

Predictable execution on many-core processors

Quentin Perret

▶ **To cite this version:**

Quentin Perret. Predictable execution on many-core processors. Electronics. INSTITUT SUPERIEUR DE L'AERONAUTIQUE ET DE L'ESPACE (ISAE); UNIVERSITE DE TOULOUSE, 2017. English. NNT : . tel-01700758

HAL Id: tel-01700758
https://hal.science/tel-01700758

Submitted on 5 Feb 2018

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Université Fédérale



Toulouse Midi-Pyrénées

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par : *l'Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)*

Présentée et soutenue le *25 avril 2017* par :

Quentin PERRET

Predictable execution on many-core processors

Exécution prédictible sur processeurs pluri-cœurs

JURY

Mme. Isabelle PUAUT, Professeur, Université de Rennes I, Présidente du jury
M. Jean-Luc BÉCHENNEC, Chargé de Recherche CNRS, LS2N, Examinateur
M. Emmanuel GROLLEAU, Professeur, ISAE-ENSMA, Rapporteur
Mme. Claire MAIZA, Maître de Conférences, INP Grenoble, Examinatrice
M. Pascal MAURÈRE, Ingénieur, Airbus Operations SAS, Encadrant industriel
Mme. Claire PAGETTI, Maître de recherche, ONERA, Directrice de thèse
M. Laurent PAUTET, Professeur, Télécom ParisTech, Rapporteur
M. Pascal SAINRAT, Professeur, Université Toulouse III, Encadrant

École doctorale et spécialité :
MITT : Domaine STIC : Réseaux, Télécoms, Systèmes et Architecture
Unité de Recherche :
ISAE/ONERA - Modélisation et Ingénierie des Systèmes (MOIS)
Directeur(s) de Thèse :
Claire PAGETTI et Éric NOULARD
Rapporteurs :
Emmanuel GROLLEAU et Laurent PAUTET

This work was funded by Airbus and the French National Association for Research and Technology (ANRT) under grant n° 2013/1394.

Abstract

This thesis focuses on the design of safety-critical avionics embedded systems. During the last 25 years, the need for computational power aboard aircrafts has been constantly growing. Embedded applications are getting bigger as new functionalities are introduced. Simultaneously, the current trend in avionics is going towards deeply integrated systems, following the Integrated Modular Avionics (or IMA) philosophy where several functions can be co-hosted on a single execution platform. To support these evolutions, aircraft manufacturers need to bring more computational power aboard and to share it safely among multiple systems. The emergence of promising technologies such as many-core processors thus appears as a good opportunity to tackle these two challenges at once. However, safety-critical systems are required to meet not only functional, but also non-functional requirements resulting from safety constraints, performance issues or control theory. In particular, control laws are verified for specific latencies, thus imposing stringent *timing* constraints on how they can be implemented in software. As for the other parts of the aircraft, safety-related software components are also legally required to be *certified*. Thus, among other constraints, certification authorities require the validation of the timing behaviour of these systems by computing safe upper-bounds on their Worst Case Execution Times (or WCET). Unfortunately, computing WCETs is becoming increasingly harder as the architectures of micro-processors are becoming more and more complex, thus dramatically increasing software certification efforts.

In this thesis, we study the suitability of the distributed architecture of many-core processors for the design of highly constrained real-time systems as is the case in avionics. We firstly propose a thorough analysis of an existing COTS processor, namely the KALRAY MPPA®-256, and we identify some of its shared resources to be paths of interference when shared among several applications. We provide an execution model to restrict the access to these resources in order to mitigate their impact on WCETs and to temporally isolate co-running applications. We describe in detail how such an execution model can be implemented with a hypervisor which practically provides the expected property of temporal isolation at run-time. Based on this, we formalize a notion of *partition* which represents the association of an application with a resource *budget*. In our approach, an application placed in a partition is guaranteed to be temporally isolated from applications placed in other partitions. Then, assuming that applications and resource budgets are given, we propose to use constraint programming in order to verify automatically whether the amount of resources requested by a budget is sufficient to meet all of the application's constraints. Simultaneously, when a budget is valid, our approach computes a schedule of the application on the subset of the processor's resources allocated to it.

Overall, we provide an end-to-end integration framework enabling to share a many-core architecture safely, and to leverage its parallel computational power for highly constrained workloads. By doing so, we pave the way for the design of future embedded avionics computers based on many-core processors.

Résumé

La problématique considérée dans cette thèse concerne la conception des systèmes avioniques embarqués soumis à des contraintes de sûreté de fonctionnement. Les besoins en puissance de calcul à bord des avions augmentent régulièrement depuis 25 ans. Les applications embarquées grossissent à cause de l'ajout de nouvelles fonctionnalités. De plus, les choix de conception tendent vers des systèmes toujours plus intégrés. En particulier l'Avionique Modulaire Intégrée (ou *IMA* en anglais) permet l'hébergement de plusieurs fonctions sur une seule cible d'exécution. Pour accompagner ces évolutions, les avionneurs doivent apporter davantage de puissance de calcul à bord et être capables de la partager entre plusieurs systèmes. L'émergence de technologies prometteuses telles que les processeurs pluri-cœurs semble ainsi être une bonne opportunité pour résoudre ces deux problèmes. Cependant, les systèmes embarqués critiques doivent non seulement respecter des exigences fonctionnelles, mais aussi des exigences non fonctionnelles découlant de contraintes de sûreté, de performance ou bien de l'automatique. En particulier, des contraintes *temporelles* fortes sont imposées sur leur implantation logicielle. Comme pour les autres parties d'un avion, les composants logiciels critiques se doivent d'être *certifiés*. Ainsi, les autorités de certification imposent notamment qu'une validation temporelle de ces systèmes soit effectuée par le calcul du temps d'exécution pire-cas (ou *WCET* en anglais) des programmes. Malheureusement, calculer des WCETs peu pessimistes s'avère être de plus en plus difficile à mesure que les architectures des processeurs se complexifient.

Dans cette thèse, nous étudions l'adéquation de l'architecture distribuée des processeurs pluri-cœurs avec les besoins des concepteurs de systèmes temps réels avioniques. Nous proposons d'abord une analyse détaillée d'un processeur sur étagère (COTS), le KALRAY MPPA®-256, et nous identifions certaines de ses ressources partagées comme étant les goulots d'étranglement limitant à la fois la performance et la prédictibilité lorsque plusieurs applications s'exécutent. Pour limiter l'impact de ces ressources sur les WCETs, nous définissons formellement un modèle d'exécution isolant temporellement les applications concurrentes. Son implantation est réalisée au sein d'un hyperviseur offrant à chaque application un environnement d'exécution isolé et assurant le respect des comportements attendus en ligne. Sur cette base, nous formalisons la notion de *partition* comme l'association d'une application avec un *budget* de ressources matérielles. Dans notre approche, les applications s'exécutant au sein d'une partition sont garanties d'être temporellement isolées des autres applications. Ainsi, étant donné une application et son budget associé, nous proposons d'utiliser la programmation par contraintes pour vérifier automatiquement si les ressources allouées à l'application sont suffisantes pour permettre son exécution de manière satisfaisante. Dans le même temps, dans le cas où un budget est effectivement valide, notre approche fournit un ordonnancement et un placement complet de l'application sur le sous-ensemble des ressources du processeur allouées à sa partition.

De manière générale, nous proposons un atelier d'intégration de bout en bout qui permet de partager une architecture pluri-cœurs de manière sûre et d'exploiter sa puissance de calcul parallèle pour des applications contraintes. De ce fait, cet atelier est un premier pas vers la conception de calculateurs avioniques futurs à base de processeurs pluri-cœurs.

Remerciements

Je tiens tout d'abord à remercier l'ensemble des membres de mon jury: Isabelle Puaut pour avoir accepté de le présider; Emmanuel Grolleau et Laurent Pautet pour la patience et la bienveillance dont ils ont fait preuve en tant que rapporteurs; et enfin Claire Maiza et Jean-Luc Béchennec pour avoir accepté d'en faire partie.

Un très grand merci à mes cinq encadrants pour leur optimisme et pour avoir tenu bon par vents et marées. Merci à Claire Pagetti pour sa patience et sa rigueur dont j'ai eu bien besoin pendant ces trois années. Merci à Éric Noulard pour sa présence et ses conseils et à qui je souhaite sincèrement le meilleur dans sa nouvelle vie. Merci à Pascal Maurère pour sa gentillesse et son humanité qui m'ont beaucoup touché et m'ont aidé à ne pas baisser les bras dans les moments difficiles. Merci à Benoît Triquet grâce à qui j'ai énormément appris et qui n'apparait pas officiellement comme encadrant de ce travail pour des raisons purement administratives ne représentant pas la réalité. Et merci à Pascal Sainrat pour avoir toujours continué à être présent, et ce malgré toutes les difficultés et les changements d'encadrement.

Je tiens également à remercier sincèrement Michel Pasquier qui, bien qu'ayant suivi un chemin différent, a été à l'initiative de ce projet de thèse et sans qui je ne serais donc pas là où j'en suis aujourd'hui. Merci à toutes celles et ceux que j'ai pu rencontrer à Airbus ou à l'ONERA et grâce a qui j'ai pris un vrai plaisir à aller travailler pendant trois ans.

Je voudrais remercier du fond du cœur toute ma famille et tous mes amis qui m'ont toujours soutenu et supporté (même quand j'étais très pénible). Et bien sûr, merci à toi Ingrid, pour avoir rendu ces années tellement plus belles.

Je dédie mon travail à Barbara qui a tracé le chemin que nous avons parcouru et à Benji qui me manque chaque jour.

Contents

I Introduction **17**

1 General introduction **19**
1.1 Context . . . 19
1.1.1 Avionics context . . . 19
1.1.1.1 Safety requirements on software . . . 19
1.1.1.2 Software certification . . . 20
1.1.2 Industrial context . . . 21
1.1.2.1 Needs in computational power . . . 21
1.1.2.2 Integrated Modular Avionics . . . 21
1.1.3 Technological context . . . 22
1.1.3.1 Towards many-core processors . . . 23
1.1.3.2 COTS many-core processors . . . 24
1.2 Thesis motivations . . . 24
1.3 Approach . . . 25
1.4 Content of the dissertation . . . 26

2 Thesis context **27**
2.1 Application model . . . 27
2.1.1 Parallel task model . . . 27
2.1.2 Data exchanges and communications . . . 28
2.2 Architecture of the KALRAY MPPA®-256 . . . 29
2.2.1 Overview . . . 29
2.2.2 Compute clusters . . . 29
2.2.2.1 Shared SRAM . . . 31
2.2.2.2 Intra-cluster communications . . . 32
2.2.2.3 Inter-cluster communications . . . 33
2.2.2.4 Debug and Support Unit . . . 33
2.2.3 k1b cores . . . 33
2.2.4 I/O clusters . . . 34
2.2.4.1 Quad-core RM . . . 34
2.2.4.2 DRAM subsystem . . . 34
2.2.5 On-chip networks . . . 36
2.2.5.1 Switches . . . 36
2.2.5.2 Routing . . . 36
2.2.6 Remote memory operations . . . 37
2.2.7 Compatibility with a safety-critical context . . . 38
2.3 Additional constraints . . . 39

2.4 Summary . . . 40

II State of the Art **41**

3 Bounding execution times **43**
3.1 WCET estimation techniques . . . 43
3.1.1 Static analysis . . . 44
3.1.2 Measurement-based techniques . . . 46
3.1.3 Moving toward many-core processors . . . 46
3.2 Shared resources on many-core processors . . . 48
3.2.1 On-chip networks . . . 48
3.2.2 DRAM subsystem . . . 49
3.3 Interference penalties on many-core processors . . . 55
3.3.1 Network on Chip . . . 55
3.3.1.1 Deadlocks . . . 55
3.3.1.2 Computing WCTT with Network Calculus . . . 57
3.3.2 Temporal bound for DRAM subsystems . . . 59
3.3.3 End-to-end timing bounds on multiple resources . . . 60
3.4 Enforcing time-predictability by design . . . 60
3.4.1 Custom-made time-predictable hardware . . . 60
3.4.1.1 Processor architectures . . . 61
3.4.1.2 On-chip networks . . . 63
3.4.1.3 DDR-SDRAM controllers . . . 63
3.4.2 Software-enforced predictability on COTS . . . 64
3.4.2.1 Interference-aware control software . . . 64
3.4.2.2 Modification of applications . . . 66
3.5 Summary . . . 69

4 Off-line scheduling of parallel tasks **71**
4.1 Off-line scheduling of DAG tasks . . . 71
4.1.1 DAG model . . . 71
4.1.2 DAG notions . . . 72
4.1.3 Off-line model transformation . . . 73
4.1.4 Off-line mapping of DAGs on multi-core . . . 75
4.1.4.1 Real-time scheduling . . . 75
4.1.4.2 Makespan-optimization . . . 76
4.2 Off-line mapping on distributed architectures . . . 78
4.2.1 Mapping on the KALRAY MPPA®-256 . . . 78
4.2.2 Mapping on other many-core architectures . . . 80
4.2.3 Core and network co-scheduling . . . 82
4.3 Summary . . . 82

III Contributions **83**

5 Execution model for the KALRAY MPPA®-256 **85**
5.1 Bounds on shared resources access time . . . 85

5.1.1 SRAM bank access time . . . 85
5.1.2 WCTT on the NoC . . . 86
5.1.2.1 Packetization . . . 86
5.1.2.2 Time required to cross the NoC . . . 87
5.1.3 DRAM access time . . . 87
5.1.3.1 Model of the arbiter . . . 87
5.1.3.2 Evaluation of T_{req}^{DDR} . . . 88
5.1.4 Summary . . . 88
5.2 Mitigation of the interferences . . . 88
5.2.1 Definition of the execution model . . . 89
5.2.2 Motivations . . . 90
5.2.2.1 Rule 1 . . . 90
5.2.2.2 Rule 2 . . . 90
5.2.2.3 Rule 3 . . . 91
5.2.2.4 Rule 4 . . . 92
5.3 Summary . . . 92

6 Integration framework **95**
6.1 Input parameters . . . 95
6.1.1 Platform model . . . 95
6.1.2 Resource budget . . . 96
6.2 Off-line computations . . . 97
6.2.1 Validate . . . 97
6.2.2 Allocate . . . 98
6.3 On-line support . . . 100
6.4 Summary . . . 100

7 Implementation of the execution model **101**
7.1 Hypervisor description . . . 101
7.1.1 Assumptions . . . 102
7.1.1.1 Hardware configuration . . . 102
7.1.1.2 Global time-synchronization . . . 103
7.1.2 Enforcing the execution model . . . 103
7.1.2.1 Rule 1 . . . 103
7.1.2.2 Rule 2 . . . 104
7.1.2.3 Rule 3 . . . 106
7.1.2.4 Rule 4 . . . 108
7.2 Experimental validation . . . 108
7.2.1 Experimental setup . . . 108
7.2.1.1 The ROSACE case study . . . 109
7.2.1.2 The ImgInv co-running application . . . 110
7.2.1.3 Observation means . . . 111
7.2.2 Scenario 1: no interference . . . 112
7.2.3 Scenario 2: NoC interferences . . . 113
7.2.4 Scenario 3: DDR-SDRAM interferences . . . 114
7.3 Discussion on design trade-offs . . . 115
7.3.1 Memory footprint . . . 115
7.3.1.1 Footprint of scheduling tables . . . 115

7.3.1.2 Impact of non-contiguous memory areas . . . 116
7.3.1.3 Good practices . . . 117
7.3.2 WCET of the hypervisor . . . 118
7.4 Summary . . . 119

8 Validation of the budgets **121**
8.1 Budget choice . . . 121
8.1.1 Assumptions . . . 122
8.1.1.1 No code fetch . . . 123
8.1.1.2 Hypervisor . . . 123
8.1.2 Minimum constraints . . . 123
8.1.2.1 Local memory constraints . . . 123
8.1.2.2 Computational power constraints . . . 124
8.2 Budget validation . . . 124
8.2.1 Modelling framework . . . 124
8.2.1.1 Conditional time-intervals . . . 125
8.2.1.2 Cumulative functions . . . 126
8.2.1.3 Usual constraints on intervals . . . 126
8.2.2 Problem formulation . . . 127
8.2.2.1 Job-level solving . . . 127
8.2.2.2 Management of I/O requests . . . 127
8.2.2.3 Decision variables . . . 127
8.2.3 Constraints . . . 128
8.2.3.1 Sub-jobs mapping constraints . . . 128
8.2.3.2 PN utilization constraints . . . 129
8.2.3.3 Data mapping constraints . . . 129
8.2.3.4 PC utilization constraints . . . 130
8.2.3.5 Precedence constraints . . . 131
8.2.3.6 Determinism constraints . . . 132
8.3 Experimental results . . . 133
8.3.1 Goals . . . 133
8.3.1.1 Increasing work-load . . . 133
8.3.1.2 Narrowing the budget . . . 134
8.3.1.3 Choosing the DMA configuration . . . 134
8.3.2 Overview of the case study . . . 134
8.3.2.1 One PE per PN . . . 134
8.3.2.2 Prompt symmetric PCs . . . 134
8.3.3 Analysis of results . . . 135
8.4 Summary . . . 141

IV Conclusion **143**

9 General conclusion **145**
9.1 Summary of contributions . . . 145
9.1.1 Main contributions . . . 145
9.1.1.1 Integration framework . . . 145
9.1.1.2 Execution model . . . 146

9.1.1.3 Implementation of the execution model 146
9.1.1.4 Automated budget validation 146
9.1.2 Limitations of the approach 147
9.1.2.1 Limits of the execution model 147
9.1.2.2 Limits of the hypervisor 147
9.1.2.3 Limits of the CSP-based validation 147
9.2 Future perspectives 148
9.2.1 Execution model and implementation 148
9.2.1.1 Asynchronous NoC Schedule 148
9.2.1.2 On-line communication request over TDM schedule 149
9.2.2 Data management during validation 149
9.2.3 Automated allocation of concrete resources to budgets 150
9.2.4 Long term perspectives 150

V Résumé étendu en français **153**

Bibliography **196**

List of figures **209**

List of tables **213**

Part I

Introduction

Chapter 1

General introduction

Contents

1.1 Context 19
1.1.1 Avionics context 19
1.1.2 Industrial context 21
1.1.3 Technological context 22
1.2 Thesis motivations 24
1.3 Approach 25
1.4 Content of the dissertation 26

1.1 Context

This document presents the work I carried-out during my CIFRE PhD with Airbus and ONERA. The main purpose of this study is to assess the suitability of many-core processors to meet the needs and constraints of future avionics systems. Overall, the context of this thesis is threefold. Firstly, the *avionics* context imposes safety-requirements regarding the design of systems, including software development tasks and certification-related issues. Secondly, the *industrial* context brings needs for high computational power on-board in order to design more performant and more cost-efficient aircrafts. And finally, the *technological* context offers new perspectives for the design of future embedded systems thanks to promising emerging technologies.

