, the SHIM standard yet lacks mechanisms to describe worst-case scenarios to provide safe bounds for timing analysis.

10.2 FUTURE WORK

Within the context of the integrated workflow presented in this thesis, from a DAG based application to a safe real-time implementation on a many-core architecture, there are several possible future work directions.

SCADE is the proprietary synchronous data flow language used in this work to write the applications used in the evaluation chapter. Its Multi-Core Code Generator (MCG) was used to provide the functional code and integration script with the target platform. Nonetheless, it restricted some of the phased execution models due to an intrinsic contiguous task call and communication model. For all of these reasons, the starting point of the workflow can be enlarged to different languages such as Lustre, Simulink or Prelude, as long as they are able to generate a DAG based task set in the end of a compilation process.

The study on execution models for real-time systems could be extended to even more models, with more 2 or 3-phased variations. The experiments can be enriched with more applications, and in particular ones that use multiple clusters, instead of being restricted to a single one. The mapping and scheduling problem can be further developed with new heuristic based algorithms or compared to exact solutions with ILP solvers, if the scalability of the solution allows it.

The Kalray MPPA3 platform has proved to be an excellent candidate for safety-critical real-time systems. The reasons for that are multiple: predictability at core and cluster level, point to point cluster communication with constant time and a shared banked memory with an independent arbiter for each bank. The methodology presented in our workflow is general enough so that it can be applied to other platforms with similar characteristics. Another direction of research is to use the global memory of the MPPA3, by modeling

its DRAM controller, or use the NoC for communication, by circumventing the threaded DMA mechanism.

This thesis presents a first step on a workflow providing all the required steps to have a real-time system with guarantee bounds on the execution time. Choosing the appropriate execution model for the targeted platform is fundamental to efficiently explore it. A good mapping and scheduling strategy can drastically reduce the overall response time. Finally, knowledge of the hardware and software is extremely important to design models that reduce the pessimism when estimating the worst-case execution time. For modern platforms, the tight integration between all of these aspect is vital and must be further develop to ensure the wide adoption of multi/many-cores for safety-critical systems.