1.1.1 Avionics context

Aircrafts must be safe and this entails that all systems having a potential impact on the flight safety are required to meet stringent safety-related constraints. When it comes to developing safety-critical software, several safety-related constraints need to be met during design, implementation, verification and test phases of the software's life cycle.

1.1.1.1 Safety requirements on software

Software issues unfortunately happened to be the root cause of several system failures causing the loss of human lives in the past. For example, in 1992, the anti-missile PATRIOT failed to intercept an incoming missile because of accumulated errors in floating point calculations [1]. As a result, 28 soldiers were killed and approximately 100 persons were injured. More recently, an Airbus A400M

LEVEL	PROBA.	SEVERITY	FAILURE CONDITION EFFECT
DAL - A	$10^{-9}/h$	Catastrophic	All failure conditions which prevent continued safe flight and landing
DAL - B	$10^{-7}/h$	Hazardous	Large reduction in safety margins or functional capabilities; higher workload or physical distress such that the crew could not be relied upon to perform tasks accurately or completely; adverse effects upon occupants
DAL - C	$10^{-5}/h$	Major	Significant reduction in safety margin or functional capabilities; significant increase in crew workload or in conditions impairing crew efficiency; some discomfort to occupants
DAL - D	$10^{-3}/h$	Minor	Slight reduction in safety margin; slight increase in crew workload; some inconvenience to occupants
DAL - E	N/A	No effect	None

Table 1.1: Description of the Design Assurance Levels from the ARP-4761 [5]

crashed during takeoff, killing all the crew on board [2]. Investigations revealed that an incorrect installation of the engine control software during production may have caused three out of four engines to stop responding to throttle commands, thus causing the accident.

In order to increase the confidence in all safety-critical systems used in aircrafts, constraints have been imposed to aircraft manufacturers. In particular, all safety-critical systems are legally required to be *certified* by independent authorities to be granted operational flight permission.

1.1.1.2 Software certification

Civil aviation is ruled by independent legal authorities. Before operational use, aircrafts are examined by the authorities to verify their safety. More precisely, manufacturers are expected to demonstrate the compliance of the aircraft with the authority's regulations to obtain *certification* and be permitted to fly civilians. Today, these regulations are mainly communicated with specific industry standards including the ARP-4754 [3] regarding system development, the DO-254 [4] regarding hardware development life cycle and the DO-178C [4] for the software development life cycle. Overall, the requirements imposed on designs depend on the *Design Assurance Level* (or *DAL*) of the system under consideration. Table 1.1 is extracted from the ARP-4761 [5] and provides a description of the five DALs. Naturally, developing high assurance software requires higher efforts since design, implementation, verification and testing are constrained in the DO-178C. Overall, software certification is mostly process-driven. As the rationale, it can be argued that properly developed software has higher chances of being correct. Yet, some of the constraints of the DO-178C do not only impact the development process but also the overall design choices for the system. As an example, DO-178C requires that applicants compute the Worst Case Execution Time (or WCET) of software programs. Computing non-pessimistic WCETs is a difficult task on modern processors. Being able to compute them efficiently on even more complex multi- or many-core processors in the future is a challenge that needs to be tackled.

The avionics context enforces safety-related constraints on the design of software. Certification problems lead to complex development processes and stringent constraints on demonstrability. Proving adherence to the standards will remain a necessity for future designs. Thus, deep software changes will be possible only if demonstrability and development processes can comply with the requirements of authorities.

1.1.2 Industrial context

When aircrafts are designed, safety is incontestably the highest priority. Yet, in an industrial context, manufacturers try to optimize performance and cost as much as possible without compromising safety. We detail the consequences on the design of embedded avionics computers below.

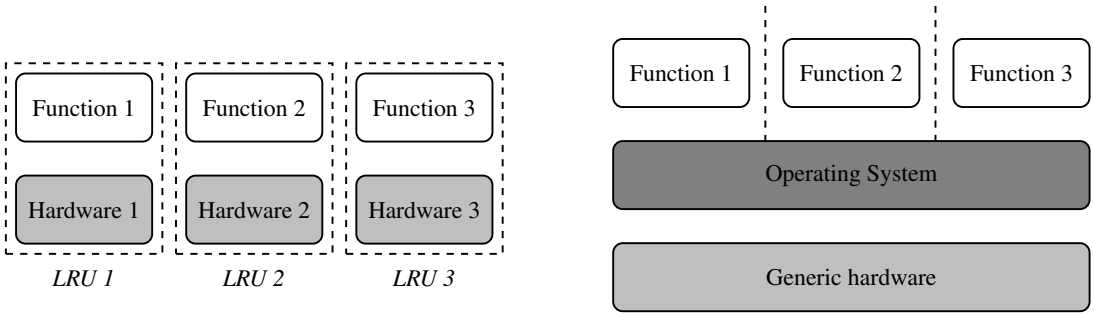
1.1.2.1 Needs in computational power

The need in embedded computational power within aircrafts is growing slower than in other areas of computer science. Yet, it still grows exponentially. Historically, new embedded computers were introduced to execute the flight control systems of each of the following aircrafts: A320 (first flight in 1987), A340 (first flight in 1991), A340 with new engines (first flight in 2001), A380 (first flight in 2005) and A350 (first flight in 2013). For each new generation, the need in computational power has been multiplied by a factor comprised between 2 and 3 in comparison with the previous generation. When looked over the past 25 years, this factor becomes close to 50. Such a raise is essentially motivated by evolutions in the control laws to either increase flight safety or reduce the aircraft's weight. For example, the *Gust Load Alleviation* (or *GLA*) method implemented in the A350 enabled to reduce the metal fatigue of the mechanical structure and thus to decrease its weight by 400 kilograms. In this case, the introduction of GLA required to add new elements to the control laws, to significantly increase the frequency at which incidence is measured, to meet short latency requirements at run-time and to implement thorough monitoring of the new parts of the system. Clearly, if GLA helped improve the A350's performance, it also came at the cost of an extra load on the CPU. Other evolutions of the flight control system alleviated the efforts on the mechanical structure such as the reduction of the rudder's oscillation tolerance on the A350. Again this enabled a significant reduction of the aircraft's weight and improved its performance overall. However, it also increased the computational work-load because of the additional complexity in the control laws, because of the new monitoring features that were required and because of the new entries introduced at system level. While it is clear that flight control systems are becoming more and more demanding, the growing need in computational power is shared more globally by many different systems in the aircraft. Currently, the cockpit-related systems are driving several innovation efforts which all rely on more electronics and more software components. Engineers investigate the feasibility of new features to improve many aspects of the aircraft, including maintenance problems or flying aids for pilots. And finally, promising technologies leveraging machine learning algorithms are now available to solve linear and logistic regression problems, to compress data using dimensionality-reduction, or to apply clustering techniques on large sets of data in a wide range of industries. While not currently in use, such algorithms are likely to find their path to embedded aerospace applications in the future. One of the key challenge here will be to bring sufficient computational resources on-board to process these potentially massive, but often parallel, workloads.

Overall, the needs for more powerful computers in aircrafts is growing from all sides. At the same time, industrial trends converge towards more integrated architectures where the same execution target can host the software parts of several systems. This concept of resource sharing is most commonly known as the concept of *Integrated Modular Avionics*.

1.1.2.2 Integrated Modular Avionics

Aircrafts embed many different systems. From the A300 and until the A340-500, Airbus' aircrafts used to feature one specific *Line Replaceable Unit* (or *LRU*) to support each and every function. A LRU is a dedicated embedded computer which is manufactured by a system supplier in order



(a) Federated approach (b) Integrated Modular Avionics

Figure 1.1: Comparison of federated and IMA architectures

to process the software components of one (and only one) system. With the exponential growth of functionality demands from airlines, aircrafts manufacturers concluded during the nineties that such an approach where the number of embedded computers grows accordingly could not be reasonably sustained any more. To tackle this issue, the concept of Integrated Modular Avionics (or IMA) was introduced. Honeywell firstly used it in 1995 within cockpit functions of the Boeing 777 aircraft[1]. Few years later, Airbus and Thales extended the concept during the design of the A380 with the introduction of the *Open IMA*. Today, the concept of IMA is becoming a standard used in most modern civil aircrafts such as the Boeing 787 Dreamliner and the Airbus A350.

As shown in Figure 1.1, the idea of IMA is to enable several applications to be hosted on a single generic platform. By sharing hardware resources (usually COTS), the number of embedded computers can be decreased and so do their weight, their production and design costs and their maintenance efforts. Yet sharing resources also brings interesting technical challenges to ensure robust partitioning between systems. Functions often need to be independent from each other and co-hosting their software parts requires to meet stringent temporal-isolation constraints. Systems with different criticality levels can be safely co-hosted on a single target if and only if failures of low-criticality systems cannot propagate to those with higher DALs. And finally, it is required for cost reasons that, despite a shared hardware, software systems should be developed, tested and *certified* independently from each other. The possibility of certifying software components independently enables *incremental certification* and greatly narrows certification costs.

Overall, the industrial context of this thesis brings two different challenges regarding the design of future avionics computers. The growing demand of computational power from applications needs to be met while, in the context of IMA systems, the property of robust partitioning between applications also needs to be ensured. Clearly, tackling these two issues simultaneously will become increasingly hard as applications grow and as execution targets are more and more shared. The emergence of new processors appears as a great opportunity to respond to the need for such computationally-efficient and computationally-shareable processors.

1.1.3 Technological context

Nowadays, the architectural paradigm upon which modern micro-processors are designed is evolving. The transistor density gains from Moore's Law [6] and the transistor speed growth both face technological limits bringing new challenges to sustain the fast-paced strive for computational performance. Deep changes in processor architectures already occurred in the past. Multi-core pro-

[1]This was the first IMA in a *civil* aircraft - IMA were already in use in several jet fighters such as the F-22, the F-35 or the Dassault Rafale at that time.

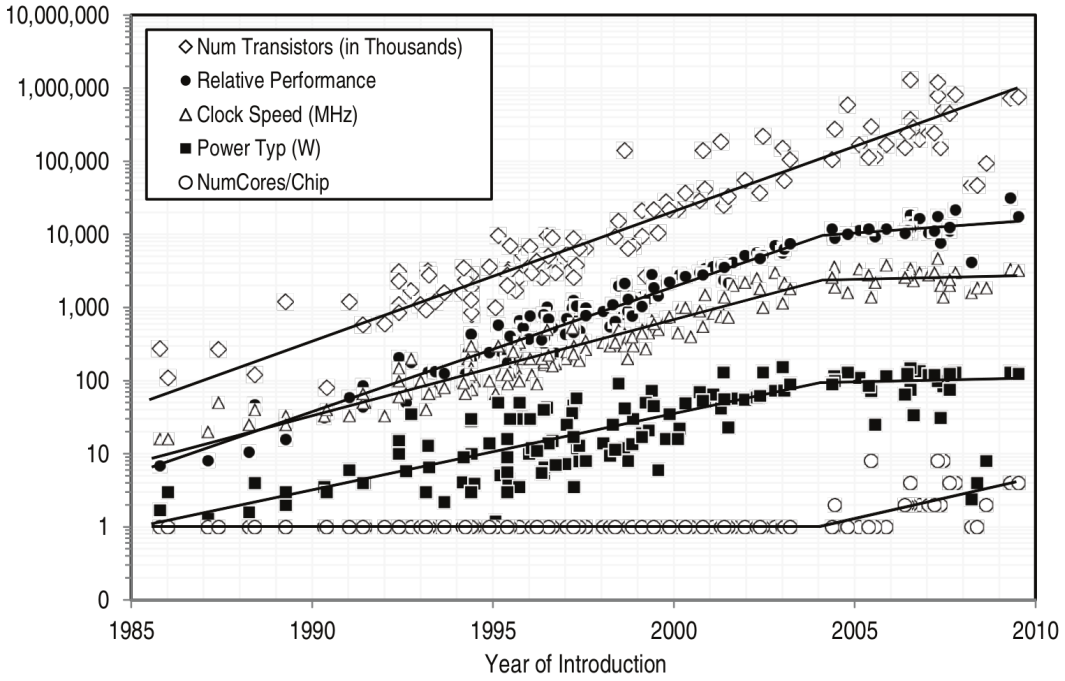


Figure 1.2: Logarithmic comparison of transistor count, frequency, power, performance and number of cores in micro-processors between 1985 and 2010 (extracted from [8])

cessors were introduced to beat the inflexion of performance scaling when pure architectural optimizations such as pipelining, speculative branching or out-of-order execution used to be the main solutions to increase performance. Today, traditional multi-core architectures are again facing difficulties to keep scaling up computational power at a reasonable energetic cost. As technological limits regarding transistor size and frequencies are close to be reached, designs are forced to rely on new architectures to sustain the pace at which performance needs are growing. In this context, new processors architectures featuring massively parallel processing arrays are now emerging. These processors are usually referred to as *many-core* processors.

1.1.3.1 Towards many-core processors

Recently, the design of *parallel* processors has gained a lot of attention. As shown in Figure 1.2, the number of transistors in micro-processors increased following Moore's Law over the past 25 years. However, it is clear that the raw sequential computational power of cores changed with the introduction of multi-core processors around 2004. The motivation for this change is related to the energetic profitability of adding further complex sequential optimization mechanisms to already complex pipelined, cached and speculative core architectures [7]. In multi-core processors, several computing cores and their peripherals (ex: DDR-SDRAM controllers, Ethernet controllers, ...) are typically interconnected by either a shared bus or a crossbar. Although such architectures successfully increased the computational power of micro-processors, it appears that, in both cases, scaling up the number of cores to hundreds or thousands now raises important implementation issues. With buses, concerns are related to performance balance of the cores vs the bus. With crossbars, concerns are related to their exponential cost as the number of end-points grows.

To overcome this issue, the concept of *tiled* architectures has been introduced [9]. Tiled micro-processors rely on several independent computing tiles interconnected by a Network-on-Chip (or

NoC) rather than a bus or a crossbar. By doing so, the architecture becomes scalable and allows to design processors with hundreds or even thousands of cores on a single chip. Those processors are referred to as the many-core processors. While they have been available only in research environments for few years, some of them are now available on the market.

1.1.3.2 COTS many-core processors

The market of *Commercial off-the-shelf* (or *COTS*) many-core processors is growing. Today, two main categories of many-core processors are available. Firstly, those dedicated to unload the main processor from computationally-intensive tasks and run as co-processors. The Intel Xeon Phi [10] family is a typical example of this category. Xeon Phi co-processors are available upon a variety of configurations with up to 72 cores consuming more than 200W. This order of magnitude in energy consumption is usually not suitable for embedded systems.

The Adapteva Epiphany [11] co-processor also falls in the same category, but with a better energetic consumption. Unfortunately, it does not feature enough computational power for the kind of applications targeted in thesis under it 16 cores configuration.

This category may also include *Graphical Processing Units* (or *GPUs*), especially if they can be used to process general purpose programs (GP-GPU) using OpenCL [12] or CUDA [13] for example. Yet, GPU are usually designed following the *Single Instruction Multiple Data* (or *SIMD*) philosophy which is not well suited to process several independent programs. We argue that *Multiple Instruction Multiple Data* (or *MIMD*) processors are better suited to our needs.

The second category of many-core processors is composed of those which can be used as main processors in a computer. Historically, the Intel Single-chip Cloud Computer (or *SCC*) [14] was probably one of the first many-core processor of that type available. With 48 cores and a potentially low (depending on frequency scaling) power consumption, the SCC is an interesting platform for our needs. Unfortunately, it has been designed only for research purposes and will not be used in any industrial product.

In the same category, the Tile Gx* [15] and Mx* [16] families from Mellanox (formerly Tilera, then EZchip) feature up to 72 and 100 processing cores respectively. This provides them with massive computational power. Unfortunately, although fairly low already, their thermal dissipation power is still too high to design electronics boards with only passive cooling. Even if it is not an absolute necessity, avoiding active cooling is an asset to design boards which do not require additional certification of the cooling system.

In contrary, the KALRAY MPPA®-256 [17] processor offers a good energetic consumptions which, under specific configurations, may be sufficiently low to avoid the problem of active cooling. With 288 cores, its raw computational power is also massive. Clearly, the KALRAY MPPA®-256 is an interesting candidate given our needs and constraints.

1.2 Thesis motivations

In an avionics context where software must be safe and certified, and an industrial context where heavy workloads need to be hosted on shared platforms, massively parallel processors such as the KALRAY MPPA®-256 offer interesting opportunities for the design of future avionics computers. Their raw computational power may theoretically enable to speedup on-board processing by a high factor, thus leaving either time for other applications to run or large margins on the timing constraints of applications. However, getting there involves to tackle several technical challenges first.

Firstly, bounds on the execution times of safety-critical software components must be found to meet certification criteria. Unfortunately, computing WCETs is getting harder as hardware complexity increases. In particular, it is well known from previous works on multi-core processors that, with parallel software, the main problems for computing WCETs lie in the management of shared resources. On massively parallel processors, some resources are massively shared. Consequently, mitigating the impact of shared resources on the execution time bounds will be a first challenge to overcome.

Secondly, the distributed architecture of many-core processors increases their programming complexity. The KALRAY MPPA®-256 has a hierarchical architecture with several types of peripherals, memories, NoCs and cores. In a safety-critical context where software designers must have total control of the hardware, being able to program and to master all parts of the processor at a very low level is especially challenging.

Finally, the cost of handmade mapping of applications on a many-core processor can quickly become prohibitive, especially when the targeted applications are large and complex. Tools automatically achieving this work would be a considerable asset to make many-core processors usable in an industrial context. However, the design of such tools is challenging since not only all the caveats of the target must be taken into account, but also because the approach is required to scale up to industrial-sized problems and to provide results in reasonable amounts of time.

1.3 Approach

As explained below, we propose to address the challenges listed above with an approach relying on three pillars. Firstly, regarding the problem of shared resources, we propose a thorough analysis of the architectural details of the KALRAY MPPA®-256. Based on this analysis, we provide an *execution model* limiting access to shared resources by applications to strict pre-defined boundaries. In particular, we focus on the management of three types of hardware resources that have a major impact on performance and predictability, namely the local memories, the NoC and the external DDR-SDRAM. Controlling the behaviour of applications when accessing those resources enables to isolate applications from each other, and thus to compute their WCETs in independence without excessive pessimism. Moreover, being able to isolate applications is a very desirable property in the context of IMA systems.

Secondly, regarding the problem of programming complexity, we propose a *hypervisor* that implements the principles stated by our execution model. This hypervisor is developed in bare-metal on the KALRAY MPPA®-256, and manages all resources under consideration. In particular, it manages the NoC and it enables the execution of applications that are distributed over several NoC end-points. Moreover, it enforces the respect of the execution model's rule at run-time and prevents propagation of faults between isolated applications. The validation of the hypervisor is achieved using experimental benchmarks. Overall, this development work not only demonstrates that using many-core processors can be made easier with an appropriate hypervisor, it also forms an execution framework running distributed applications, and thus to exploiting the computational power of such massively parallel architectures.

Finally, we address the issue of mapping large applications on the KALRAY MPPA®-256 automatically using constraint programming. We provide formal models of applications, of the hardware platform and of the needs for resources from applications. Based on this, we formulate the mapping problem using a modern solver and we perform extensive parametric studies to evaluate the capability of our approach to deal with a large industrial application provided with many

different sets of resources. Overall, we show that, by leveraging advanced features of state-of-the-art constraint solvers, mapping large and constrained applications on many-core processors can be done automatically and efficiently in a short-enough amount of time to be usable industrially.

1.4 Content of the dissertation

The remainder of this dissertation is structured as follows. Chapter 2 clarifies the inputs of this thesis. This includes the model of applications and details about the COTS many-core processor under consideration. Moreover, constraints inherited from the industrial context are described in further details. Chapter 3 and Chapter 4 review the state of art regarding predictable execution on multi- and many-core processors. The former covers the techniques employed to bound the execution times of programs while the latter focuses on scheduling and mapping problems on parallel architectures. Chapter 5 details and motivates the use of an execution model to master the access to shared resources and thus to improve predictability. Chapter 6 describes how the property of temporal isolation provided by the execution model can be leveraged to design a many-core integration work-flow. Overall it enables multiple parallel applications to be designed, implemented, verified and tested independently despite sharing hardware resources. Chapter 7 details how the applications are isolated on-line using a hypervisor that implements the rules of the execution model. Chapter 8 provides a solution to the problem of automatically mapping applications on some of the resources of many-core processor. Finally, Chapter 9 summarises the contributions of this thesis and outlines possibilities of future work.

Chapter 2

Thesis context

Contents

2.1 Application model 27
2.1.1 Parallel task model 27
2.1.2 Data exchanges and communications 28
2.2 Architecture of the KALRAY MPPA®-256 29
2.2.1 Overview 29
2.2.2 Compute clusters 29
2.2.3 k1b cores 33
2.2.4 I/O clusters 34
2.2.5 On-chip networks 36
2.2.6 Remote memory operations 37
2.2.7 Compatibility with a safety-critical context 38
2.3 Additional constraints 39
2.4 Summary 40

In this chapter, we present the inputs of the PhD work imposed by the industrial context. We firstly describe in Section 2.1 an abstract model of the avionics applications under consideration. Secondly, we provide in Section 2.2 extensive details on the COTS target that was chosen by Airbus and motivate this choice. Finally, we clarify in Section 2.3 the constraints that must be met in an aerospace and industrial context, such as the need for WCET computability or the limitations on eligible mapping and scheduling techniques to keep certification costs low.

2.1 Application model

In this section, we formalize the model of applications considered in the rest of the dissertation. This model is representative of existing control applications at Airbus and is thus imposed in the context of this thesis. An application is a tuple $< \tau, \delta >$ where τ is a finite set of parallel tasks and δ is a finite set of data.

2.1.1 Parallel task model

The set of tasks $\tau = \{\tau_1, \ldots, \tau_n\}$ is composed of several tasks $\tau_i =< S_i, P_i, T_i >$ with:

- $S_i = \{\tau_i^1, \ldots, \tau_i^{n_i}\}$ is a finite set of n_i sub-tasks. All sub-tasks are simultaneously activated at the activation of τ_i and share the same implicit deadline equal to the period of τ_i. In addition, each sub-task is defined by $\tau_i^j =< C_i^j, M_i^j, I_i^j, O_i^j >$ with:
  - C_i^j the WCET of the sub-task;
  - M_i^j the memory footprint of the sub-task, including the size of code, static and read-only data;
  - I_i^j and O_i^j respectively the input and output buffers [1] in which τ_i^j reads or writes.
- $P_i \subset S_i \times S_i$ is a set of ordered pairs of sub-tasks. P_i represents precedence relations constraining the order of execution of the sub-tasks. More precisely, $(\tau_i^x, \tau_i^y) \in P_i$ indicates that τ_i^x must complete its execution before τ_i^y can start. The set of precedences is assumed to be cycle-free and so, P_i can be represented with a *Directed Acyclic Graph* (or *DAG*).
- T_i is the period of τ_i. It also serves as implicit deadlines for all sub-tasks of τ_i.

We define the *Hyperperiod* of the taskset as $H = \underset{\forall i \in [1,n]}{lcm}(T_i)$ where lcm is the *least common multiple* function.

2.1.2 Data exchanges and communications

The data in $\delta = \{\delta_1, \ldots, \delta_m\}$ are produced and consumed by the sub-tasks of the application. More precisely, with $S = \bigcup_{\tau_i \in \tau} S_i$ the set of all sub-tasks in the application, each data δ_k is defined as a tuple $\delta_k =< m_k, prod, cons >$ where:

- m_k represents the size of the data;
- $prod : \delta \mapsto S$ associates each data to its unique producing sub-task;
- $cons : \delta \mapsto 2^S$ associates each data to the set of consuming sub-tasks.

Any two sub-tasks can exchange data, including pairs of sub-tasks that are not constrained by precedence relations. However, two sub-tasks constrained by precedence relations are always assumed to exchange data. Indeed, specifying a specific execution order of sub-tasks that do not communicate is usually not necessary in practice.

No assumption is made on the tasks periods. However, the model allows precedence relations only between sub-tasks of the same task, thus having the same period. Multi-rate precedences are not allowed. No such constraint is imposed on data. A data can be produced in a task and consumed in an other, even if the two tasks do not run at the same pace. The only assumption on such multi-rate data is that the consumed value is always the freshest one, that is, the one that was produced during the last activation of the producing sub-task. Doing so enables to bound the freshness of the consumed multi-rate data. The underlying assumption here is that the treatment made by a task of any multi-rate data is robust to its maximum production-to-consumption delay, that is equal to the period of the producing sub-task.

Example 1 (Example of application model)**.** *We consider an application composed of 2 tasks* τ_1 *and* τ_2*.* τ_1 *is composed of 4 sub-tasks* $\{\tau_1^1, \ldots, \tau_1^4\}$ *and* τ_2 *of 7 sub-tasks* $\{\tau_2^1, \ldots, \tau_2^7\}$*. The 11*

[1] The I/O buffers are used to represent the interactions with external components, i.e. transactions with the external DDR-SDRAM or reception/emission of Ethernet frames.

Sub-task	C_i^j (in ck)	M_i^j (in KiB)	I_i^j	O_i^j
τ_1^1	500	519	i_1	$\varnothing$
τ_1^2	500	397	$\varnothing$	$\varnothing$
τ_1^3	700	642	$\varnothing$	$\varnothing$
τ_1^4	400	262	$\varnothing$	$\varnothing$
τ_2^1	1,000	287	$\varnothing$	$\varnothing$
τ_2^2	400	799	$\varnothing$	$\varnothing$
τ_2^3	500	542	$\varnothing$	$\varnothing$
τ_2^4	800	764	$\varnothing$	$\varnothing$
τ_2^5	900	490	$\varnothing$	$\varnothing$
τ_2^6	500	399	$\varnothing$	o_1
τ_2^7	600	12	$\varnothing$	$\varnothing$

Table 2.1: Example of sub-tasks parameters

*sub-tasks exchange 15 data $\{\delta_a, \ldots, \delta_o\}$, read from 1 input buffer i_1 and write in 1 output buffer o_1. The periods of τ_1 and τ_2 respectively are $T_1 = 24$ and $T_2 = 48$. The precedence constraints between the sub-tasks are depicted in Figure 2.1a. The production and consumption of data and I/O buffers are depicted in Figure 2.1b. Table 2.1 provides the attributes of all sub-tasks (*ck *stands for clock cycle).*

2.2 Architecture of the KALRAY MPPA®-256

The KALRAY MPPA®-256 [17] is a many-core processor featuring 288 cores on a single chip. It targets the market of real-time computing-intensive low-power applications thanks to good temporal properties and a high computational power versus energy consumption ratio. In the following sections, we provide extensive details on the second version of the KALRAY MPPA®-256 architecture named *Bostan*.

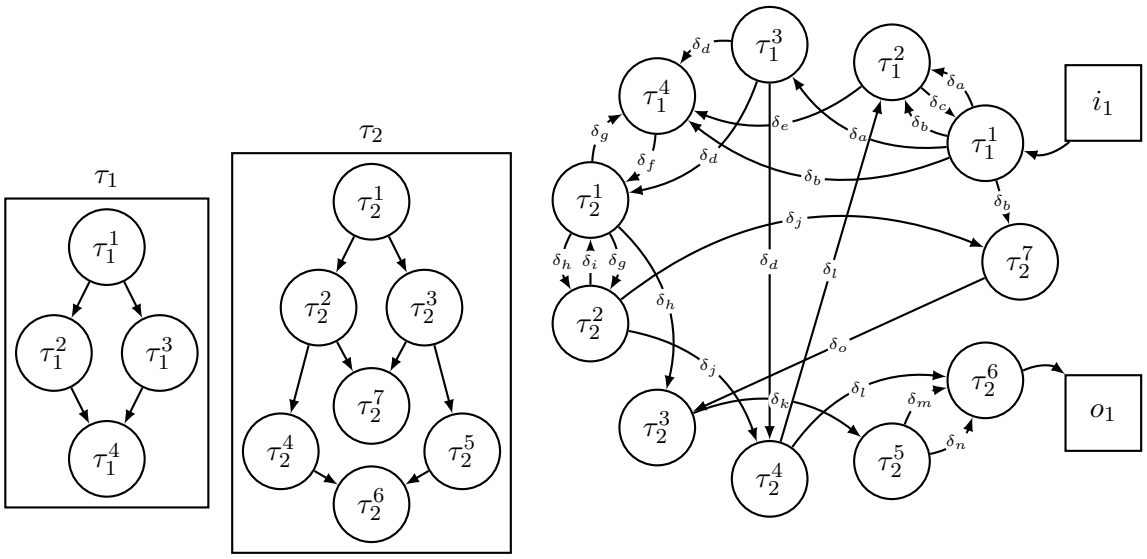
2.2.1 Overview

The KALRAY MPPA®-256 is organized in 16 *compute clusters* dedicated to run user code. Four additional *I/O clusters* serve as interfaces for managing communications with out-of-chip components or functions such as DDR-SDRAM or Ethernet. As depicted in Figure 2.2, all the clusters are interconnected with a dual 2D-torus NoC enabling point-to-point communications with explicit message-passing.

2.2.2 Compute clusters

As depicted in Figure 2.3, each compute cluster features:

- 16 cores, denoted as the *Processing Elements* (or *PEs*), for executing user code;
- 1 additional core denoted as *Resource Manager* or (*RM*) in charge of managing and configuring the cluster's local resources;
- 1 *Debug and Support Unit* (or *DSU*) to ease programming and debugging with JTAG;
- 1 *Direct Memory Access* (or *DMA*) unit to automate data sending over the NoC;



(a) Precedence constraints (b) Data and I/O buffers

Figure 2.1: Example of application model

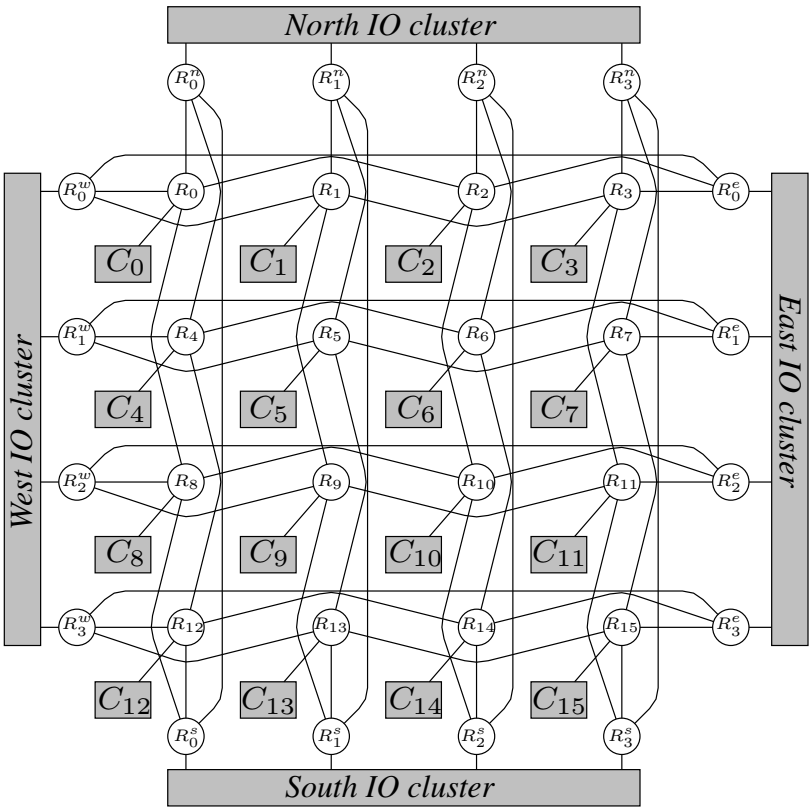


Figure 2.2: Architecture of the KALRAY MPPA®-256

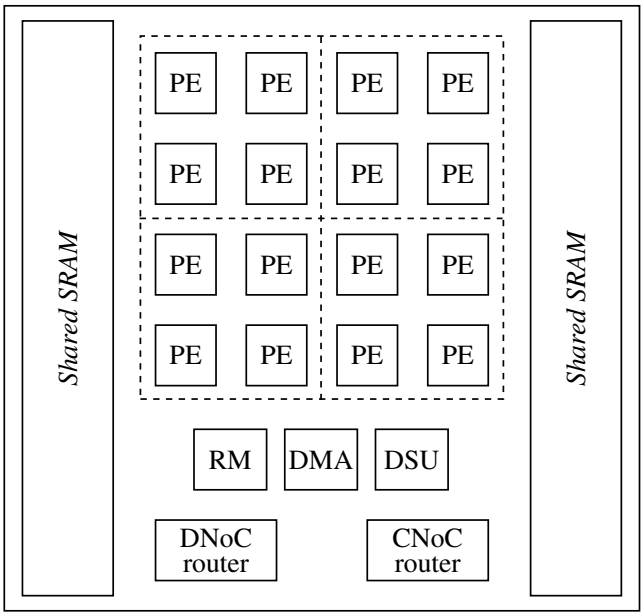


Figure 2.3: Architecture of compute clusters in the KALRAY MPPA®-256

- 2MiB of shared SRAM organized in 16 independent banks;
- 1 access to each NoC.

2.2.2.1 Shared SRAM

The 2 MiB of SRAM in a compute cluster are organized in 16 independent banks. Two local masters of the cluster (among PEs, the RM, the DMA or the DSU) can target two different banks without suffering from interferences. Their memory access paths are not dependent. However, concurrent accesses from two masters targeting the same bank will be arbitrated with a hierarchical arbitration policy depicted in Figure 2.4.

Arbitration policy Each local SRAM bank is accessible from 20 memory access paths: 16 serving the PEs (for both instruction and data caches), 1 for the RM, 1 for the DSU, 1 for the DMA when *receiving* packets from the NoC and 1 for the DMA when *emitting* packets. Each bank has a memory arbiter interleaving concurrent accesses from any of the 20 memory masters. PEs accesses are arbitrated in a Round-Robin fashion. Similarly, memory requests from the DSU, the RM and the DMA in emission are arbitrated in Round-Robin. The resulting accesses from this first level of arbitration enter another Round-Robin arbiter before eventually accessing the bank. The case of the received packets from the NoC is treated differently. The memory requests issued by the DMA when receiving packets have full priority over any other memory transactions to access any local SRAM bank. As a consequence, requests from PEs, the RM, the DMA in emission or the DSU are systematically stalled while the DMA writes the received data into memory. This is done to avoid congestion on the NoC.

Addressing modes The local memory addressing can be configured in two modes at startup: *interleaved* or *blocked* addressing schemes. When in *interleaved* mode, contiguous memory addresses jump from bank to bank every 64 bytes. This usually distributes the memory requests evenly over the 16 banks and provides good average performances without paying

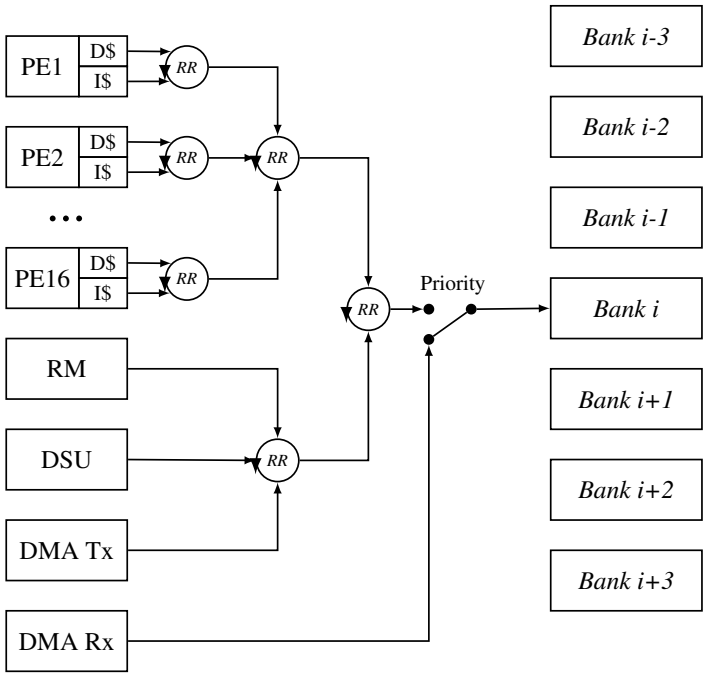


Figure 2.4: Hierarchical SRAM arbiter of KALRAY MPPA®-256

particular attention to the memory mapping. When in *blocked* mode, contiguous memory areas are not shuffled among the banks and each bank can be addressed directly with a linear 128KiB address space.