Part IV

APPENDIX

A

KALRAY MPPA3 HARDWARE DIAGRAMS

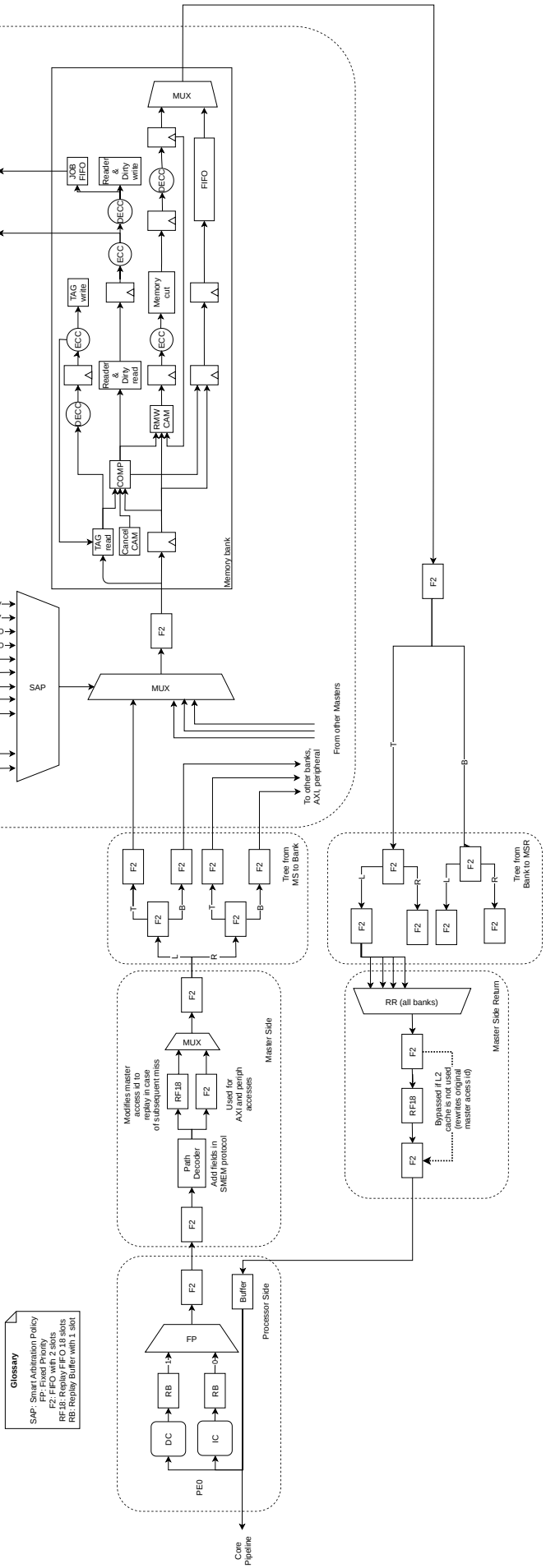


Figure A.1: SMEM data and control path diagram

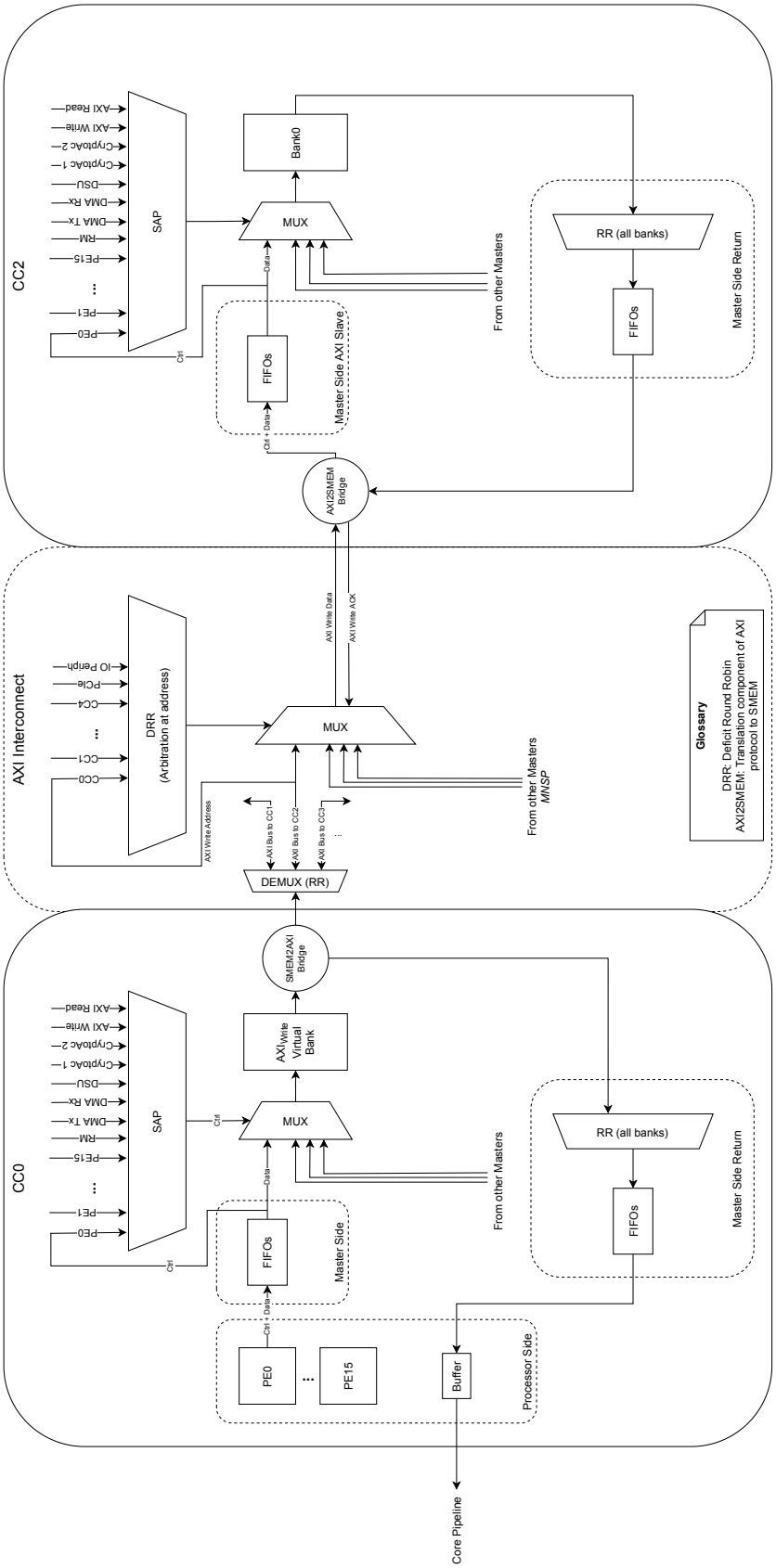


Figure A.2: Multi-Cluster data and control path diagram

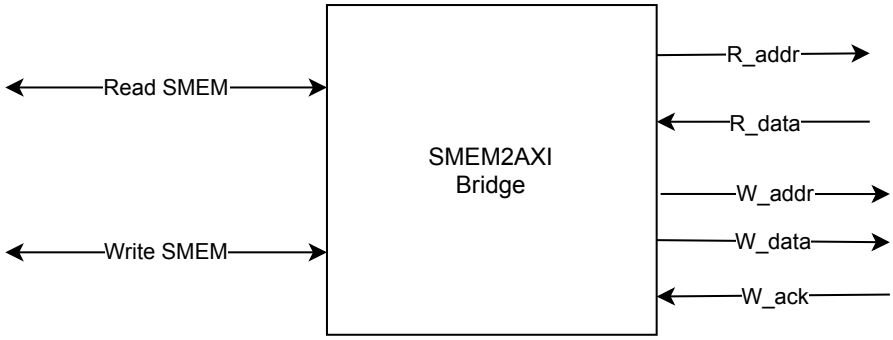


Figure A.3: SMEM to AXI Bridge

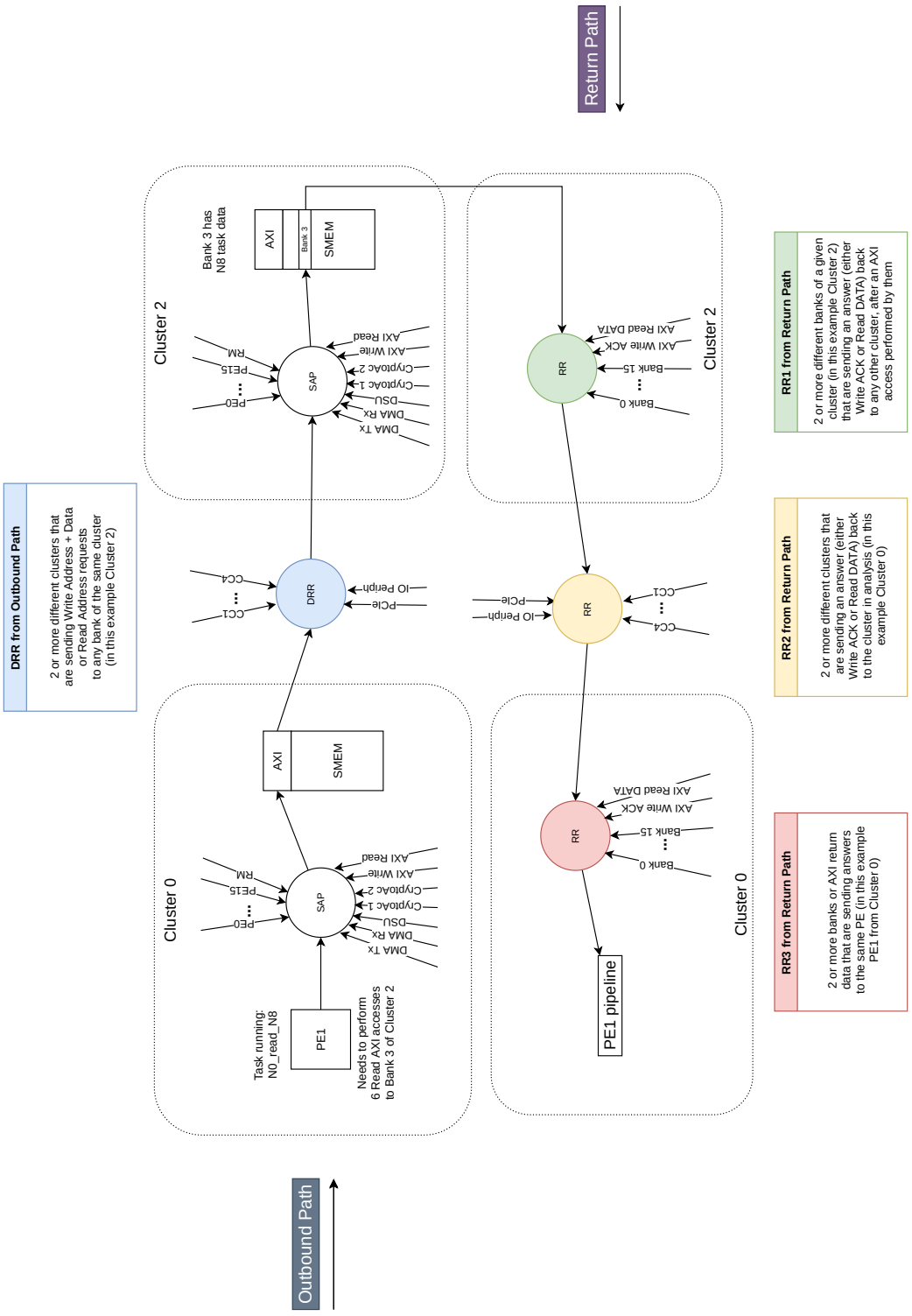


Figure A.4: Multi-Cluster Outbound and Return path arbiters

B

SCADE PLATFORM DEPENDENT CODE

B.1 INITIALIZATION CODE

Listing B.1 presents the initialization template code described in Section 8.2.3.2. It mixes `Python` and `C` code to generate the final initialization code at the end. This listing contains the `#include` section of a typical `C` program, global declarations and the main template code. This template is used by the SCADE helper script that imports the `Python` library containing all the the program information: workers, allocation, ...

```
<%namespace file="/mako_utils/mc_utils.mako" name="Utils"/>\
\
#include <stdio.h>
#include <stdlib.h>

/***** Init of MPPA includes ******/
/* Default inc paths */
#include <kv3/boot_c.h>
#include <machine/kv3/mppa3-80/gic.h>
#include <machine/kv3/mppa3-80/mailbox.h>

/* libmppahal inc paths */
#include <libmppahal/kv3/apic_gic.h>
#include <libmppahal/kv3/interrupt.h>
#include <libmppahal/kv3/registers.h>
#include <libmppahal/kv3/power.h>
#include <libmppahal/kv3/diagnostic.h>

#include "sync.h"
#include "timer.h"
#include "releases.h"
/***** End of MPPA includes ******/

/***** Init of SCADE tasks includes ******/
${Utils.print_includes(allocation.get_task_allocation())}\
<%
nonroot_workers = [w for w in allocation.get_task_allocation() if not w.
    is_root_worker()]
all_workers = [w for w in allocation.get_task_allocation()]
main_worker = allocation.get_root_worker()
%>\
/***** End of SCADE tasks includes ******/

/***** Init of additional MPPA boilerplate ******/
/*--------------------------------------------*/
#define PES_IT_NUM 1
#define GLOBAL_BARRIER_IT_NUM 2

#define GLOBAL_BARRIER_MAILBOX 124
```

```
% for w in all_workers:
#define SYNC_CL0_CO${w.get_index()}_RX_ID ${123 - w.get_index()}
% endfor

// 2KB of stack per PE (enough to perform printf)
#define PE_STACK_SIZE  0x800
/*--------------------------------------------*/
/***** End of additional MPPA boilerplate ******/

#define NB_STEPS 1

## display the main context in the form of structure
${ip.print_context_def()}

/* Allocate a global variable for each channel */
% for ch in reversed(mf.get_all_channels()):
/* ${ch.get_name()} */
%     if ch.get_initial_token():
${ch.get_type().get_name()} ${ch.get_name()}[2] __attribute__((section (".data_bank${
    Utils.get_channel_receiver(ch)}")));
%     else:
${ch.get_type().get_name()} ${ch.get_name()} __attribute__((section (".data_bank${
    Utils.get_channel_receiver(ch)}")));
%     endif
% endfor

int main ()
{
  eot_pes_configure(EOT_MODE_USE_WAITIT);

  /* Configure global barrier */
  mailbox_configure(GLOBAL_BARRIER_MAILBOX);
% for w in all_workers:
  mailbox_set_mask(GLOBAL_BARRIER_MAILBOX, ${w.get_index()});
  if (apic_gic_init_one_it(${w.get_index()}, GLOBAL_BARRIER_IT_NUM) != 0) {
    fprintf(stderr, "Error configuring GIC\n");
    assert(0 && "GIC configuration failed\n");
  }
  if (apic_gic_connect_it_src_2_it_dest(${w.get_index()}, GLOBAL_BARRIER_MAILBOX,
      GLOBAL_BARRIER_IT_NUM) != 0) {
    fprintf(stderr, "Error APIC GIC connect\n");
    assert(0 && "APIC GIC connection failed\n");
  }
% endfor

% for w in all_workers:
  __kvx_start_pe(${w.get_index()}, (void *)thread_PE${w.get_index()}, NULL, &
      stack_PE${w.get_index()}[PE_STACK_SIZE - 1]);
  eot_mailbox_set_mask_pe(${w.get_index()});
% endfor

  eot_wait_pes();
```

```
  printf("*** All done in RM ***\n");
  return 0;
}
```

Listing B.1: Initialization *template* code