2.2.2.2 Intra-cluster communications

The communication between local elements inside a compute cluster are essentially achieved with shared memory. The two exceptions lie in the PEs shared registers and the notification system signaling to the RM and PEs the occurrence of specific events.

Cache coherency Each core in a compute cluster features one private instruction cache of 8KiB and one private data cache of 8KiB. All caches use the *Least Recently Used* (or *LRU*) line replacement policy. Cache coherency is not enforced by any hardware mechanisms inside clusters and not between clusters either. Specific instructions are provided to achieve uncached memory accesses, to purge write-buffers and to invalidate caches.

Memory Management Units Each core features a MMU capable of both virtual addressing and memory protection. *Translation Lookaside Buffers* (or *TLBs*) can be configured to enforce local caching policies (Write-Through or Bypass) on specific memory pages.

Events and shared registers As depicted in Figure 2.3, PEs are bundled in groups of four neighbors. In each group, PEs can send events to each others to achieve synchronization barriers. Moreover, the RM is capable of sending and receiving events to/from all PEs in the cluster. Each PE can also write directly to the 32 upper registers of its three neighbors.

2.2.2.3 Inter-cluster communications

All the inter-cluster communications are achieved by explicit message passing or remote DMA through the NoC. The DMA in each cluster unloads the cores from the work of pushing or receiving data through the network interfaces.

Sending data The DMA features a micro-engine capable of sending data through 8 *Tx channels*. Each Tx channel can be configured independently. In particular, each Tx channel can be set-up with a specific route and specific traffic regulation parameters. Those configurations are done using memory-mapped registers that should be written by the RM. PEs may be allowed to write such configurations if and only if the RM previously updated permissions registers to explicitly allow it.

The micro-engine of the DMA can run up to 8 threads simultaneously. Each thread will push data through any of the 8 Tx channels. In particular, each thread executes instructions from a specific binary code resulting from the compilation of a *micro-code* expressed in a light Kalray-specific assembly language. Threads can read parameters from memory-mapped registers that are written by the RM, or possibly the PEs if permission registers allow it. Those parameters can be used to pass the base address and the length of a buffer that shall be sent by the DMA for example.

Receiving data The DMA features 256 *Rx channels* that can be configured independently. Each Rx channel can be associated with a buffer of fixed length in the local SRAM. All the packets received through this channel will be written in this memory area. In addition, each Rx channel can be configured in a specific mode to notify the RM when an *End of Transmission* (or *EOT*) message has been received or when the total amount of data received overpassed a predefinite trigger for example.

2.2.2.4 Debug and Support Unit

Each cluster has a local DSU which provides JTAG links to help programming and debugging applications. Each DSU also provides a messaging interface to allow cores to push messages directly on the JTAG link. This can be used to enable lightweight logging mechanisms for example. Finally, each DSU has a 64-bits timestamp, counting cycles from boot time. All the timestamps of all DSUs are fully synchronous by design on Bostan. It provides a global notion of time shared by all clusters.

2.2.3 k1b cores

All the 288 cores of the KALRAY MPPA®-256 are designed upon the same *k1b* architecture which implements a 5-issue *Very Long Instruction Word* (or *VLIW*) ISA with a 7-stage instruction pipeline, running at 600MHz by default.

Arithmetics The k1b cores are capable of 32 and 64 bits arithmetics on integers and also feature several *Floating-Point Units* for multiply-add instructions on single and double precision IEEE754-2008 floating point numbers.

Execution modes The k1b cores can run in different execution modes to support the utilization of an Operating System. This mode can be either *privilege* or *user* mode. Usually, applications should run in user mode. Transitions to privilege mode should occur only on hardware traps, interrupts or system calls and should be treated by the OS.

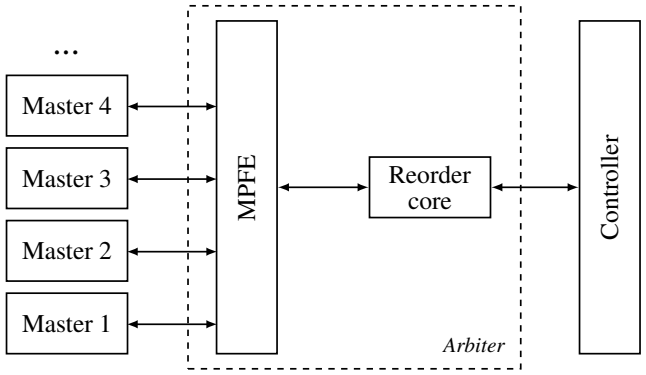


Figure 2.5: DDR-SDRAM arbiter of the KALRAY MPPA®-256

MMU As mentioned in Section 2.2.2.2, each k1b core also features an MMU to enable virtual addressing, ensure memory protection and enforce specific caching policies on memory pages. The MMU includes two TLBs: a fully-associative one of 8 entries denoted as the *Locked TLB* (or *LTLB*); and a 2-way set associative one of 128 entries denoted as the *Join TLB* (or *JTLB*). The TLB configuration supports arbitrary 2^n-sized memory pages.

2.2.4 I/O clusters

Each I/O cluster features: four RMs forming a Quad-core, a single bank of 2MiB of SRAM, four DMAs and several I/O peripherals.

Thanks to their privileged access to the two DDR-SDRAM controllers, the north and south I/O clusters are often denoted as the *IODDR*. Similarly, the east and west clusters are named *IOETH* because of their proximity with the Ethernet controllers. In the *Bostan* version of the KALRAY MPPA®-256, the north and east I/O clusters are grouped in an *I/O subsystem* (similar for south and west I/O clusters) which enables tighter coupling of resources. For example, in an I/O subsystem, the RMs of one cluster can target directly the SRAM bank of the other cluster.

2.2.4.1 Quad-core RM

In each I/O cluster, the four RMs implement the k1b ISA. Each RM has a private instruction cache of 32KiB and the four RMs share a configurable data cache of 128KiB. The data cache that can be either shared or split in four private data caches of 32KiB.

2.2.4.2 DRAM subsystem

Both the north and the south I/O clusters feature a DRAM subsystem containing a DDR3-SDRAM controller and an arbiter.

Overview The DRAM subsystem can be accessed by 8 *masters* able to read and write in DDR through the 8 ports of the *Multi-Port Front-End* (or *MPFE*, Figure 2.5). Two ports are dedicated to handle requests from the 4 RMs and the DSUs of the IODDR and the IOETH cluster in the I/O subsystem. Two other ports are reserved for reads and writes from PCIe backend DMAs. The remaining four ports are reserved for the four DMAs of the IODDR.

Arbitration As depicted in Figure 2.5, the DRAM arbiter of the KALRAY MPPA®-256 is composed of two elements. The *Multi Port Front End* (or *MPFE*) serializes the requests issued

by the masters. Each master (or group of masters) is assigned a priority. In case of concurrent requests, the master with the highest priority will be given full access by the MPFE. Competing masters with the same priority will be arbitrated in Round-Robin. In addition, the MPFE features a *starvation counter* that measures the time during which a request has been pending because of higher priority requests. When this counter reaches a pre-defined level, the pending master is given the maximum priority, thus avoiding starvation.

The second element of the KALRAY MPPA®-256's DRAM arbiter is the *Reorder Core*. It receives requests forwarded by the MPFE and stores them in a pool of $N_{req}^{pool} = 8$ elements. Then, it applies the following four rules to decide the order in which the requests will be issued to the controller.

Rule 1 Requests from high priority masters go first;

Rule 2 Requests on active rows have priority on request targeting non-active rows;

Rule 3 Requests targeting a recently opened page have low priority during t_{RC};

Rule 4 *RD* requests have priority over *WR* if previous request was a *RD* and vice versa.

Every time a request is issued to the controller, the reorder core accepts a new request from the MPFE and re-computes the order of requests with the same four rules.

Example 2 (Reordering of DRAM requests)**.** *We assume ascending priorities. The following requests are present in the reordering pool:*

Req. 1 RD *of priority 7 to a non-active row in bank 0;*

Req. 2 WR *of priority 4 to an active row in bank 1;*

Req. 3 RD *of priority 4 to a non-active row in bank 0;*

Req. 4 RD *of priority 4 to a non-active row in bank 1;*

Req. 5 WR *of priority 7 to an active row in bank 2;*

Req. 6 WR *of priority 7 to a non-active row in bank 1;*

Req. 7 WR *of priority 4 to an active row in bank 3;*

Assuming the four rules of the reorder core, the requests will be served in the following order:

Req. 5 *wins Rule 1 with Req. 1 and Req. 6 and wins Rule 2;*

Req. 6 *wins Rule 1 with Req. 1 and wins Rule 4;*

Req. 1 *wins Rule 1;*

Req. 7 *wins Rule 2 (row of Req. 2 has been closed by Req. 6);*

Req. 3 *wins Rule 3 (bank 0 is the least recently opened);*

Req. 4 *wins Rule 4;*

Req. 2 *last request.*

We will provide further information on the architecture of DRAM subsystems and the methods that have been applied to used them in a predictable manner in Section 3.2.2.

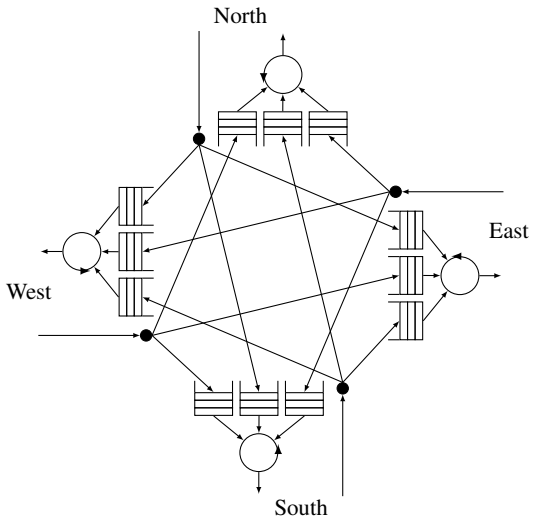


Figure 2.6: NoC switch of the KALRAY MPPA®-256. The local interface is not depicted for clarity.

2.2.5 On-chip networks

The KALRAY MPPA®-256 features two parallel NoCs implementing the same 2D-torus topology shown in Figure 2.2. The *Data-NoC* (or *DNoC*) is dedicated for the exchange of large data. The *Control-NoC* or *CNoC* is used to send small control messages to achieve inter-cluster synchronization for example. Both NoCs are wormhole-switched, source-routed, 32 bits wide and running at 600MHz by default.

2.2.5.1 Switches

All the NoC switches are identical (Figure 2.6). The switches have 5 interfaces: North, South, East, West and Local. Each interface has four buffers to store packets received by any of the four other interfaces. In case some flows are conflicting to access an output interface, they are arbitrated in Round-Robin at packet level. Since output buffers are not shared, two flows crossing a switch without interface conflicts will not interfere.

2.2.5.2 Routing

The route that will be followed by a packet is fully defined by software prior to emission. The default header of a DNoC packet includes 43-bits field to store a route. It contains a concatenation of directions encoded on two bits (`0x0` to go North, `0x1` to go South, `0x2` to go East, `0x3` to go West) starting from LSB.

The final destination node of the packet is encoded into the route. The packet reaches the final node when the next direction that it should take is the direction from which it entered the switch. For example, a packet arriving from the east interface is considered as arrived at destination if its next direction is east.

Example 3 (Routes on the KALRAY MPPA®-256)**.** *A packet following the route:*

```
0x2DA = 0b1011011010
```

goes east, east, south, west and gets delivered.

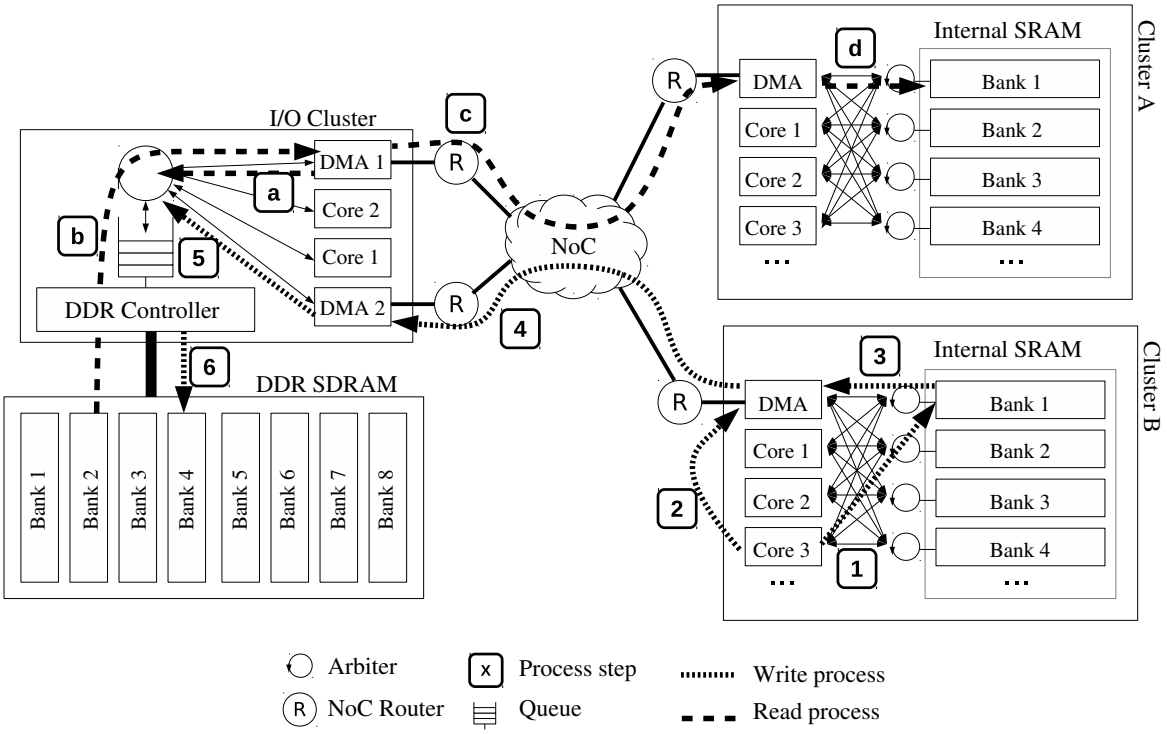


Figure 2.7: Example of remote memory operations on a simplified model of the KALRAY MPPA®-256

The NoC also implements additional features to handle multicast (the packet is delivered to *some* of the nodes on its route) and broadcast (the packet is delivered to *all* nodes on its route) modes. Packet headers can also be extended to account for routes not fitting into 43 bits, up to 104 bits.

2.2.6 Remote memory operations

In this section, we provide further details about the *remote memory operations* on the KALRAY MPPA®-256. A *remote write* corresponds to the process of writing data into the memory (either local or external) of a remote cluster. Symmetrically, a *remote read* corresponds to the process of reading data from a distant cluster. We provide two examples of remote reads and writes targeting the external DDR3-SDRAM memory as shown in Figure 2.7. One may note that remote reads and writes targeting local SRAM of a remote cluster can be achieved with similar processes.

Example 4 (Remote write)**.** *In this example, the core 3 of cluster B requires to write data in the bank 4 of the external DDR3-SDRAM memory. We detail each step of this write process as shown in figure 2.7 (the* Write process *is numbered with* numbers *and depicted with dotted arrows in figure 2.7):*

1. *The requesting computing core (core 3 of cluster B) writes the data to be sent in the local memory of the compute cluster. As the banks of the local SRAM are shared among many potential requesters, there may be an arbitration at this level in the case of concurrent accesses to the bank.*

2. *The core signals to the local DMA its intention to send the data.*

3. *DMA reads the data (written by core 3 at step 1) from the local memory. Once again, any concurrent access to the same bank will involve an arbitration.*

4. *DMA sends the data through the NoC. If the amount of data to send is important, it will be split in several packets constituting a flow. All the packets will cross the NoC following the same path. If one or several parts of this NoC path are shared with other NoC flows, the arbitration between the flows will occur at packet level.*

5. *The sink DMA (DMA 2 of the I/O cluster) receives the packets and initiates DDR3-SDRAM write transactions. If other masters (I/O cores, other DMAs, ...) access the external memory concurrently, an arbitration process will occur. This phase assumes that the sink DMA has been configured before reception to associate one of its reception queues to a specific DDR3-SDRAM address (an address in bank 4 of the DDR3-SDRAM here).*

6. *Once the sink DMA write(s) request(s) is/are elected by the memory arbiter, data is written into the DDR3-SDRAM array.*

Example 5 (Remote read)**.** *In this example, the core 2 of cluster A needs to read data from the bank 2 of the DDR-SDRAM to store it in the bank 1 of its local memory. We detail each step of this read process as shown in figure 2.7 (the* Read process *is numbered with* letters *and dashed arrows in figure 2.7):*

a. *The DMA of the I/O cluster initiates a DDR3-SDRAM read transaction. Once again, any concurrent access to the memory with any other master will involve arbitration.*

b. *Once the DMA command is elected, data is transferred from DDR3-SDRAM to the DMA.*

c. *DMA sends packets through the NoC. Again, important amounts of data are packetized and arbitrated with concurrent flows at packet level.*

d. *DMA of the compute cluster receives the packets and attempts to write them back into the local memory. Again, we assume that this DMA has been pre-configured to associate one of its reception queue to a specific memory area of the local memory (the bank 1 in this example).*

We remark that a read process is fairly equivalent to a write process. Indeed, a read by a computing core is equivalent to a write from an IO cluster. The difference is that the destination of the data is not the external memory but the internal memory of a Compute cluster.

In this example, the phase a. *of the read process is initiated by the DMA of the I/O cluster. It is assumed here that the DMA is either responding to a read request that was previously sent by the requesting cluster or started automatically following a scheduling table computed off-line.*

2.2.7 Compatibility with a safety-critical context

The KALRAY MPPA®-256 can be considered as a good COTS target candidate for the design of hard-real time systems for several reasons listed below.