Listing B.2 presents the generated C code from the template in Listing B.1.

```
#include <stdio.h>
#include <stdlib.h>

/***** Init of MPPA includes ******/
/* Default inc paths */
#include <kv3/boot_c.h>
#include <machine/kv3/mppa3-80/gic.h>
#include <machine/kv3/mppa3-80/mailbox.h>

/* libmppahal inc paths */
#include <libmppahal/kv3/apic_gic.h>
#include <libmppahal/kv3/interrupt.h>
#include <libmppahal/kv3/registers.h>
#include <libmppahal/kv3/power.h>
#include <libmppahal/kv3/diagnostic.h>

#include "sync.h"
#include "timer.h"
#include "releases.h"
/***** End of MPPA includes ******/

/***** Init of SCADE tasks includes ******/
#include "kcg_consts.h"
#include "root.h"
#include "N0_task.h"
#include "N1_task.h"
#include "N3_task.h"
#include "N2_task.h"
#include "N4_task.h"
/***** End of SCADE tasks includes ******/

/***** Init of additional MPPA boilerplate ******/
/*-----------------------------------------------*/
#define PES_IT_NUM 1
#define GLOBAL_BARRIER_IT_NUM 2

#define GLOBAL_BARRIER_MAILBOX 124
#define SYNC_CL0_C00_RX_ID 123
#define SYNC_CL0_C01_RX_ID 122
#define SYNC_CL0_C02_RX_ID 121

// 2KB of stack per PE (enough to perform printf)
#define PE_STACK_SIZE  0x800
/*-----------------------------------------------*/
/***** End of additional MPPA boilerplate ******/
```

```
#define NB_STEPS 1

typedef struct {
  outC_root outC;
  inC_root inC;
} WU_root;

/* Allocate a global variable for each channel */
N0_task_out_ch_o2_type N0_task_out_ch_o2 __attribute__((section (".data_bank2")));
N0_task_out_ch_o1_type N0_task_out_ch_o1 __attribute__((section (".data_bank1")));
N0_task_in_ch_type N0_task_in_ch __attribute__((section (".data_bank1")));
N1_task_out_ch_type N1_task_out_ch __attribute__((section (".data_bank2")));
N2_task_out_ch_type N2_task_out_ch __attribute__((section (".data_bank2")));
N3_task_out_ch_type N3_task_out_ch __attribute__((section (".data_bank2")));
N3_task_in_ch_type N3_task_in_ch __attribute__((section (".data_bank2")));
N4_task_out_ch_type N4_task_out_ch __attribute__((section (".data_banko")));

int main ()
{
  eot_pes_configure(EOT_MODE_USE_WAITIT);

  /* Configure global barrier */
  mailbox_configure(GLOBAL_BARRIER_MAILBOX);
  mailbox_set_mask(GLOBAL_BARRIER_MAILBOX, 0);
  if (apic_gic_init_one_it(0, GLOBAL_BARRIER_IT_NUM) != 0) {
    fprintf(stderr, "Error configuring GIC\n");
    assert(0 && "GIC configuration failed\n");
  }
  if (apic_gic_connect_it_src_2_it_dest(0, GLOBAL_BARRIER_MAILBOX,
      GLOBAL_BARRIER_IT_NUM) != 0) {
    fprintf(stderr, "Error APIC GIC connect\n");
    assert(0 && "APIC GIC connection failed\n");
  }
  mailbox_set_mask(GLOBAL_BARRIER_MAILBOX, 1);
  if (apic_gic_init_one_it(1, GLOBAL_BARRIER_IT_NUM) != 0) {
    fprintf(stderr, "Error configuring GIC\n");
    assert(0 && "GIC configuration failed\n");
  }
  if (apic_gic_connect_it_src_2_it_dest(1, GLOBAL_BARRIER_MAILBOX,
      GLOBAL_BARRIER_IT_NUM) != 0) {
    fprintf(stderr, "Error APIC GIC connect\n");
    assert(0 && "APIC GIC connection failed\n");
  }
  mailbox_set_mask(GLOBAL_BARRIER_MAILBOX, 2);
  if (apic_gic_init_one_it(2, GLOBAL_BARRIER_IT_NUM) != 0) {
    fprintf(stderr, "Error configuring GIC\n");
    assert(0 && "GIC configuration failed\n");
  }
  if (apic_gic_connect_it_src_2_it_dest(2, GLOBAL_BARRIER_MAILBOX,
      GLOBAL_BARRIER_IT_NUM) != 0) {
    fprintf(stderr, "Error APIC GIC connect\n");
    assert(0 && "APIC GIC connection failed\n");
  }
```

```
  __kvx_start_pe(0, (void *)thread_PE0, NULL, &stack_PE0[PE_STACK_SIZE - 1]);
  eot_mailbox_set_mask_pe(0);
  __kvx_start_pe(1, (void *)thread_PE1, NULL, &stack_PE1[PE_STACK_SIZE - 1]);
  eot_mailbox_set_mask_pe(1);
  __kvx_start_pe(2, (void *)thread_PE2, NULL, &stack_PE2[PE_STACK_SIZE - 1]);
  eot_mailbox_set_mask_pe(2);

  eot_wait_pes();

  printf("*** All done in RM ***\n");
  return 0;
}
```

Listing B.2: Initialization generated code

B.2 TASK AND COMMUNICATION CODE

Listing B.3 defines auxiliary functions that are used in Listing B.4 to invoke the actual code generation for local variables, task calls and channel communication.

```
<%def name="decl_locals_for_remote_write(w)">\
/* local copies of channels */
% for mt in w.get_methods():
%   for ch in mt.get_input_channels():
%       if allocation.get_fifo_size(ch) == 2:
  size_t ${ch.get_name()}_r_idx=0;
%       endif
%   endfor
%   for ch in mt.get_output_channels():
%       if allocation.get_fifo_size(ch) == 2:
  size_t ${ch.get_name()}_w_idx=1;
%       else:
  ${ch.get_type().get_name()} ${ch.get_name()}_l;
%       endif
%   endfor
% endfor
</%def>\
\
<%def name="init_context_fields_for_remote_write(w)">\
<%Utils:iter_channel_fields w="${w}" args="f, path, ch">\
%   if ch.get_initial_token() is None:
%     if f.is_channel() == 'receiver':
  ${path} = &${ch.get_name()};
%     else:
  ${path} = &${Utils.get_channel_var(ch)}_l;
%     endif
%   else:
%     if f.is_channel() == 'receiver':
  ${path} = &${ch.get_name()}[${ch.get_name()}_r_idx];
%     else:
  ${path} = &${ch.get_name()}[${ch.get_name()}_w_idx];
```

```
%     endif
%   endif
</%Utils:iter_channel_fields>\
</%def>\
  <%def name="call_method_for_remote_write(mt, w, debug, indent)">\
<%Utils:call_method_tt mt="${mt}" w="${w}" debug="${debug}" indent="${indent}">
<%def name="begin_send(ch)">\
%   if allocation.needs_blocking_send(ch):
    SEM_WAIT(${ch.get_name()}_empty); /* TODO change for cnoc sync */
%   endif
</%def>
<%def name="end_send(ch)">\
    /* Virtual write task ${mt.get_name()}_write_${ch.get_receiver().get_name()} */
% if debug:
    mppa_tracepoint(${proj_name}, ${mt.get_name()}_w__in);
% endif
    memcpy(&${ch.get_name()}, &${ch.get_name()}_l, sizeof(${ch.get_type().get_name()
        }));
% if debug:
    mppa_tracepoint(${proj_name}, ${mt.get_name()}_w__out);
% endif
%   if ch.get_initial_token() is not None:
    ${ch.get_name()}_w_idx = (${ch.get_name()}_w_idx+1)%2;
%   endif
</%def>
<%def name="begin_recv(ch)">\
</%def>
<%def name="end_recv(ch)">\
%   if allocation.needs_blocking_send(ch):
    SEM_SIGNAL(${ch.get_name()}_empty); /* TODO change for cnoc sync */
%   endif
%   if ch.get_initial_token() is not None:
    ${ch.get_name()}_r_idx = (${ch.get_name()}_r_idx+1)%2;
%   endif
</%def>
</%Utils:call_method_tt>\
</%def>\
```

Listing B.3: Auxiliary functions for the task and communication *template* code

Listing B.4 uses the auxiliary functions defined in Listing B.3 to template the code generation for the main worker of a SCADE program: the `root` node, usually mapped to the first available core.

```
## Creation of main thread for root worker (thread0)
uint64_t stack_PE${main_worker.get_index()}[PE_STACK_SIZE] __attribute__((section (".
    data_bank${main_worker.get_index()}")));
void *thread_PE${main_worker.get_index()}(void *args) __attribute__((section (".
    text_bank${main_worker.get_index()}")));
void *thread_PE${main_worker.get_index()}(__attribute__((__unused__)) void *args)
{
  uint64_t origin_of_time_PE${main_worker.get_index()};
  size_t loop;
```

```
  ${Utils.print_context_decl(main_worker) | indent_lines(2)}\

  ${ip.get_input_ctx()}

  ${decl_locals_for_remote_write(main_worker)}

  /* initialize context fields for channels */
${init_context_fields_for_remote_write(main_worker)}\

  /* init and reset functions of the tasks */
${Utils.print_init_call(main_worker) | indent_lines(2)}

  __kvx_timer64_setup(UINT64_MAX, UINT64_MAX, 0, TIMER_0);
  /* Inter PE sync */
  mailbox_notify(GLOBAL_BARRIER_MAILBOX, __kvx_get_cpu_id());
  wait_apic_it_and_clear(GLOBAL_BARRIER_IT_NUM);
  // Reset PM with RE
  __builtin_kvx_wfxl(KVX_SFR_PMC, KVX_SFR_WFXL_VALUE(PMC_PM0C,_KVX_PM_RE));
  // Timer use as alternative
  origin_of_time_PE${main_worker.get_index()} = __kvx_timer64_get_value(TIMER_0);
  // Set PM0 to count for Processor Clock Cycle
  __builtin_kvx_wfxl(KVX_SFR_PMC, KVX_SFR_WFXL_VALUE(PMC_PM0C,_KVX_PM_PCC));

  /* execute main program loop */
  for(loop=0; loop < NB_STEPS; loop++) {
%   for mt in main_worker.get_methods():
${call_method_for_remote_write(mt=mt, w=main_worker, debug=debug, indent=4)}
%   endfor

  }
  printf("Perf count duration root PE:  %"PRIu64" cycles\n", __builtin_kvx_get(
      KVX_SFR_PM0));
  printf("*** All done in root PE ***\n");

  eot_exit_pe();
  return 0;
}
```

Listing B.4: Task and communication *template* code

Listing B.5 shows the generated C code from Listing B.4. Note that this code follows the time-triggered method described in Section 8.2.3.2.

```
  uint64_t stack_PE0[PE_STACK_SIZE] __attribute__((section (".data_banko")));
  void *thread_PE0(void *args) __attribute__((section (".text_banko")));
  void *thread_PE0(__attribute__((__unused__)) void *args)
  {
    uint64_t origin_of_time_PE0;
    size_t loop;
    WU_root Wu_Ctx_root;

    (&Wu_Ctx_root.inC)->i = 0; //kcg_int32

    /* local copies of channels */
```

```
    N3_task_in_ch_type N3_task_in_ch_l;
    N0_task_in_ch_type N0_task_in_ch_l;


    /* initialize context fields for channels */
    (&Wu_Ctx_root.outC)->N0_task_in_ch = &N0_task_in_ch_l;
    (&Wu_Ctx_root.outC)->N3_task_in_ch = &N3_task_in_ch_l;
    (&Wu_Ctx_root.outC)->N4_task_out_ch = &N4_task_out_ch;

    /* init and reset functions of the tasks */
#ifndef KCG_USER_DEFINED_INIT
  root_init(&Wu_Ctx_root.outC);
#else
#ifndef KCG_NO_EXTERN_CALL_TO_RESET
  root_reset(&Wu_Ctx_root.outC);
#endif /* KCG_NO_EXTERN_CALL_TO_RESET */
#endif /* KCG_USER_DEFINED_INIT */



  __kvx_timer64_setup(UINT64_MAX, UINT64_MAX, 0, TIMER_0);
  /* Inter PE sync */
  mailbox_notify(GLOBAL_BARRIER_MAILBOX, __kvx_get_cpu_id());
  wait_apic_it_and_clear(GLOBAL_BARRIER_IT_NUM);
  // Reset PM with RE
  __builtin_kvx_wfxl(KVX_SFR_PMC, KVX_SFR_WFXL_VALUE(PMC_PM0C,_KVX_PM_RE));
  // Timer use as alternative
  origin_of_time_PE0 = __kvx_timer64_get_value(TIMER_0);
  // Set PM0 to count for Processor Clock Cycle
  __builtin_kvx_wfxl(KVX_SFR_PMC, KVX_SFR_WFXL_VALUE(PMC_PM0C,_KVX_PM_PCC));

  /* execute main program loop */
  for(loop=0; loop < NB_STEPS; loop++) {
    /* Core task call root_1 */
    while(origin_of_time_PE0 - __kvx_timer64_get_value(TIMER_0) <= loop*
        reaction_period+rel_root_1)
        ;
    root_1(&Wu_Ctx_root.inC, &Wu_Ctx_root.outC);
    /* Virtual write task root_1_write_N3_task */
    memcpy(&N3_task_in_ch, &N3_task_in_ch_l, sizeof(N3_task_in_ch_type));
    /* Virtual write task root_1_write_N0_task */
    memcpy(&N0_task_in_ch, &N0_task_in_ch_l, sizeof(N0_task_in_ch_type));

    /* Core task call root_2 */
    while(origin_of_time_PE0 - __kvx_timer64_get_value(TIMER_0) <= loop*
        reaction_period+rel_root_2)
        ;
    root_2(&Wu_Ctx_root.outC);
  }

  printf("Perf count duration root PE:  %"PRIu64" cycles\n", __builtin_kvx_get(
      KVX_SFR_PM0));
  printf("*** All done in root PE ***\n");
```

```
  eot_exit_pe();
  return 0;
}
```