Low energy consumption Although saving power is obviously beneficial to reduce fuel consumption in an aircraft, it also greatly helps the design of certifiable electronics cards. Indeed, it

appears that, under specific conditions, the power consumption of the KALRAY MPPA®-256 is probably sufficiently low to cool with purely passive systems. Being able to avoid active cooling dramatically simplifies the design of electronics cards that are robust to vibrations constraints.

Timing-compositionality The k1b cores exhibit the property of *full timing-compositionality*, that, as we will further explain in Section 3.1, greatly simplifies the computation of the safe WCETs required for certification.

High programmability On the KALRAY MPPA®-256, very few operations are achieved implicitly - the hardware does not support cache coherency and messages are sent on the NoC explicitly. Although this specificity does not simplify programming, it does help in mastering the platform with software. In addition, NoC communications are heavily configurable using custom DMA micro-codes and the source routing mechanism. So, many different communication patterns can be implemented in a very flexible but still controlled way.

Information availability The willingness of KALRAY to share extensive information regarding their architecture's details with clients is a significant asset from an industrial point of view. Indeed, as we will further explain in Section 3.1, the common industrial practice of computing WCET using *static analysis* requires fine-grained models of the platform. Being able to obtain the informations to construct these models is thus of major importance.

2.3 Additional constraints

In an industrial environment, two criteria are of particular interest: cost and performance. Designing inexpensive avionics systems usually implies strong design constraints on the choice of hardware components, on the architecture of systems, and on the certification of software. As an example, current industrial practices privilege COTS over bespoke hardware components in many situations for both cost and performance reasons. Designing and producing *powerful* custom-made processors requires important investments. In contrary, choosing COTS can dramatically reduce hardware design costs and narrow the design lead time overall. As a consequence, we will focus in this thesis on the challenges for designing software managing the caveats of a COTS platform such as the KALRAY MPPA®-256. The architecture and implementation of such software are detailed in Chapters 5 and 7.

Constraint 1. *COTS processors must be used for performance.*

In safety critical systems, and in particular in avionics, industrial standards, such as the DO-178C [4], impose the computation of safe upper-bounds on the *Worst-Case Execution Time* (or *WCET*) of software programs. Computing such bounds can be challenging [18] because execution times vary depending on input parameters and current hardware states. Moreover, the WCET estimation problem is becoming increasingly harder with the emergence of multi and many-core processors [19]. This is due to more complex internal behaviours, shared resources and interferences. The challenge is thus twofold: 1) since modern architecture are particularly complex to analyze, producing *safe* WCET estimations appears especially difficult; and 2) WCET estimations must be *tight* to avoid too much resource over-provisioning that eventually leads to poor performance. In this context, the major constraint is to design systems that can be analyzed in order to *make WCET computation feasible*. In Chapter 3, we will overview the classical methods to estimate WCET on modern multi- and many-core platforms and how systems can be designed in order to be make their analysis practically doable.

Constraint 2. *Computing WCET of program using static analysis techniques must be practically feasible.*

Industrial practices imply that off-line scheduling is often preferred to on-line scheduling when dealing with safety-critical applications. The reason for this is twofold. Firstly, the execution of off-line scheduled tasks can be achieved using simple scheduling tables. Since this requires a minimal run-time support, the certification cost of the system software can be kept low. And secondly, mapping tasks off-line simplifies the overall timing analysis of the system by avoid context-switching costs or preemption-related overheads. For this reason, we will focus in this thesis on the problems related to the off-line scheduling of safety-critical application only. We propose in Chapter 8 an off-line approach for mapping hard-time applications on the KALRAY MPPA®-256.

Constraint 3. *Scheduling of tasks must be achieved off-line.*

Finally, modern avionics applications are massive. Managing such applications requires approaches not only providing good results on small examples but also on industrial-sized problem instances. For this reason, our goal in this thesis will be not only to find solutions to the different problems related to mapping and scheduling but also to find solutions having a direct practical interest to solve current industrial problems. Consequently we will evaluate the scalability of the mapping technique detailed in Chapter 8 over a real avionics application from Airbus.

Constraint 4. *All mapping and scheduling techniques must scale to industrial-sized applications.*

2.4 Summary

In this chapter, we have detailed the context of the thesis, its consequences on the input data of the various programming and scheduling problems to be solved, as well as the methods that can be envisaged to do so. We consider real avionics applications modeled by multi-rate parallel tasks with explicit data exchanges. Our goal will be to execute these applications on the KALRAY MPPA®-256 that can arguably be considered as a good candidate for the design of future avionics computers. The industrial context will limit our investigations to off-line mapping techniques in order to have better control of the target and to keep the certification costs as low as possible. Finally, since we aim at solving industrial problems, we will have a major requirement regarding the scalability of our solutions in order to deal with applications of realistic sizes as shown in Chapter 8.

Part II

State of the Art

Chapter 3

Bounding execution times

Contents

3.1 WCET estimation techniques 43
3.1.1 Static analysis 44
3.1.2 Measurement-based techniques 46
3.1.3 Moving toward many-core processors 46
3.2 Shared resources on many-core processors 48
3.2.1 On-chip networks 48
3.2.2 DRAM subsystem 49
3.3 Interference penalties on many-core processors 55
3.3.1 Network on Chip 55
3.3.2 Temporal bound for DRAM subsystems 59
3.3.3 End-to-end timing bounds on multiple resources 60
3.4 Enforcing time-predictability by design 60
3.4.1 Custom-made time-predictable hardware 60
3.4.2 Software-enforced predictability on COTS 64
3.5 Summary 69

In order to fulfil constraints 1 and 2, we will analyze the existing methods that enable safe upper-bounding of execution times on modern platforms througout this chapter. Section 3.1 firstly reviews the classical techniques of WCET estimation and details the properties of *time-compositionality* and *time-composability* that are desirable to properly take into account interferences on shared resources within multi and many-core processors. Secondly, Section 3.2 describes the analytical methods that have been proposed to compute *interference penalties* for the most sensitive shared resources on many-core platforms; namely Networks on Chip and DDR-SDRAM subsystems. Finally, Section 3.4 details the main two approaches in the design of WCET-analyzable systems thanks to either custom-made time-predictable hardware or interference-aware bespoke control software on COTS.

3.1 WCET estimation techniques

Definition 1 (Worst-Case Execution Time)**.** *The Worst-Case Execution Time of a software program is the uppermost duration required for its complete execution under any input configurations and any initial hardware states.*

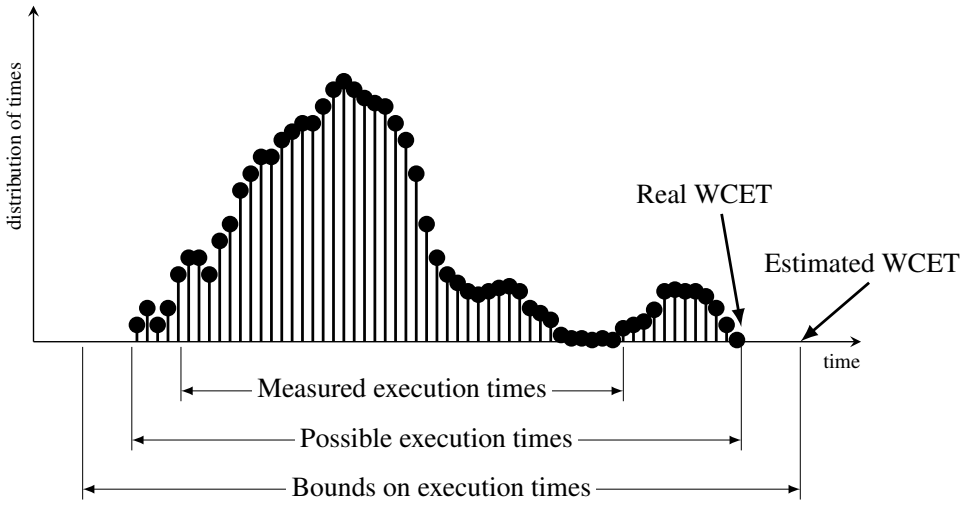


Figure 3.1: Variability on the execution times of a program

The problem of computing, or at least safely upper-bounding, the WCET of a sequential software program is a complex research topic. In [18], Wilhelm *et al.* propose an overview of the techniques and the available tools to compute WCETs. The techniques proposed in the literature are usually classified under one of the following categories:

- *static analysis* : abstract models of both the program and the processor are used and analyzed to compute (safe) bounds on the WCET.
- *measurement-based techniques* : the WCET bound is derived from measurement of the actual execution times of the program on the real target or a simulator under representatives test conditions.

Figure 3.1 shows the classical distribution of execution times of a program. In general, the WCET estimated using static analysis upper-bounds the real WCET whereas the largest observed execution time necessarily under-estimates it. In both cases, the methods rely on a three-steps procedure:

1. Find all the possible execution paths of the program. Each path is determined by a set of input values of the program that will impact the conditional branches in the tasks code.
2. Determine the execution time of all possible execution paths. Since this execution time depends on the architecture of the target on which the code runs, the traits of the hardware must be taken into account.
3. Find the largest execution time among all possible execution paths.

In the following two sections, we will illustrate the application of this procedure for both static analysis and measurement-based techniques.

3.1.1 Static analysis

Figure 3.2 shows the three-steps procedure for computing WCET with static analysis techniques. Firstly, an abstract representation of the program, usually denoted as the *Control Flow Graph* (or

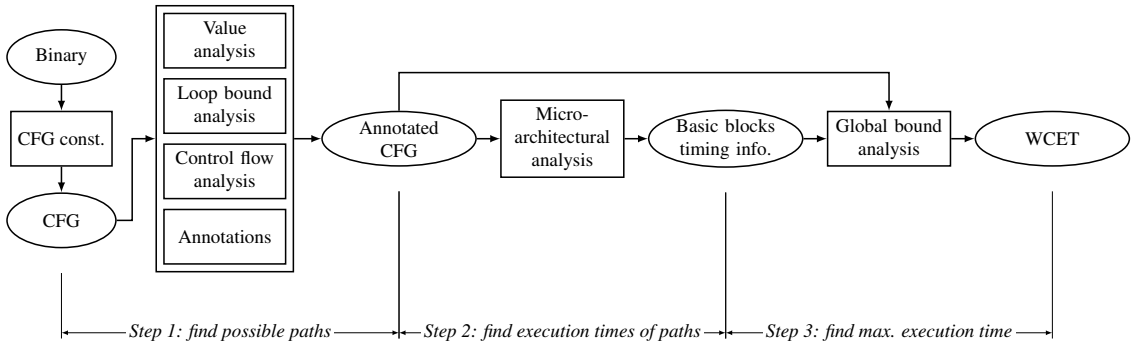


Figure 3.2: WCET estimation procedure using static analysis techniques

CFG), is extracted from the binary code. The CFG's nodes represent sequential pieces of code also denoted as the *basic blocks*. The edges represent the jumps and the branches between the basic blocks.

Remark 1. *Usually, the choice of representing the program by its CFG rather than its* Abstract Syntax Tree *(or* AST*) is motivated by the difficulty of linking the nodes of the AST with actual sequences of instructions in the binary because of the compiler transformations.*

Step 1: Once obtained, the CFG is enhanced by three kinds of flow information. Firstly, the *value analysis* computes the ranges for the processor registers and the local variables at all points of the CFG. Secondly, these ranges are used during the *loop bound analysis* and the *control flow analysis* in order to respectively find bounds on the loops of the program and remove all the paths that are practically unfeasible. Finally, the user may manually add *annotations* to help the WCET computation by explicitly setting a loop bound for example.

Step 2: The purpose of the *micro-architectural analysis* is to determine the execution times of the basic blocks. To do so, an accurate model of the hardware platform is required. The main hardware mechanisms of the processor such as the instruction pipeline or the caches need to be precisely analyzed to compute realistic timing values. The feasibility of such an analysis is obviously limited to processors where low-level details on the architecture are made available by the chip manufacturer. Moreover, the hardware should exhibit good temporal properties such as predictable cache replacement policies [20] or the absence of timing anomalies [21] in order to make the problem tractable.

Step 3: Finally, the WCET of the program is the longest path in the CFG. Since the number of possible execution paths in a realistically sized program can be very large, their exhaustive enumeration is often impossible in practice. In order to tackle this issue, several approaches, such as the *Implicit Path Enumeration Technique* (or *IPET*) [22], have been proposed in literature. With IPET (that is probably the most used technique today), the flow constraints and the timing information are expressed using *Integer Linear Programming* (or *ILP*). Further details on other *global bound analysis* techniques can be found in [18].

Several available commercial and academic tools such as aiT [23], Bound-T [24], OTAWA [25] or Heptane [26] use static analysis to compute bounds on WCETs.

3.1.2 Measurement-based techniques

The confidence on the WCET estimation produced when using a measurement-based technique highly depends on the coverage of possible execution paths during the experiments. In order to ensure that a program is run on all its potential execution paths, one may consider to execute it for all possible input configurations. However, assuming integer, or even floating point numbers that are defined on large ranges as input data, the exhaustive enumeration of all combinations will lead to a prohibitive number of test cases, making this approach inapplicable for real-sized problems. Several approaches in the literature have been proposed to tackle this problem by using instrumented code [27], leveraging simulation means [28] or optimizing the test-cases coverage [29]. Still, all these approaches remain either too complex for practical usage or simply sub-optimal. Thus, the bounds on the WCET eventually produced by such methods are often not considered as safe.

The measurement of execution times on the real target is often argued to enable the production of WCET estimates without needing a fine grain understanding of the low-level mechanisms of the hardware. However, the problem of measuring representative execution times of all program paths lies in the capability of covering the possible initial hardware states during the experiments. Moreover, the need of a real hardware target to produce estimates can be an issue when the WCETs of programs are required in the early stages of a project. These drawbacks can both be mitigated thanks to simulation means. However, the development of a cycle-accurate simulator involves heavy costs and requires a perfect understanding of the temporal behavior of all components of the processor architecture, thus breaking the benefit of the black-box approach.

Usually, the lack of confidence in the WCET bounds produced when using measurement-based techniques is answered by the addition of arbitrary safety margins, often chosen following the system designer's experience. To the best of our knowledge, the only commercially-used tool for WCET estimation that uses measurement-based technique is Rapitime distributed by Rapita Ltd [30].

Recently, probabilistic methods [31] have been proposed to derive WCETs bounds from measurements with a more solid mathematical argumentation. The idea is to model the execution times of a program as a distribution and to use the *Extreme Value Theory* (or *EVT*) [32] to extrapolate information on the distribution's tail. However, the applicability of EVT is conditioned by assumptions on the statistical independence and the continuity of the input data. In [33], Lu *et al.* questioned the correctness of these assumptions in the context of real-time embedded systems. As an answer, current approaches such as those proposed in the PROARTIS [34] and PROXIMA [35] projects have laid the foundations for randomized hardware and/or software [36, 37].

The promising probabilistic approaches still require further efforts to demonstrate their ability to robustify the measured WCETs, thus leaving the static analysis as the main solution for the design of industrial time-critical systems.

3.1.3 Moving toward many-core processors

The complex features of modern microprocessors often enlarge the state space that should be explored for producing safe WCET estimates using static analysis. Moreover, the emergence of multi- and many-core processors brings the additional problem of taking into account the *interferences* that tasks may suffer when accessing shared resources such as a shared bus, a shared cache or a shared DRAM subsystem [19]. An exhaustive exploration of the state space in this situation requires to keep traces of all potential interferences at the cost of a much larger effort in analysis. So, current state-of-the-art approaches derive the methods applied on mono-core processors to multi-

and many-core by assuming desirable timing properties on the system under analysis to make the problem tractable. Overall, these approaches rely on *time-compositional* or *time-composable* hypotheses.

With *time-compositional* [38] systems, the analysis of the program execution is decoupled from the process of accounting for the interferences.

Definition 2 (Timing-compositionality)**.** *A system is said to have the property of timing-compositionality if its temporal behavior can be inferred from the individual temporal behavior of all its components.*

An example of timing compositionality can be found in Atanassov *et al.* [39]. Indeed, the authors assume the property of timing compositionality when they explain that the WCET of a program accessing a DRAM can be obtained by summing the WCET of the program without considering refreshes to the maximum refresh-related delays.

More generally, the timing verification of timing-compositional systems usually occurs in two steps. Firstly, each component in the system is analyzed in isolation. It means for example that the WCET of each task is estimated without considering any interferences from co-running tasks. Then, the worst-case scenario of interferences is analyzed and an appropriate *timing penalty* is deduced for each task and applied on its WCET in isolation.

Definition 3 (Timing penalty)**.** *A timing penalty is a bound on the maximum interference-related delay that a program can suffer when accessing a shared resource.*

The inherent assumption that the sum of the WCET in isolation and its according timing penalties can be considered as a sound on the real WCET is directly provided by the property of timing compositionality.

In an orthogonal manner, *time-composable* [38] systems are composed of components whose timing behaviour are not temporally correlated, despite potentially shared resources.

Definition 4 (Timing-composability)**.** *A system has the property of timing-composability if the temporal behavior of each of its components does not depend on the behavior of other components.*

A classical example of a time-composable system is a shared bus accessed in a *Time Division Multiplexing* (or *TDM*) fashion where each bus master is assigned a private periodic slot during which it has full access to the bus. In this example, each bus master will behave invariably no matter how the other slots are used.

In general, the timing verification of time-compositional systems mainly occurs in final stages of implementation where the contribution of each component is known and can be used to deduce all the individual penalties that may, or may not, eventually satisfy the timing requirement. On the other hand, time-composable systems usually rely on resource reservation techniques that can be decided in the early steps of the implementation but usually lead to non work-conserving systems. For many-core processors, the problem of interferences between co-running tasks is solved with assumptions on the timing properties of the system. We discuss the accounting of interferences on time compositional shared resources of many-core processors in section 3.3 and the design opportunities to enforce either timing-compositionality or timing composability through bespoke hardware or controlled software in section 3.4.