Listing B.5: Task and communication generated code

BIBLIOGRAPHY

[1] Jaume Abella, Damien Hardy, Isabelle Puaut, Eduardo Quinones, and Francisco J Cazorla. "On the comparison of deterministic and probabilistic WCET estimation techniques." In: *2014 26th Euromicro Conference on Real-Time Systems*. IEEE. 2014, pp. 266–275.

[2] Dennis Abts, Steve Scott, and David J Lilja. "So many states, so little time: Verifying memory coherence in the Cray X1." In: *Proceedings International Parallel and Distributed Processing Symposium*. IEEE. 2003, 10–pp.

[3] Thomas L Adam, K. Mani Chandy, and JR Dickson. "A comparison of list schedules for parallel processing systems." In: *Communications of the ACM* 17.12 (1974), pp. 685–690.

[4] Irune Agirre, Jaume Abella, Mikel Azkarate-Askasua, and Francisco J Cazorla. "On the tailoring of CAST-32A certification guidance to real COTS multicore architectures." In: *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*. IEEE. 2017, pp. 1–8.

[5] 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*. 2007, pp. 251–256.

[6] Benny Akesson, Williston Hayes Jr, and Kees Goossens. "Classification and analysis of predictable memory patterns." In: *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2010, pp. 367–376.

[7] Ahmed Alhammad and Rodolfo Pellizzoni. "Schedulability analysis of global memory-predictable scheduling." In: *Proceedings of the 14th International Conference on Embedded Software*. 2014, pp. 1–10.

[8] Coralie Allioux. "Exploration of Timing Anomalies on Simplistic Processor with Model-Checking." In: *13th Junior Researcher Workshop on Real-Time Computing*. 2019.

[9] 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: *RTNS*. 2015, pp. 129–138.

[10] Charles André. "SyncCharts: A visual representation of reactive behaviors." In: *I3S, Sophia-Antipolis, France, Tech. Rep. RR* (1996), pp. 95–52.

[11] Konstantin Andreev and Harald Racke. "Balanced graph partitioning." In: *Theory of Computing Systems* 39.6 (2006), pp. 929–939.

[12] William Aspray. "The Intel 4004 microprocessor: What constituted invention?" In: *IEEE Annals of the History of Computing* 19.3 (1997), pp. 4–15.

[13] Johannes Ax, Gregor Sievers, Julian Daberkow, Martin Flasskamp, Marten Vohrmann, Thorsten Jungeblut, Wayne Kelly, Mario Porrmann, and Ulrich Rückert. "CoreVA-MPSoC: A many-core architecture with tightly coupled shared and local data memories." In: *IEEE Transactions on Parallel and Distributed Systems* 29.5 (2017), pp. 1030–1043.

[14] Hamdi Ayed, Jérôme Ermont, Jean-luc Scharbarg, and Christian Fraboul. "Towards a unified approach for worst-case analysis of Tilera-like and Kalray-like NoC architectures." In: *2016 IEEE World Conference on Factory Communication Systems (WFCS)*. IEEE. 2016, pp. 1–4.

[15] Stanley Bak, Gang Yao, Rodolfo Pellizzoni, and Marco Caccamo. "Memory-aware scheduling of multicore task sets for real-time systems." In: *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*. IEEE. 2012, pp. 300–309.

[16] Felice Balarin, Luciano Lavagno, Praveen Murthy, Alberto Sangiovanni-Vincentelli, CD Systems, et al. "Scheduling for embedded real-time systems." In: *IEEE Design & Test of Computers* 15.1 (1998), pp. 71–82.

[17] Clément Ballabriga, Hugues Cassé, Christine Rochange, and Pascal Sainrat. "OTAWA: an open toolbox for adaptive WCET analysis." In: *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*. Springer. 2010, pp. 35–46.

[18] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, Mahesh Balakrishnan, and Peter Marwedel. "Scratchpad memory: A design alternative for cache on-chip memory in embedded systems." In: *Proceedings of the Tenth International Symposium on Hardware/Software Codesign. CODES 2002 (IEEE Cat. No. 02TH8627)*. IEEE. 2002, pp. 73–78.

[19] Ayoosh Bansal, Jayati Singh, Yifan Hao, Jen-Yang Wen, Renato Mancuso, and Marco Caccamo. "Reconciling predictability and coherent caching." In: *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. IEEE. 2020, pp. 1–6.

[20] Jessé Barreto de Barros, Renato Coral Sampaio, and Carlos Humberto Llanos. "An adaptive discrete particle swarm optimization for mapping real-time applications onto network-on-a-chip based MPSoCs." In: *Proceedings of the 32nd Symposium on Integrated Circuits and Systems Design*. 2019, pp. 1–6.

[21] Daniel Bates, Alex Bradbury, Andreas Koltes, and Robert Mullins. "Exploiting tightly-coupled cores." In: *Journal of Signal Processing Systems* 80.1 (2015), pp. 103–120.

[22] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Akesson, 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)*. IEEE. 2016, pp. 14–24.

[23] Matthias Becker, Saad Mubeen, Dakshina Dasari, Moris Behnam, and Thomas Nolte. "Scheduling multi-rate real-time applications on clustered many-core architectures with memory constraints." In: *2018 23rd Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2018, pp. 560–567.

[24] Chawki Benchehida, Mohammed Kamel Benhaoua, Houssam-Eddine Zahaf, and Giuseppe Lipari. "Task and Communication Allocation for Real-time Tasks to Networks-on-Chip Multiprocessors." In: *2020 Second International Conference on Embedded & Distributed Systems (EDiS)*. IEEE. 2020, pp. 9–14.

[25] MD Bennett and Neil C Audsley. "Predictable and efficient virtual addressing for safety-critical real-time systems." In: *Proceedings 13th Euromicro Conference on Real-Time Systems*. IEEE. 2001, pp. 183–190.

[26] Albert Benveniste, Paul Caspi, Stephen A Edwards, Nicolas Halbwachs, Paul Le Guernic, and Robert De Simone. "The synchronous languages 12 years later." In: *Proceedings of the IEEE* 91.1 (2003), pp. 64–83.

[27] 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.

[28] Gérard Berry. "SCADE: Synchronous design and validation of embedded control software." In: *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*. Springer, 2007, pp. 19–33.

[29] Gerard Berry, Sabie Moisan, and Jean-Paul Rigault. "Esterel: Towards a synchronous and semantically sound high-level language for real-time applications." In: *Proc. IEEE Real-Time Systems Symposium*. IEEE Computer Society Press. 1983, pp. 30–40.

[30] Marko Bertogna, Michele Cirinei, and Giuseppe Lipari. "Schedulability analysis of global scheduling algorithms on multiprocessor platforms." In: *IEEE Transactions on parallel and distributed systems* 20.4 (2008), pp. 553–566.

[31] Frédéric Boussinot. "Reactive C: An extension of C to program reactive systems." In: *Software: Practice and Experience* 21.4 (1991), pp. 401–428.

[32] Matthew Bridges, Neil Vachharajani, Yun Zhang, Thomas Jablin, and David August. "Revisiting the sequential programming model for multi-core." In: *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*. IEEE. 2007, pp. 69–84.

[33] Thomas Carle, Dumitru Potop-Butucaru, Yves Sorel, and David Lesens. "From dataflow specification to multiprocessor partitioned time-triggered real-time implementation." In: *Leibniz Transactions on Embedded Systems* 2.2 (2015), pp. 01–1.

[34] NY-C Chang, Y-Z Liao, and T-S Chang. "Analysis of shared-link AXI." In: *IET Computers & Digital Techniques* 3.4 (2009), pp. 373–383.

[35] Che-Wei Chang, Jian-Jia Chen, Tei-Wei Kuo, and Heiko Falk. "Real-time task scheduling on island-based multi-core platforms." In: *IEEE Transactions on Parallel and Distributed Systems* 26.2 (2014), pp. 538–550.

[36] Che-Wei Chang, Jian-Jia Chen, Waqaas Munawar, Tei-Wei Kuo, and Heiko Falk. "Partitioned scheduling for real-time tasks on multiprocessor embedded systems with programmable shared SRAMs." In: *Proceedings of the tenth ACM international conference on Embedded software*. 2012, pp. 153–162.

[37] Sheng-Wei Cheng, Che-Wei Chang, Jian-Jia Chen, Tei-Wei Kuo, and Pi-Cheng Hsiu. "Many-core real-time task scheduling with scratchpad memory." In: *IEEE Transactions on Parallel and Distributed Systems* 27.10 (2016), pp. 2953–2966.

[38] Edmund M Clarke. "Model checking." In: *International Conference on Foundations of Software Technology and Theoretical Computer Science*. Springer. 1997, pp. 54–56.

[39] Jean-Louis Colaço, Bruno Pagano, Cédric Pasteur, and Marc Pouzet. "Scade 6: From a Kahn semantics to a Kahn implementation for multicore." In: *2018 Forum on Specification & Design Languages (FDL)*. IEEE. 2018, pp. 5–16.

[40] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. "Scade 6: A formal language for embedded critical software development." In: *2017 International Symposium on Theoretical Aspects of Software Engineering (TASE)*. IEEE. 2017, pp. 1–11.

[41] Jean-Louis Colaço and Marc Pouzet. "Type-based initialization analysis of a synchronous dataflow language." In: *International journal on software tools for technology transfer* 6.3 (2004), pp. 245–255.

[42] Christoph Cullmann, Christian Ferdinand, Gernot Gebhard, Daniel Grund, Claire Maiza, Jan Reineke, Benoît Triquet, and Reinhard Wilhelm. "Predictability considerations in the design of multi-core embedded systems." In: *Proceedings of Embedded Real Time Software and Systems* 36 (2010), p. 42.

[43] Dennis Dams, Rob Gerth, and Orna Grumberg. "Abstract interpretation of reactive systems." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19.2 (1997), pp. 253–291.

[44] Andrew Danowitz, Kyle Kelley, James Mao, John P Stevenson, and Mark Horowitz. "CPU DB: recording microprocessor history." In: *Queue* 10.4 (2012), pp. 10–27.

[45] Dakshina Dasari, Benny Akesson, Vincent Nelis, Muhammad Ali Awan, and Stefan M Petters. "Identifying the sources of unpredictability in COTS-based multicore systems." In: *2013 8th IEEE international symposium on industrial embedded systems (SIES)*. IEEE. 2013, pp. 39–48.

[46] Dakshina Dasari and Vincent Nelis. "An Analysis of the Impact of Bus Contention on the WCET in Multicores." In: *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*. IEEE. 2012, pp. 1450–1457.

[47] Gordon B Davis. "Anytime/anyplace computing and the future of knowledge work." In: *Communications of the ACM* 45.12 (2002), pp. 67–73.

[48] 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* 54.3 (2018), pp. 607–661.

[49] Robert I Davis and Alan Burns. "A survey of hard real-time scheduling for multiprocessor systems." In: *ACM computing surveys (CSUR)* 43.4 (2011), pp. 1–44.

[50] Keryan Didier, Dumitru Potop-Butucaru, Guillaume Iooss, Albert Cohen, Jean Souyris, Philippe Baufreton, and Amaury Graillat. "Correct-by-construction parallelization of hard real-time avionics applications on off-the-shelf predictable hardware." In: *ACM Transactions on Architecture and Code Optimization (TACO)* 16.3 (2019), pp. 1–27.

[51] Benoît Dupont de Dinechin. "Consolidating High-Integrity, High-Performance, and Cyber-Security Functions on a Manycore Processor." In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–4.

[52] 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.

[53] Maximilien Dupont de Dinechin, Matheus Schuh, Matthieu Moy, and Claire Maiza. "Scaling up the memory interference analysis for hard real-time many-core systems." In: *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2020, pp. 330–333.

[54] Mark Dowson. "The Ariane 5 software failure." In: *ACM SIGSOFT Software Engineering Notes* 22.2 (1997), p. 84.

[55] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. "Predictable flight management system implementation on a multicore processor." In: *Embedded Real Time Software (ERTS'14)*. 2014.

[56] Christian Ferdinand and Reinhold Heckmann. "ait: Worst-case execution time prediction by static program analysis." In: *Building the Information Society*. Springer, 2004, pp. 377–383.

[57] Björn Forsberg, Maxim Mattheeuws, Andreas Kurth, Andrea Marongiu, and Luca Benini. "A Synergistic Approach to Predictable Compilation and Scheduling on Commodity Multi-Cores." In: *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems*. 2020, pp. 108–118.

[58] Hauke Fuhrmann and Reinhard von Hanxleden. "Taming graphical modeling." In: *International Conference on Model Driven Engineering Languages and Systems*. Springer. 2010, pp. 196–210.

[59] Gernot Gebhard. "Timing anomalies reloaded." In: *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2010.

[60] Georgia Giannopoulou, Kai Lampka, Nikolay Stoimenov, and Lothar Thiele. "Timed model checking with abstractions: Towards worst-case response time analysis in resource-sharing manycore systems." In: *Proceedings of the tenth ACM international conference on Embedded software*. 2012, pp. 63–72.

[61] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. "Scheduling of mixed-criticality applications on resource-sharing multicore systems." In: *2013 Proceedings of the International Conference on Embedded Software (EMSOFT)*. IEEE. 2013, pp. 1–15.

[62] 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.

[63] Joël Goossens, Shelby Funk, and Sanjoy Baruah. "Priority-driven scheduling of periodic task systems on multiprocessors." In: *Real-time systems* 25.2-3 (2003), pp. 187–205.

[64] Amaury Graillat. "Code Generation for Multi-Core Processor with Hard Real-Time Constraints." Theses. Univ. Grenoble Alpes, Nov. 2018. URL: `https://tel.archives-ouvertes.fr/tel-02069346`.

[65] Amaury Graillat, Claire Maiza, Matthieu Moy, Pascal Raymond, and Benoît Dupont de Dinechin. "Response time analysis of dataflow applications on a many-core processor with shared-memory and network-on-chip." In: *Proceedings of the 27th International Conference on Real-Time Networks and Systems*. 2019, pp. 61–69.

[66] Amaury Graillat, Matthieu Moy, Pascal Raymond, and Benoît Dupont De Dinechin. "Parallel code generation of synchronous programs for a many-core architecture." In: *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE. 2018, pp. 1139–1142.