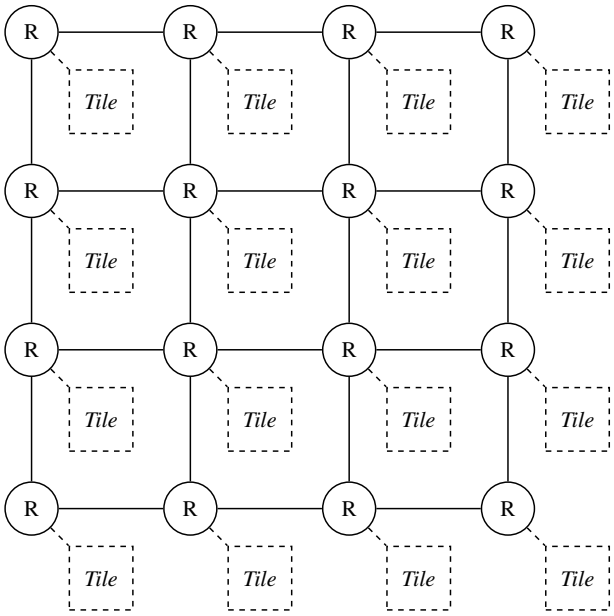


Figure 3.3: Example of a 4x4 tiled many-core processor with a 2D-mesh Network on Chip.

3.2 Shared resources on many-core processors

In many-core processors, the management of shared resources is of major importance. The large number of cores implies that some resources are massively shared. Being able to compute tight and safe interference penalties for tasks accessing them is thus needed to avoid excessive pessimism in the WCET estimation. In many-core processors, the NoCs and the DRAM subsystems are the resources that are shared among the largest number of requestors. Their management is thus especially important. In the next sections, we will overview the basics on NoCs and DDR-SDRAM.

3.2.1 On-chip networks

Modern many-core processors essentially differ from multi-core processors in their interconnect technology. Where multi-core chips usually feature buses or crossbars, many-core processors integrate a Network-on-Chip allowing *tiled* designs offering better scalability [9, 7]. Indeed, NoCs enable point-to-point communications between processing elements and also I/O or RAM peripherals. Since this technology allows several communications to occur at the same time, it improves the degree of parallelism over bus-based approaches.

In NoCs, just like in classical networks, data is sent in packets that are routed through one or several network equipments before arriving at destination. A packet is split into several atomic chunks of data called *Flow control digits* (or *flits*). In general, the payload of a packet can be either a message exchanged between two processing elements or data coming/going from/to I/O peripherals and memory. Yet, some existing COTS many-core processors, such as the Mellanox Tile-Gx* [15], feature several specialized NoCs to avoid mixing traffics of different types.

NoCs can be designed upon a variety of different topologies among which 2D-meshes (example in Figure 3.3) like in the Tile-Gx* [15] or the Intel SCC [14] and 2D-toruses like in the KALRAY MPPA®-256 (example in Figure 2.2) appear to be most used in COTS processors. Existing NoCs usually implement virtual cut-through [40] or wormhole-switching [41] as flow-control strategy [42]; the latter being the most popular over existing COTS. This is essentially because of

the small-sized buffers in wormhole routers. It allows an implementation of the NoC that is both area and power efficient. Details on switching strategy can be found in [43, Appendix F].

Example 6 (Wormhole switching)**.** *Figure 3.4 shows the example of two packets of four flits crossing a wormhole-switched network. The first packet P_1 is sent by S_1 at destination of D_1. The second packet P_2 is sent from S_2 at destination of D_2. From cycle $t = 2$ to $t = 6$, P_1 is stalled while P_2 is transmitted through the shared interface of the bottom-left switch. Then, both packets progress at each cycle until they are delivered.*

Since the NoC is a shared resource, its timing behaviour is of main interest for the design of a many-core-based real-time computer. The notion of *Worst-Case Traversal Time* (or *WCTT*) upper-bounds the maximal latency that a message can suffer when going through the network. WCTTs of packets are often required to bound WCETs of programs communicating on the NoC. For example, programs may stall while waiting for the reception of a NoC packet. In this case, upper-bounding the stalling delay of the program involves upper-bounding the WCTT of the packet. The problem of computing sounds and tights WCTTs on wormhole-switched NoCs is an active research topic [44, 45, 46, 47]. We will explore in Section 3.3.1 the methods for providing timing guarantees on 2D-torus wormhole switched networks.

3.2.2 DRAM subsystem

The *Dynamic Random Access Memory* (or *DRAM*) is a simple, cheap and compact type of memory that is widely used in modern computers to store code and data of running programs. In many-core processors, the DRAM is massively shared by a number of cores. The KALRAY MPPA®-256 for example features 2 DDR-SDRAM controllers that are shared among 288 cores. In this context, the management of DRAM accesses is critical for both performance and predictability.

In DRAM, data are stored in capacitive *cells* organized in 2D-arrays usually called *banks* as shown on Figure 3.5. The data stored in the cells is made available to other components through a data bus thanks to bidirectional *Sense Amplifiers*. By design, the sense amplifiers can be connected to only one row of cells at a time. Moreover, the width of one row of cell is usually a lot larger than the width of the data bus, and so, the rows are split into smaller columns on which the operations of reading or writing data can be issued.

In order to manage this complexity, the DRAM chips obey to *commands* that are sent to them by a master *DRAM controller*. For example, in order to access a specific memory address, the DRAM controller will issue a complex sequence of commands to connect the sense amplifiers to the correct row with *Activate* (or *ACT*) and *Precharge* (or *PRE*) commands and to operate on the specific column using *Read* (or *RD*) or *Write* (or *WR*) commands. Figure 3.6 depicts the possible sequences of commands that can be issued to a DRAM bank.

In addition to the RD, WR, ACT and PRE, the controller must periodically issue *Refresh* (or *REF*) commands to avoid the corruption of data. Indeed, the capacitive nature of DRAM cells is subject to *leakage current* which deteriorates the voltage stored in the capacitors. The voltages are kept clearly identifiable as a logical 0 or 1 by *refreshing* the capacitors' voltages fast enough to never enter an indistinguishable 0 or 1 state.

For physical reasons, complex timing constraints are imposed on a series of consecutive commands to ensure the correct functioning of a DRAM system. These constraints (detailed in the JEDEC standard [48]) must be respected by the DRAM controller at runtime. Assuming legal sequences [1] of commands, we provide to the reader an overview of the major timing constraints that can be found in the JEDEC standard.

[1] By legal sequences, we mean sequences that respect the state machine of Figure 3.6.

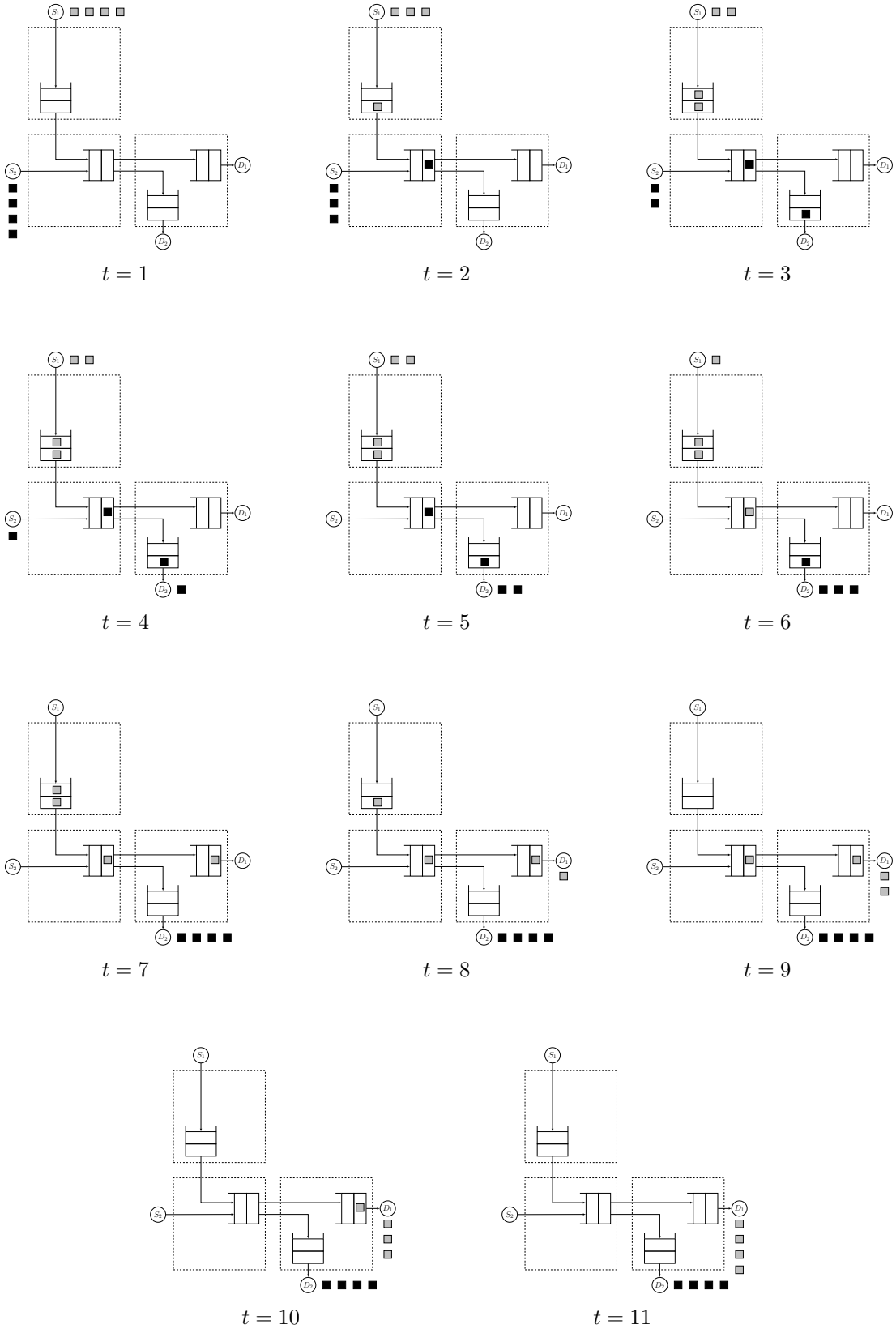


Figure 3.4: Example of two packets of four flits crossing a wormhole-switched network

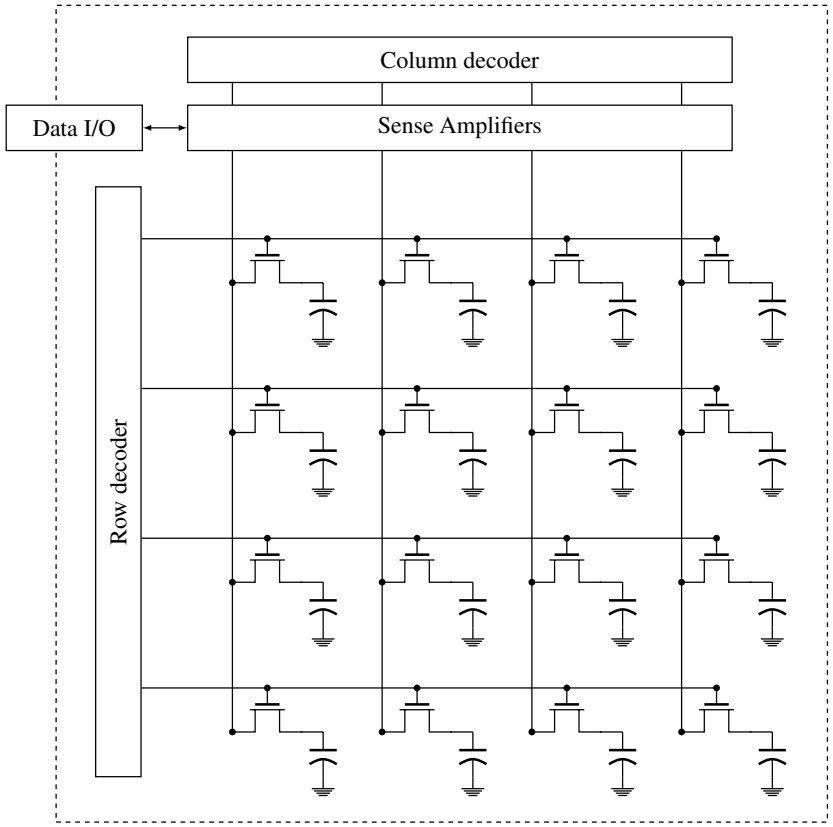


Figure 3.5: Architecture of a DRAM bank.

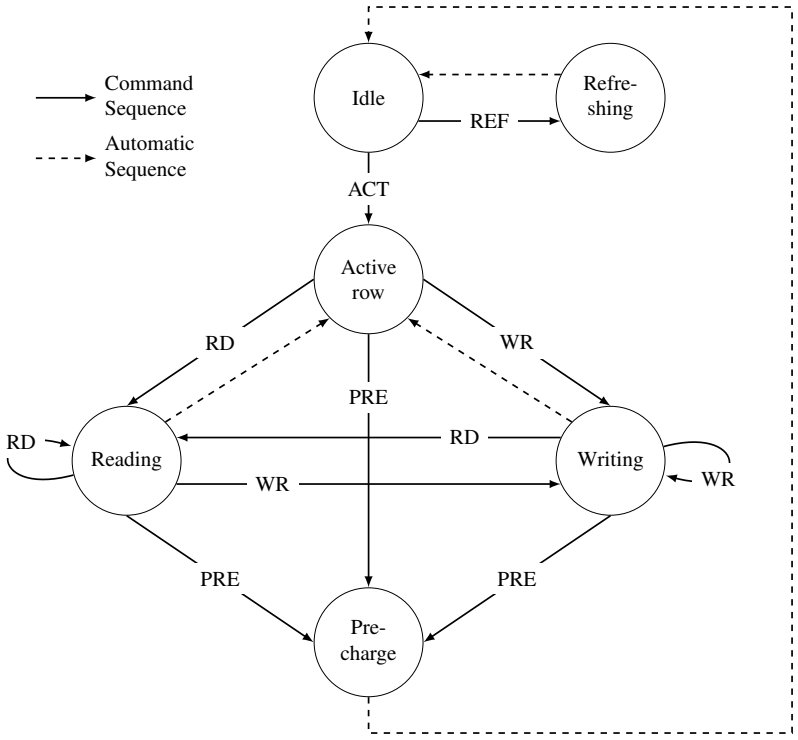


Figure 3.6: Simplified state machine of a DRAM bank access protocol.

ACT The row activate command is associated to four major timing constraints:

- t_{RCD} Row to Column Delay. This parameter represents the time physically required to connect the sense amplifier to the row. The memory controller must wait during at least t_{RCD} cycles after the ACT before it can issue a RD or a WR.
- t_{RAS} Row Access Strobe. When the sense amplifiers are connected to a row, they slightly modify the voltage of the storage capacitors of the cells. So, just after the connection, the sense amplifiers *restore* the voltages of the capacitors in order to avoid any corruption on the data. t_{RAS} is the time required by the sense amplifiers for this restoration. For this reason, t_{RAS} is the minimum time a row must remain opened before it can be precharged.
- t_{RRD} Row activate to Row activate Delay. It is the minimum time required between two ACT commands. This allows to limit peak current.
- t_{FAW} Four row Activation Window. It is a sliding window during which no more than 4 ACT commands can be issued in order to limit peak current.

PRE The precharge command is associated to two major timing constraints:

- t_{RP} Row Precharge delay. It is the time required to disconnect from the current row and reset the sense amplifiers to a neutral voltage level.
- t_{RC} Row Cycle. $t_{RC} = t_{RAS} + t_{RP}$ is a commonly used indicator for DDR SDRAM performance as it is the minimum time allowed between two ACT can be issued to the same bank.

RD The read command is associated to two major timing constraints:

- t_{CAS} Column Access Strobe. It is the duration required by the memory to place on the data bus the requested data. This parameter is also often noted t_{CL}.
- t_{burst} It is the time (in cycles) required to transfer a complete burst. If the memory data bus is w_{bus} bytes large, a complete burst will be transferred in s_{burst}/w_{bus} beats of data. In DDR SDRAM systems, the double data rate mechanism allows to transfer two beats of data by cycle. So, $t_{burst} = s_{burst}/(2 \times w_{bus})$ cycles.

WR The Write command is associated to four major timing constraints

- t_{burst} Same as RD.
- t_{CWD} Column Write Delay. Delay between the WR command and the placement of data on the bus.
- t_{WTR} Write To Read delay. Minimum time between a WR and a RD command. This constraint is related to the bus switching time. t_{WTR} is not local to a bank but a global device constraint.
- t_{WR} Write Recovery delay. This is the time between the end of the data burst and the complete propagation of the data in the DRAM array. So, t_{WR} is the minimum amount of time to wait after a column write command before a precharge command can be issued.

REF The Refresh command is associated to two major timing constraints:

- t_{REFI} Refresh interval. It is the time between two REF commands issued by the controller. So, $t_{REFI} = T_{refresh}/N_{refresh}$ (usually 7.8μs or 3.9μs depending on the operating temperature).

Parameter	Nanoseconds	Cycles	Data beats
t_{CK}	1.25	1	2
t_{BURST}	5	4	8
t_{CAS}	13.75	11	22
t_{RP}	13.75	11	22
t_{RCD}	13.75	11	22
t_{WR}	21.25	17	34
t_{WTR}	7.5	6	12
t_{RAS}	35	28	56
t_{RC}	48.75	39	78
t_{FAW}	30	24	48
t_{RRD}	6.25	5	10
t_{CWD}	10	8	16
t_{RFC}	260	208	416
t_{REFI}	3906	3125	6250

Table 3.1: Example of timing constraints extracted from the Micron 4GB MT41K512M8-125 DDR3 SDRAM module documentation [49].

t_{RFC} Refresh Cycle. It is the time required to refresh N_{row}^{ref} rows. This parameter depends on the memory density. It can be (over-)approximated by $N_{row}^{ref} \times t_{RC}$.

In order to give to the reader the order of magnitude of each previously enumerated timing parameters, we provide in table 3.1 an example of timing constraints extracted from the documentation of a Micron DDR3-SDRAM module [49].

The sense amplifiers are commonly seen as *row-buffers*[2], in which requests can be issued efficiently. Similarly to classical caches, the memory requests can thus be classified as *row hits* when the targeted data is present into the row-buffer or *row-misses* when the request targets a closed row. In modern DRAM controller, the management of such row buffers usually implements one of the two following policies:

- *open-page*: *PRE* commands are issued only in case of a row-miss of a refresh, thus exploiting the spatial and temporal locality principle. In general, the open-page policy performs best with high row-hit ratios.
- *close-page*: every *RD* or *WR* command is systematically followed by a *PRE* command. This performs best under high row-miss ratios when accesses mostly target random locations in memory.

Example 7 (Row-buffer management policies)**.** *Table 3.2 shows an example of six memory requests and the associated sequences of commands in open-page and close-page row-buffer management policies. The Figure 3.7 shows the time diagram of commands with the associated timing constraints.*

Finding a tight upper-bound on the time required to read or write a data from/to a COTS DRAM system can be challenging because of the complexity of its access protocol. Indeed, the time required to access a data highly depends on the current DRAM state and is thus strongly dependent on

[2] Also denoted as the *row caches* in some papers

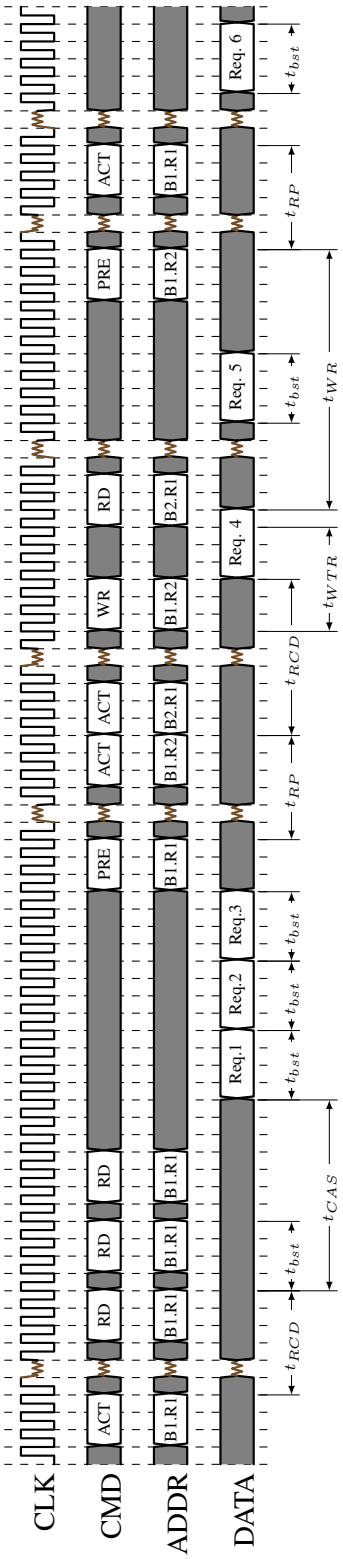


Figure 3.7: DRAM commands under open-page row-buffer management policies for requests of example 7

INPUT REQUEST				GENERATED COMMANDS	
Req. no.	Type	Bank	Row	Open-page	Close-page
1	Read	1	1	*ACT - RD*	*ACT - RD - PRE*
2	Read	1	1	*RD*	*ACT - RD - PRE*
3	Read	1	1	*RD*	*ACT - RD - PRE*
4	Write	1	2	*PRE - ACT - WR*	*ACT - WR - PRE*
5	Read	2	1	*ACT - RD*	*ACT - RD - PRE*
6	Read	1	1	*PRE - ACT - RD*	*ACT - RD - PRE*

Table 3.2: Example of DRAM commands issued under open-page and close-page row-buffer management policies

the history of requests. Moreover, modern multi and many-core processors usually feature several components able of addressing the DRAM directly. However, the DRAM controller still answers a series of requests that results from the arbitration of the concurrent accesses. The performance of the DRAM subsystem in a multi or many-core context is thus strongly correlated to the ability of the arbiter to issue the requests in a *favorable* order that will enable to hide most of the JEDEC timing constraints thank to pipelining effects.

In the next section, we will analyze the methods that have been proposed to compute bounds on the *Worst-Case Traversal Time* of NoC packets and the DDR-SDRAM access durations.

3.3 Interference penalties on many-core processors

Time-compositional approaches enable decoupled analysis of the program execution and the interferences on shared resources. Yet, the underlying assumption when applying such methods is that analyzing the shared resources interferences is possible. In this section, we overview the state-of-the-art methods for computing bounds on interference penalties when accessing NoCs and DRAM subsystems.

3.3.1 Network on Chip

3.3.1.1 Deadlocks

In wormhole switched networks, one message can be holding one resource while requesting others, and thus, cause a deadlock [50, 51]. The avoidance of deadlock is a major problem that must be solved before any of the WCTT estimation techniques that will be presented below can be applied.

Example 8 (Deadlock on a wormhole-switched NoC)**.** *Figure 3.8 depicts an example of deadlock in a wormhole-switched network. We can see the flits $F(R_X)$ of 4 packets in the FIFO queues of the interfaces of 4 routers. 3 out of 4 flits $F(R_1)$ at destination of the router R_1 went through the router R_3 and are stored in one queue of the router R_4 waiting for availability of the link to R_1. Because of the back-pressure mechanism, the fourth $F(R_1)$ flit is still queueing in R_3 as the queue of R_4 is full. It is also maintaining occupied the link between R_3 and R_4 as all the flits of one packet must be consecutive. At the same time, the flits $F(R_2)$ blocking the flits $F(R_1)$ are waiting for the link between R_1 and R_2 to become idle. Similarly, the flits $F(R_3)$ occupying this link are waiting for the link between R_2 and R_3 to become idle but this link is occupied by the $F(R_4)$ flits*

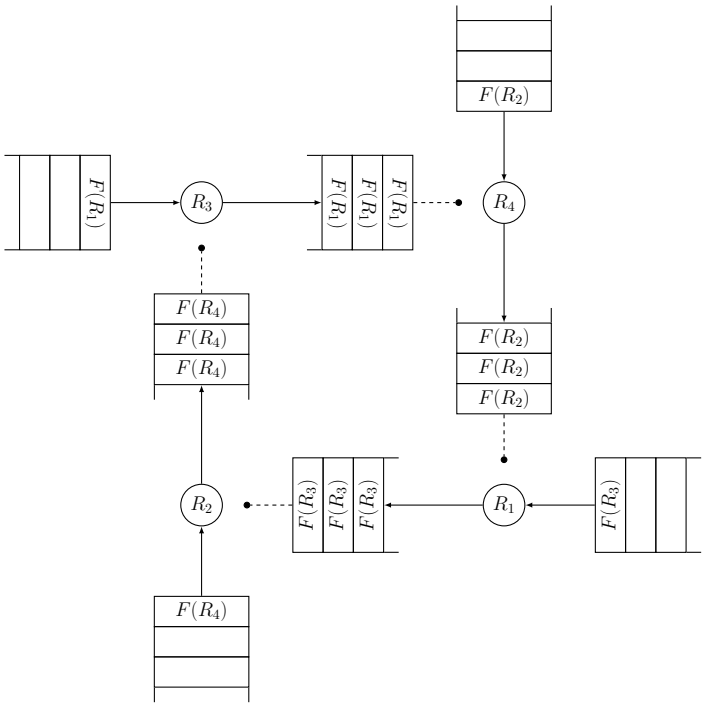


Figure 3.8: Example of deadlock on a wormhole-switched NoC.

themselves waiting for the R_3 to R_4 link that is occupied by the $F(R_1)$ flits. We can see clearly here the occurrence of an unsolvable cyclic dependency leading to a deadlock. In this case, the time needed by a packet to cross the network cannot be bounded.

Different approaches have been proposed in order to tackle the problem of deadlocks. In [52], Dally *et al.* provided a method to construct deadlock-free *non-adaptive* routing algorithms. They pointed out the correlation between deadlocks and the presence of cycles in the *Channel Dependency Graph* (or *CDG*) which represents the connections for all pairs of communication channels. They demonstrate that a routing algorithm is deadlock-free if and only if it produces CDG without cycles and they provide a method for breaking the cycles in a CDG using the notion of *Virtual Channel*.

In [50], Duato extended the work of Dally *et al.* and developed a theoretical framework for the design of deadlock-free *adaptive* routing algorithms for wormhole-switched networks. He demonstrated that the absence of cycles in the CDG is not required to construct adaptive deadlock-free routing algorithms and, based on that, he proved that, in this case, only the absence of cycles in the extended channel dependency graph of a routing sub-function is required. In [53], he extended his results to more general class of adaptive routing algorithms.

Many additional contributions have been proposed to tackle the problem of deadlock avoidance in wormhole switched networks [54, 55, 56]. In [51], Fleury & Fraigniaud proposed a unified definition of a routing function that captures most of the already existing definitions of the literature and extend their work to treat the problem of multicast routing algorithms.

In [57], Dupont de Dinechin *et al.* proposed a different solution to the deadlock avoidance problem on the KALRAY MPPA®-256. They use the hardware limiters of the NoC interfaces in order

to regulate the traffic injected on the network. By doing so, they are able to bound the maximum possible backlog for all the routers' buffers and they show that, assuming a proper configuration of the HW limiters, no buffer ever overflows, thus avoiding any potential deadlock.

3.3.1.2 Computing WCTT with Network Calculus

In a hard-real time context, the estimation of the WCTT of a NoC packet is commonly achieved using *Network Calculus* [58, 59] or *Real-time calculus* [60]. In either cases, the deadlock problem is usually assumed to be solved before the application of the technique.

The term of *Network Calculus* (or *NC*) refers to a collection of mathematical results used for the modeling and the analysis of deterministic queueing systems thanks to *Min-Plus* algebra. Using NC, the communication flows are modeled as cumulative functions counting the amount of data injected in the network over time. The traffic of all communication flows is assumed to be limited at source. The maximum envelope of traffic allowed for a communication flow is represented by the concept of an *arrival curve*.

Definition 5 (Arrival curve)**.** *An arrival curve $\alpha(t)$ is said to constrain a cumulative flow $x(t)$ if it verifies the relation 3.1.*

$$\forall s \leq t,\ x(t) - x(s) \leq \alpha(t-s) \tag{3.1}$$

Thanks to the Min-plus convolution denoted $\otimes$, the equation 3.1 can be rewritten as:

$$x(t) \leq (\alpha \otimes x)(t)$$

The major assumption of NC is that the injected traffic is constrained by arrival curves. In return, network nodes are assumed to provide some form of guaranteed services. The scheduling operated by a network node is modeled under the notion of *service curve*.

Definition 6 (Service curve)**.** *A network node $\mathcal{N}$ is said to offer a service curve β to an input flow $x(t)$ if its output flow verifies the relation 3.2.*

$$\forall t, \exists s \leq t,\ y(t) - x(s) \geq \beta(t-s) \tag{3.2}$$

Using Min-plus convolution, equation 3.2 can be rewritten as:

$$y(t) \geq (\beta \otimes x)(t)$$

The succession of network nodes crossed by a flow can be modeled by concatenating the nodes using the Min-plus convolution of their respective service curves.

Example 9 (Concatenation of network nodes)**.** *Let $\mathcal{N}_1$ and $\mathcal{N}_2$ be two network nodes respectively offering the service curves β_1 and β_2. Any flow crossing $\mathcal{N}_1$ and $\mathcal{N}_2$ consecutively is offered the service curve $\beta_1 \otimes \beta_2$*

Based on these models, NC helps in computing safe bound on the maximum *backlog* of each network node which represents the maximum amount of buffered data in a network switch. The backlog of a network node is defined as the difference between its input and output flow, i.e, $x(t) - y(t)$. Equation 3.3 is classical NC result providing a bound on the backlog.

$$x(y) - y(t) \leq \sup_{s \geq 0}\{\alpha(s) - \beta(s)\} \tag{3.3}$$

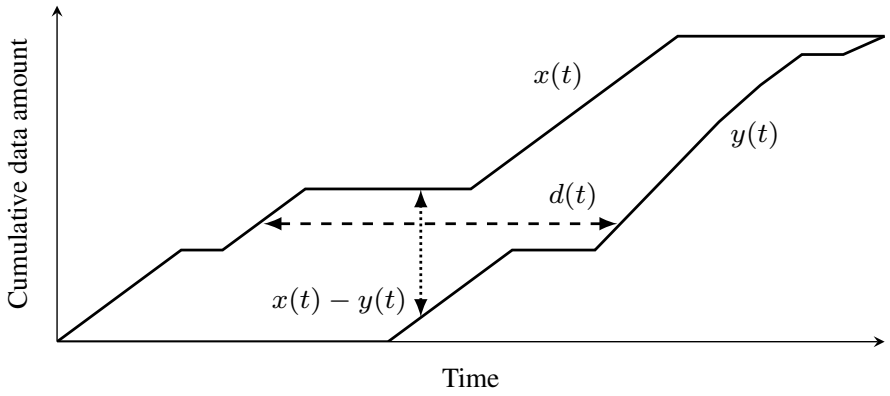


Figure 3.9: Example of two cumulative flows with graphical representation of the backlog and the delay

If the backlog never overpasses the size of the switch's buffers, then it is sound to assume that no buffer will overflow. Similarly, NC provides upper-bounds on the maximum *delay* suffered by a flow crossing the network. Let $\Delta(s) = inf\{\tau \geq 0 \ : \ \alpha(s) \leq \beta(s+\tau)\}$ be the virtual delay of a system having α as input and β as output, let $h(\alpha, \beta)$ be the supremum of all values of $\delta(s)$, the delay $d(t)$ is given by Equation 3.4.

$$d(t) \leq h(\alpha, \beta) \tag{3.4}$$

Figure 3.9 shows two cumulative function representing an input flow $x(t)$ and an output flow $y(t)$ with a graphical representation of the backlog (◂·····▸) and delay (◂ - -▸)functions.

Qian *et al.* [61, 45] and Zhan *et al.* [62] demonstrated the effectiveness of Network Calculus on NoCs, which was mostly used for the timing analysis of macroscopic-sized networks beforehand. Since then, NC has then been successfully applied to NoCs several times and notably on the KALRAY MPPA®-256 in [46] and [47].

Dupont de Dinechin *et al.* [46] used the work of Cruz [59] and Zhang [63] on (ρ, σ) calculus to derive backlog and delay bounds on the KALRAY MPPA®-256. They derived from the main (ρ, σ) relations a quadratic optimization problem whose solutions can be transcribed into configurations of the hardware limiters featured by the KALRAY MPPA®-256's NoC interfaces. Giannopoulou *et al.* proposed in [47] a similar method to compute (ρ, σ) parameters of the KALRAY MPPA®-256's hardware limiters based on backlog and delay constraints and they extended the approach to bound the number of memory accesses performed during the reception of NoC packets in a pre-definite time frame.

In this section, we focused on WCTT estimation for NoCs and, in particular, on the classical methods for avoiding deadlocks and the application of Network Calculus. In general, the WCET of tasks busy-waiting for packet reception must be computed with the addition of stalling timing penalties depending on the WCTT. This means that the WCET of such task is computed with timing penalties that are functions of the packets' WCTTs. Thus, the temporal validation of programs as required by certification depends not only on the WCETs of tasks but also on the WCTTs of packets sent through the NoC. Consequently, management of NoC sharing appears as a major issue for predictability.

In addition to the NoC, it appears that many-core architectures can also put high pressure on memory subsystems since the number of cores composing the processor can be high. Serving memory requests from many cores requires a high throughput. When such a resource is massively shared, being able to provide tight timing guarantees to applications can become challenging. The state-of-the-art methods that were proposed to upper-bound the duration of DDR-SDRAM transactions are overviewed in the next section.

3.3.2 Temporal bound for DRAM subsystems

Several works have already been carried-out to bound memory access durations of existing unmodified applications on COTS targets. Usually, they rely on detailed models of the memory access protocol provided together with a fine grained analysis of the COTS memory controller. For example, Wu *et al.* proposed in [64] a worst-case analysis for DDR-SDRAM in multi-core processors that benefits from an explicit DRAM state model. They assume a global FIFO queue of size n arbitrating the requests concurrently issued by the n cores. This provides a bounded service time for all requesters. With a similar approach, Ding *et al.* analyzed in [65] the DRAM access protocol with FIFO arbitration and close-page row-buffer management policy.

Unfortunately, modern DRAM controllers (such as the arbiter of the KALRAY MPPA®-256 detailed in Section 2.2.4.2) usually feature more complex out-of-order arbiters with open-page row-buffer management policy enabling better memory throughput in average [66]. The *First-Ready First-Come First-Serve* (or *FR-FCFS*) arbitration scheme, which is probably the most widely used in existing COTS processors, prioritize requests on already opened rows over requests on closed rows no matter how long they have been pending. So, theoretically, the service time of a request cannot be bounded because of potential starvation problems. Several contributions have been proposed to overcome this issue by modeling the whole the memory hierarchy, including the caches, instead of the DRAM controller's policy alone. Although not implementing a pure FR-FCFS policy, the KALRAY MPPA®-256's arbiter also prioritize requests on already opened rows over others in the Reordering pool. Since this behaviour is very comparable to a FR-FCFS arbiter, the analysis of other works focusing on open-page FR-FCFS controllers appears to be relevant in our case.

In [67], Kim *et al.* tackled the problem of bounding memory interference delays with a realistic FR-FCFS arbitration policy thanks to three assumptions on the system model: each task has been allocated a sufficient space in cache to store a complete DRAM row; all memory accesses are aligned with DRAM columns and tasks do not share memory. The authors propose to use two different approaches: 1) the *request-driven* approach which essentially consists in summing the maximum delays involved by each memory request issued from a task; and 2) the *job-driven* approach where the delays are derived from the worst interference scenario of co-running tasks. The authors argue that combining these two approaches enables to compute safe bounds on the worst case response time of each task in situation of interference.

In [68], Yun *et al.* extended the analysis of [67] to consider modern COTS processors where: 1) each core can issue several outstanding requests to mask memory latency; 2) the arbitration scheme uses FR-FCFS with higher priorities for *RD* over *WR* commands; and 3) *RD* and *WR* commands are issued in *batches* using a classical watermark mechanism [69]. They also assume specific hardware configurations such as partitioned caches and DRAM banks to limit interferences by design.

The problem of handling refreshes is commonly treated separately from the DRAM system analysis [39, 70, 67] by adding to the memory delay initially computed a refresh penalty E_R defined

as the fixed point of the recursive equation 3.5 with $E_R^0 = 0$.

$$E_R^{k+1} = \left\lceil \frac{(\text{total delay from analysis}) + E_R^k}{t_{REFI}} \right\rceil \times t_{RFC} \tag{3.5}$$

On the KALRAY MPPA®-256, the accesses to the DDR-SDRAM are not due to cache refills of evictions. To the best of our knowledge, no already published work consider many-core processors where memory transactions are achieved using DMA transfers that are likely to be large and explicitly initiated by software. Overall, bounding the WCET of a task accessing the main memory requires to bound the duration of memory transactions. In a time-compositional approach, the bound on the memory access duration when interfering with other memory masters can be used as a timing penalty that is summed to the WCET of the task without interferences. Despite accurate models, the memory latency bounds provided by purely analytical approaches are usually driven by a pathological pattern of requests for which the average-case optimization mechanisms of the controller performs poorly. As explained in section 3.4, if the bound provided by these approaches are not sufficient, the issue is usually overcome using either bespoke DRAM controllers which achieve highly timing-predictable accesses to memory or appropriate software to avoid pathological situations.