[67] Friedrich Gretz and Franz-Josef Grosch. "Blech, Imperative Synchronous Programming!" In: *Languages, Design Methods, and Tools for Electronic System Design*. Springer, 2020, pp. 161–186.

[68] Lena Grimm. "From Lustre to Graphical Dataflow Programs." `https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-mt.pdf`. Master's thesis. Kiel University, Department of Computer Science, May 2019.

[69] Alban Gruin, Thomas Carle, Hugues Cassé, and Christine Rochange. "Speculative Execution and Timing Predictability in an Open Source RISC-V Core." In: *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2021, pp. 393–404.

[70] Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. "WCET analysis with MRU cache: Challenging LRU for predictability." In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4s (2014), pp. 1–26.

[71] Sebastian Hahn and Jan Reineke. "Design and analysis of SIC: a provably timing-predictable pipelined processor core." In: *Real-Time Systems* 56.2 (2020), pp. 207–245.

[72] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. "Towards compositionality in execution time analysis: definition and challenges." In: *ACM SIGBED Review* 12.1 (2015), pp. 28–36.

[73] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. "The synchronous data flow programming language LUSTRE." In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320.

[74] Nicolas Halbwachs. "Synchronous programming of reactive systems." In: *International Conference on Computer Aided Verification*. Springer. 1998, pp. 1–16.

[75] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. "CoMPSoC: A template for composable and predictable multi-processor system on chips." In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 14.1 (2009), pp. 1–24.

[76] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, Owen O'brien, and Partha Roop. "Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation." In: *ACM Transactions on Embedded Computing Systems (TECS)* 13.4s (2014), pp. 1–26.

[77] Damien Hardy and Isabelle Puaut. "WCET analysis of instruction cache hierarchies." In: *Journal of Systems Architecture* 57.7 (2011), pp. 677–694.

[78] 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)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2017.

[79] David Harel. "Statecharts: A visual formalism for complex systems." In: *Science of computer programming* 8.3 (1987), pp. 231–274.

[80] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. "The influence of processor architecture on the design and the results of WCET tools." In: *Proceedings of the IEEE* 91.7 (2003), pp. 1038–1054.

[81] Joseph Herkert, Jason Borenstein, and Keith Miller. "The Boeing 737 MAX: Lessons for engineering ethics." In: *Science and engineering ethics* 26.6 (2020), pp. 2957–2974.

[82] Salma Hesham, Jens Rettkowski, Diana Goehringer, and Mohamed A Abd El Ghany. "Survey on real-time networks-on-chip." In: *IEEE Transactions on Parallel and Distributed Systems* 28.5 (2016), pp. 1500–1517.

[83] Kentaro Honda, Sasuga Kojima, Hiroshi Fujimoto, Masato Edahiro, and Takuya Azumi. "Mapping method of matlab/simulink model for embedded many-core platform." In: *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE. 2020, pp. 182–186.

[84] "IEEE Standard for Software-Hardware Interface for Multi-Many-Core." In: *IEEE Std 2804-2019* (2020), pp. 1–84. DOI: 10.1109/IEEESTD.2020.8985663.

[85] Mathieu Jan, Mihail Asavoae, Martin Schoeberl, and Edward A Lee. "Formal semantics of predictable pipelines: a comparative study." In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2020, pp. 103–108.

[86] Anjali Joshi and Mats PE Heimdahl. "Model-based safety analysis of simulink models using SCADE design verifier." In: *International Conference on Computer Safety, Reliability, and Security*. Springer. 2005, pp. 122–135.

[87] Peter Kafka. "The automotive standard ISO 26262, the innovative driver for enhanced safety assessment & technology for motor cars." In: *Procedia Engineering* 45 (2012), pp. 2–10.

[88] 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: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2014, pp. 145–154.

[89] Hyoseung Kim, Arvind Kandhalu, and Ragunathan Rajkumar. "A coordinated approach for practical OS-level cache management in multi-core real-time systems." In: *2013 25th Euromicro Conference on Real-Time Systems*. IEEE. 2013, pp. 80–89.

[90] Jung-Eun Kim, Man-Ki Yoon, Richard Bradford, and Lui Sha. "Integrated modular avionics (IMA) partition scheduling with conflict-free I/O for multicore avionics systems." In: *2014 IEEE 38th Annual Computer Software and Applications Conference*. IEEE. 2014, pp. 321–331.

[91] Sung Il Kim, Jong-Kook Kim, Hyoung Uk Ha, Tae Ho Kim, and Kyu Hyun Choi. "Efficient task scheduling for hard real-time tasks in asymmetric multicore processors." In: *International Conference on Algorithms and Architectures for Parallel Processing*. Springer. 2012, pp. 187–196.

[92] Ondrej Kotaba, Jan Nowotsch, Michael Paulitsch, Stefan M Petters, and Henrik Theiling. "Multicore in real-time systems–temporal isolation challenges due to shared resources." In: *16th Design, Automation & Test in Europe Conference and Exhibition*. 2013.

[93] Yu-Kwong Kwok and Ishfaq Ahmad. "Static scheduling algorithms for allocating directed task graphs to multiprocessors." In: *ACM Computing Surveys (CSUR)* 31.4 (1999), pp. 406–471.

[94] 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 (2014), pp. 736–773.

[95] Thierry Le Sergent, Adnan Bouakaz, and Guilherme Goretkin. "SCADE AADL." In: *ERTS 2018*. 2018.

[96] Guoning Liao, Erik R Altman, Vinod K Agarwal, and Guang R Gao. "A comparative study of multiprocessor list scheduling heuristics." In: *1994 Proceedings of the Twenty-Seventh Hawaii International Conference on System Sciences*. Vol. 1. IEEE. 1994, pp. 68–77.

[97] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D Patel, Stephen A Edwards, and Edward A Lee. "Predictable programming on a precision timed architecture." In: *Proceedings of the 2008 international conference on Compilers, architectures and synthesis for embedded systems*. 2008, pp. 137–146.

[98] Cláudio Maia, Luis Nogueira, Luis Miguel Pinho, and Daniel Gracia Pérez. "A closer look into the aer model." In: *2016 IEEE 21st International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE. 2016, pp. 1–8.

[99] Claire Maiza, Hamza Rihani, Juan M Rivas, Joël Goossens, Sebastian Altmeyer, and Robert I Davis. "A survey of timing verification techniques for multi-core real-time systems." In: *ACM Computing Surveys (CSUR)* 52.3 (2019), pp. 1–38.

[100] Louis Mandel and Marc Pouzet. "ReactiveML: a reactive extension to ML." In: *Proceedings of the 7th ACM SIGPLAN international conference on Principles and practice of declarative programming*. 2005, pp. 82–93.

[101] Florence Maraninchi. "The Argos language: Graphical representation of automata and description of reactive systems." In: *IEEE Workshop on Visual Languages*. Vol. 3. Citeseer. 1991.

[102] 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*. 2015, pp. 87–96.

[103] Yannick Moy, Emmanuel Ledinot, Hervé Delseny, Virginie Wiels, and Benjamin Monate. "Testing or formal verification: Do-178c alternatives and industrial experience." In: *IEEE software* 30.3 (2013), pp. 50–57.

[104] Frank Mueller. "Compiler support for software-based cache partitioning." In: *ACM Sigplan Notices* 30.11 (1995), pp. 125–133.

[105] Claire Pagetti, Julien Forget, Heiko Falk, Dominic Oehlert, and Arno Luppold. "Automated generation of time-predictable executables on multicore." In: *Proceedings of the 26th International Conference on Real-Time Networks and Systems*. 2018, pp. 104–113.

[106] Claire Pagetti, David Saussié, Romain Gratia, Eric Noulard, and Pierre Siron. "The ROSACE case study: From simulink specification to multi/many-core execution." In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2014, pp. 309–318.

[107] Marco Paolieri, Eduardo Quinones, and Francisco J Cazorla. "Timing effects of DDR memory systems in hard real-time multicore architectures: Issues and solutions." In: *ACM Transactions on Embedded Computing Systems (TECS)* 12.1s (2013), pp. 1–26.

[108] Paolo Pazzaglia, Alessandro Biondi, and Marco Di Natale. "Optimizing the functional deployment on multicore platforms with logical execution time." In: *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2019, pp. 207–219.

[109] Rodolfo Pellizzoni, Emiliano Betti, Stanley Bak, Gang Yao, John Criswell, Marco Caccamo, and Russell Kegley. "A predictable execution model for COTS-based embedded systems." In: *2011 17th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE. 2011, pp. 269–279.

[110] 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.

[111] Quentin Perret, Pascal Maurere, Eric Noulard, Claire Pagetti, Pascal Sainrat, and Benoit Triquet. "Temporal isolation of hard real-time applications on many-core processors." In: *Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2016, pp. 1–11.

[112] 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.

[113] Daniel Pilaud, N Halbwachs, and JA Plaice. "LUSTRE: A declarative language for programming synchronous systems." In: *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY*. Vol. 178. 1987, p. 188.

[114] Pascal Raymond. "Synchronous program verification with lustre/lesar." In: *Modeling and Verification of Real-Time Systems* (2008), p. 7.

[115] Jan Reineke, Isaac Liu, Hiren D Patel, Sungjun Kim, and Edward A Lee. "PRET DRAM controller: Bank privatization for predictability and temporal isolation." In: *2011 Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE. 2011, pp. 99–108.

[116] Hamza Rihani. "Many-Core Timing Analysis of Real-Time Systems." PhD thesis. Université Grenoble Alpes, Dec. 2017. URL: `https://tel.archives-ouvertes.fr/tel-01875711`.

[117] Hamza Rihani, Matthieu Moy, Claire Maiza, and Sebastian Altmeyer. "WCET analysis in shared resources real-time systems with TDMA buses." In: *Proceedings of the 23rd International Conference on Real Time and Networks Systems*. 2015, pp. 183–192.

[118] 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*. ACM. 2016, pp. 67–76.

[119] Juan M Rivas, Joël Goossens, Xavier Poczekajlo, and Antonio Paolillo. "Implementation of memory centric scheduling for COTS multi-core real-time systems." In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[120] Benjamin Rouxel, Steven Derrien, and Isabelle Puaut. "Tightening contention delays while scheduling parallel applications on multi-core architectures." In: *ACM Transactions on Embedded Computing Systems (TECS)* 16.5s (2017), p. 164.

[121] Benjamin Rouxel, Stefanos Skalistis, Steven Derrien, and Isabelle Puaut. "Hiding communication delays in contention-free execution for spm-based multi-core architectures." In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[122] 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)*. IEEE. 2015, pp. 220–229.

[123] Hassan Salamy and Jagannathan Ramanujam. "An effective solution to task scheduling and memory partitioning for multiprocessor system-on-chip." In: *IEEE transactions on computer-aided design of integrated circuits and systems* 31.5 (2012), pp. 717–725.

[124] Martin Schoeberl. "Time-predictable computer architecture." In: *EURASIP Journal on Embedded Systems* 2009 (2009), pp. 1–17.

[125] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, et al. "T-CREST: Time-predictable multi-core architecture for embedded systems." In: *Journal of Systems Architecture* 61.9 (2015), pp. 449–471.

[126] 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: *Design Automation Conference*. IEEE. 2010, pp. 332–337.

[127] Matheus Schuh, Claire Maiza, Joël Goossens, Pascal Raymond, and Benoît Dupont de Dinechin. "A study of predictable execution models implementation for industrial data-flow applications on a multi-core platform with shared banked memory." In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2020, pp. 283–295.

[128] Zheng Shi and Alan Burns. "Real-time communication analysis for on-chip networks with wormhole switching." In: *Second ACM/IEEE International Symposium on Networks-on-Chip (nocs 2008)*. IEEE. 2008, pp. 161–170.

[129] Cyril Six, Sylvain Boulmé, and David Monniaux. "Certified and efficient instruction scheduling: application to interlocked VLIW processors." In: *Proceedings of the ACM on Programming Languages* 4.OOPSLA (2020), pp. 1–29.

[130] Stefanos Skalistis and Angeliki Kritikakou. "Timely Fine-grained Interference-sensitive Run-time Adaptation of Time-triggered Schedules." In: *2019 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2019, pp. 233–245.

[131] Stefanos Skalistis and Alena Simalatsar. "Worst-case execution time analysis for many-core architectures with NoC." In: *International Conference on Formal Modeling and Analysis of Timed Systems*. Springer. 2016, pp. 211–227.

[132] Muhammad R Soliman and Rodolfo Pellizzoni. "PREM-based optimal task segmentation under fixed priority scheduling." In: *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2019.

[133] Lloyd Robert Still and Leandro Soares Indrusiak. "Memory-aware genetic algorithms for task mapping on hard real-time networks-on-chip." In: *2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*. IEEE. 2018, pp. 601–608.

[134] Vivy Suhendra and Tulika Mitra. "Exploring locking & partitioning for predictable shared caches on multi-cores." In: *Proceedings of the 45th annual Design Automation Conference*. 2008, pp. 300–303.

[135] Vivy Suhendra, Chandrashekar Raghavan, and Tulika Mitra. "Integrated scratchpad memory optimization and task scheduling for MPSoC architectures." In: *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*. 2006, pp. 401–410.

[136] Jinghao Sun, Feng Li, Nan Guan, Wentao Zhu, Minjie Xiang, Zhishan Guo, and Wang Yi. "On computing exact WCRT for DAG tasks." In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–6.

[137] Wei-Tsun Sun, Hugues Cassé, Christine Rochange, Hamza Rihani, and Claire Maiza. "Using execution graphs to model a prefetch and write buffers and its application to the Bostan MPPA." In: *9th European Congress on Embedded real time Software and Systems (ERTS 2018)*. 2018.

[138] 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.

[139] Saud Wasly and Rodolfo Pellizzoni. "Hiding memory latency using fixed priority scheduling." In: *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2014, pp. 75–86.

[140] Bpifrance Press Website. *Projet collaboratif ES3CAP, porté par Kalray*. Last accessed 26 January 2022. URL: https://presse.bpifrance.fr/lancement-du-projet-collaboratif-es3cap-porte-par-kalray-et-dote-dun-budget-de-222-me-pour-le-developpement-dune-plateforme-pour-les-systemes-intelligents-de-demainnbsp/.

[141] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. "The worst-case execution-time problem—overview of methods and survey of tools." In: *ACM Transactions on Embedded Computing Systems (TECS)* 7.3 (2008), pp. 1–53.

[142] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. "Worst case analysis of DRAM latency in multi-requestor systems." In: *2013 IEEE 34th Real-Time Systems Symposium*. IEEE. 2013, pp. 372–383.

[143] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Emiliano Betti, and Marco Caccamo. "Memory-centric scheduling for multicore hard real-time systems." In: *Real-Time Systems* 48.6 (2012), pp. 681–715.

[144] Gang Yao, Rodolfo Pellizzoni, Stanley Bak, Heechul Yun, and Marco Caccamo. "Global real-time memory-centric scheduling for multicore systems." In: *IEEE Transactions on Computers* 65.9 (2015), pp. 2739–2751.

[145] Simon Yuan, Li Hsien Yoong, and Partha S Roop. "Compiling esterel for multi-core execution." In: *2011 14th Euromicro Conference on Digital System Design*. IEEE. 2011, pp. 727–735.

[146] Shuai Zhao, Xiaotian Dai, Iain Bate, Alan Burns, and Wanli Chang. "DAG scheduling and analysis on multiprocessor systems: Exploitation of parallelism and dependency." In: *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE. 2020, pp. 128–140.