3.3.3 End-to-end timing bounds on multiple resources

Many-core processors have a hierarchical architecture. For this reason, the interferences on shared resources often appear simultaneously at different levels. For example, a transaction to the main memory initiated by a core usually imply potential interferences at both the NoC and the DDR-SDRAM levels. Hence, the timing behaviour of the two resources are inter-dependent and must not be considered separately. The end-to-end timing analysis of multi-resource interferences is thus a major problem to bound execution time many-core processors. Yet, this problem was mostly unexplored in past contributions. In [71], we provided models of all shared resources on a KALRAY MPPA®-256 processor and we proposed an approach to compose the output of these models in an end-to-end timing analysis of DDR-SDRAM transaction. We will detail this contribution in Chapter 5.

3.4 Enforcing time-predictability by design

We overviewed in Section 3.3 the methods for computing interference penalties when accessing the NoC and the DRAM subsystem on a many-core processor. Since most of existing COTS have been designed to optimize average performance rather than worst-case performance, considering chaotic accesses to the NoC and/or the DRAM can lead to prohibitive interference penalties. In order to overcome this problem, a number of contributions have been made to design systems with predictability and worst-case behaviours in mind. We will firstly overview in sub-section 3.4.1 the proposition that were made to use bespoke time-predictable hardware components. Secondly, we will analyze in sub-section 3.4.2 the approaches relying on software-based solutions to enhance the predictability of COTS hardware.

3.4.1 Custom-made time-predictable hardware

Since this thesis aims at directly fitting into an industrial context, only COTS components will be considered in the rest of the dissertation. However, one may note that the precise analysis of

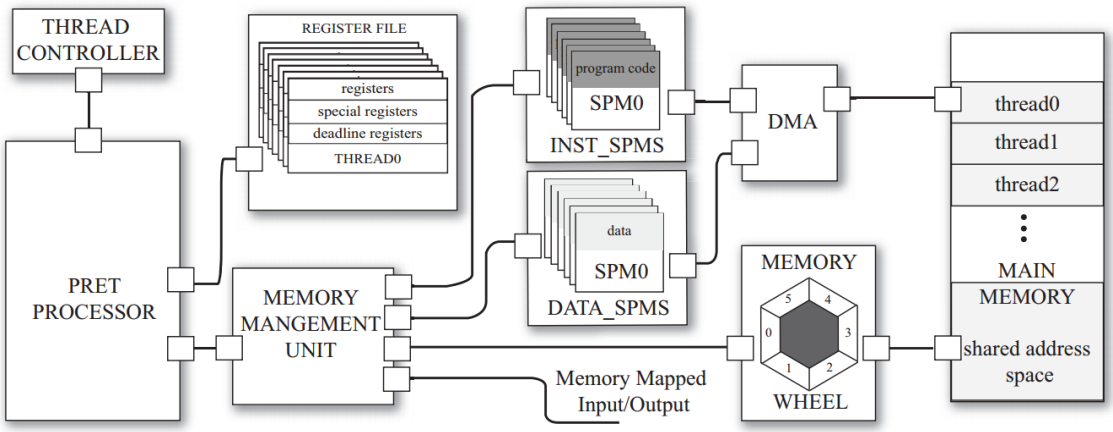


Figure 3.10: Block diagram of the PRET architecture (extracted from [76]).

the existing bespoke time-predictable architectures and the key concepts of their design such as the budget-based approach of CompSOC [72], the TDM scheduling of Argo [73] or the bank-privatization proposed in ROC [74] have been major sources of inspiration when developing our execution model for the KALRAY MPPA®-256 and its associated hypervisor. In the next section, we will overview the methods that are commonly used at architecture level to design predictable processors. And then, we will analyze how NoCs and DDR-SDRAM subsystems can be made predictable by design.

3.4.1.1 Processor architectures

Following the case of Edwards and Lee [75], Lickly *et al.* presented in [76] a processor designed following the *PRET* philosophy. As shown in Figure 3.10, it implements a SPARC V8 ISA [77], a six-stages thread-interleaved pipeline without bypasses in the processing core and separated instruction and data scratchpads (instead of classical caches) that are connected to a DMA unit in charge of moving data between the scratchpads and the main memory. All the components of the PRET architecture including the thread-interleaved pipeline [78, 79], the processing cores [80] and the scratchpad memories [81] have been chosen for their efficiency and their good timing predictability, thus enabling easier WCET estimations. In [82] Zimmer *et al.* extended this work and introduced the FlexPRET processor in order address the problems related to mixed-criticality systems by leveraging hardware-based isolation. They propose an architecture based on timing-extended version of the RISC-V ISA [83] with a 5-stages classical RISC pipeline enabling arbitrary interleaving of threads. The threads are classified in either the *hard real-time threads* (or *HRTTs*) category or in the *soft real-time threads* (or *SRTTs*) category. Both the HRTTs and SRTTs can have guarantees to be scheduled at specific clock cycles. Empty clock cycles are filled by SRTTs arbitrated in a Round-Robin fashion.

Many other time-predictable architectures have been proposed in the literature. The T-CREST platform [84] aims at providing a WCET analyzable multi-processor system-on-chip (or *SoC*) architecture. As shown on Figure 3.11, a T-CREST chip implements several Patmos [85] cores that are connected to both a memory tree to access the external DDR-SDRAM and to a NoC for direct core-to-core communication. In order to simplify the WCET analysis, the Patmos cores feature Very Long Instruction Word (VLIW) pipelines; dedicated scratchpads memories and several spe-

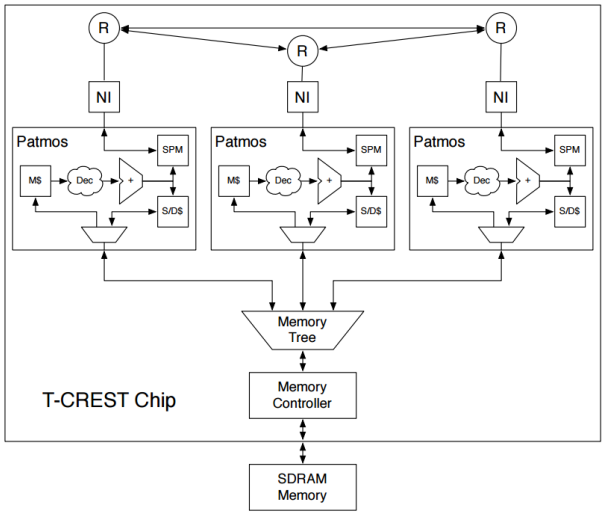


Figure 3.11: Block diagram of the TCREST architecture (extracted from [84]).

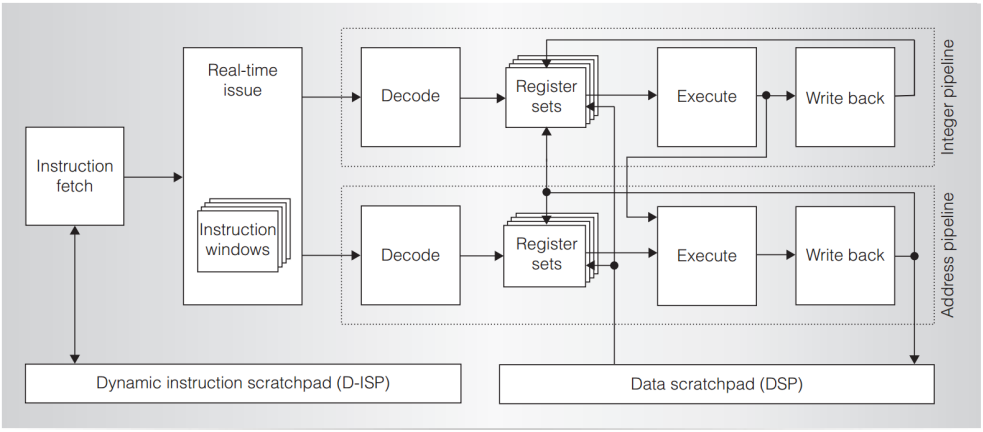


Figure 3.12: Block diagram of the Merasa architecture (extracted from [87]).

cialized caches [86] such as the method cache for storing complete functions or a split data cache to store the stack, heap and constants and static data in different caches. Overall, the VLIW cores and the separated caches enable an efficient timing analysis and WCET estimation.

In the context of the Merasa [87] project (and more recently parMerasa [88]), a WCET-analyzable multi-core architecture has been proposed. It comprises up to 16 cores implementing the TriCore instruction set [89] with separated pipelines, register files, instruction windows and caches for real-time and non-real-time threads. The inter-core interferences are managed using a complex 2-level bus arbiter which firstly arbitrate in each core the requests between the co-running threads caches and scratchpads and secondly the requests to access the main memory that passed the first arbitration level [90]. This helps in providing temporal isolation between co-running threads, hence simplifying isolated timing analysis.

On the other hand, the CompSOC [72] approach relies on the notion of *virtual execution platforms* to allow an independent design, verification and execution of several co-hosted applications. This independence is enabled by a hardware and software template [91] enforcing strong temporal

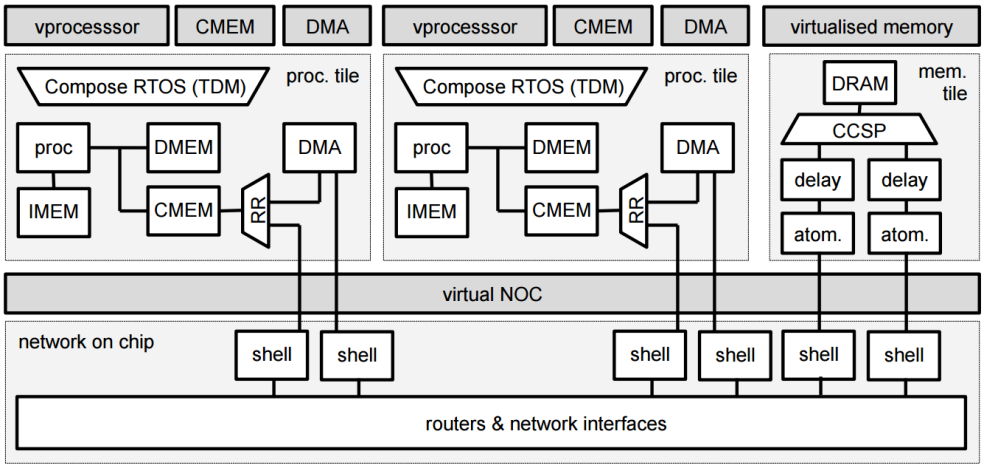


Figure 3.13: Block diagram of a CompSOC instance (extracted from [72]).

isolation between virtual execution platforms. Temporal isolation enables composable execution of different applications hosted on different virtual platforms thus enabling de-coupled WCET estimations. Figure 3.13 shows an example of CompSOC instance.

3.4.1.2 On-chip networks

The design of time-predictable interconnects and especially time-predictable on-chip networks has been an active research topic over the past few years. In the context of the T-CREST project, Kapapaki *et al.* proposed in [73] the Argo NoC. Argo uses a global and static Time Division Multiplexing (or *TDM*) scheduling scheme to provide time-guarantees to applications. It is composed of simple routers (no buffering is required since no collisions can happen thanks of the TDM schedule) that are connected to the cores memories with Network Interfaces which all integrate a DMA unit in charge of sending the data over the NoC.

In a similar fashion, the Aethereal [92] NoC that is used in CompSOC also uses global TDM scheduling to avoid the concurrency of packets travelling at any time.

3.4.1.3 DDR-SDRAM controllers

The problem of achieving time-predictable DDR-SDRAM access is a major concern for the design of time-critical embedded systems [20].

In the context of CompSOC, Akesson *et al.* proposed the *Predator* [93] memory controller. Predator uses a close-page row-buffer-management policy [94] together with a Credit-Controlled Static-Priority Arbiter [95]. All memory requesters are assigned (ρ, σ) constraints in order to provide timing guarantees to the initiating applications with network calculus [59]. In [96], Paolieri *et al.* proposed an *Analyzable Memory Controller* (or *AMC*) that also uses a close-page row-buffer-management policy. AMC implements one request queue per hard-real-time tasks. All the queues are arbitrated in a Round-Robin fashion which, unlike in Predator, does not require pre-allocation of any sort of budget but rather eases the analysis since no starvation can ever occur.

Reineke *et al.* proposed in [97] the *PRET DRAM controller* that has been designed to take into account the DRAM device not as a single resource but rather as a set of resources (the banks and ranks) to be shared between one or more requester individually. To do so, they partition the address space based on the internal DRAM structure in order to interleave the accesses targeting the same physical resource. The *Rank-switching Open-Row DRAM Controller* (or *ROC*) of Krishnapillai *et al.* [74] leverages bank privatization to limit the impact of row conflicts in order to enable a beneficial open-row row-buffer-management policy. Moreover, ROC uses rank-switching techniques in order to mask write-to-read latencies. The privatization mechanism offers isolated performances enabling decoupled estimation of memory transaction durations. In addition, the rank-switching techniques improve the overall memory throughput.

In [98], Ecco and Ernst extended previous state-of-the-art real-time DDR-SDRAM controllers by introducing read/write bundling techniques. They highlight that existing controllers do not reorder DRAM commands in function of their direction (read or write). Because of this, timing bounds need to be computed while assuming alternating read and write patterns since it exhibits the highest latency because of bus reversals. The authors introduce a complex arbitration scheme which bundles RD and WR commands at the channel level. The arbiter operates in rounds to select the pending requests that will be issued by the controller. At each round, the priority is given to a request that does not involve bus turnarounds, thus maximizing consecutive reads or consecutive writes. The controller remains predictable in the sense that safe timing bounds can be derived from the timing analysis of the authors while read and write bundling improves the overall memory throughput.

3.4.2 Software-enforced predictability on COTS

Software-based solutions to enhance the predictability of COTS differ in their nature. Essentially two classes of solutions can be identified: solutions with a system-level control software and solutions where applications are modified. In the former, applications are left untouched and only the system software is modified to take interferences into account. This essentially makes possible to run unmodified legacy applications on new hardware platforms in a predictable manner. In the latter, the applications themselves are modified to avoid the problematic situations. The benefits expected from these approaches is to offer not only a good time-predictability but also better usage of resources. Both categories are detailed below.

3.4.2.1 Interference-aware control software

Fisher proposed in [99] the world's first commercial and certifiable (under the Safety Integrity Level 4 of the railway industry regulations [100]) run-time software for time-critical systems on a multi-core processor. In this approach named *Deterministic Adaptative Scheduling*, the tasks in the system must be classified by the system designer under the *critical* or *non-critical* category. Then, the time is divided in critical and non-critical slots. During non-critical time slots, all the non critical tasks are allowed to run concurrently on all the cores. During the critical slots, only the core running the critical task is allowed to run and all the other cores are stalled until the end of the slot. By doing so, the critical tasks never suffer from potential interferences on shared resources since they always execute in isolation. An example of the Deterministic Adaptative Scheduling technique is depicted in figure 3.14. This approach provides good timing predictability to critical applications but can lead to excessive under-utilization of the cores, especially with many critical tasks.

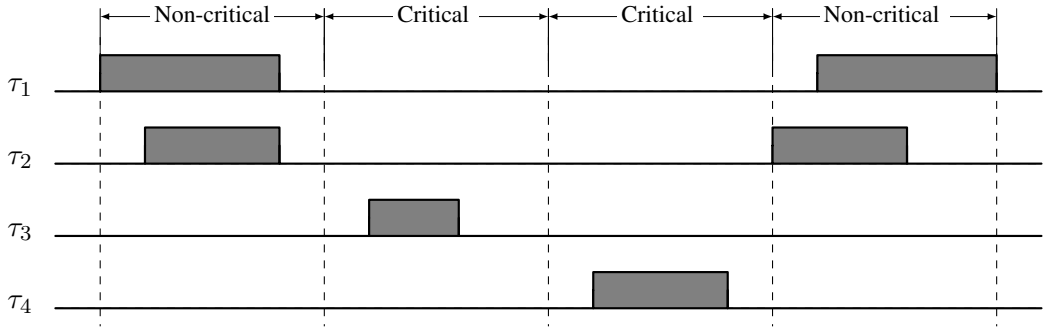


Figure 3.14: Example of Deterministic Adaptative Scheduling with τ_1 and τ_2 non critical tasks and τ_3 and τ_4 critical tasks.

In [101], Jean *et al.* proposed MARTHY (figure 3.15), a hypervisor providing the property of robust partitioning by mastering the accesses initiated by applications to shared resources. In the idea, an instance of MARTHY is duplicated and locked in the private cache of each core in the processor. Then, an appropriate core and MMU configuration is used in order to hide the locked ways of the cache to the application and to detect all the events considered as *sensible*, meaning all the events that will have an impact on the traffic going through the interconnect or directly on the state of other cores. As shown in Figure 3.16, when one of these events is detected while an application is running, it is signaled by an exception in order to give back the control to the local instance of the hypervisor. MARTHY will then process the action demanded by the application, but only during a pre-computed time slot in which it can access the shared resources in complete isolation. All the instances of MARTHY access the shared resources without competitors thanks to a global TDM schedule. Basically, this architecture enables several time-critical applications to run simultaneously as long as they do not access shared resources. By doing so, the limitations of the Deterministic Adaptative Scheduling have been (at least partially) overcome. Moreover, the avoidance of competition for accessing shared resource ensures a *time-composable* execution of applications, thus easing the WCET estimation. However, some applications can still suffer large slowdowns [102] in specific cases. One of the reasons explaining those performances is the non-synchronization between the applications and the global TDM tick. Such an approach makes trivial the execution of legacy applications without modifications but also suffers from potentially high latencies to access the resources when misaligned with the TDM schedule.

In [103], Yun *et al.* introduced Memguard (figure 3.17), a memory regulation system aiming at providing guaranteed bandwidth to applications on a multi-core target. Basically, each core is assigned a budget in the form of a maximum number of bus accesses within a specific time slot. These bus accesses are detected using Last Level Cache (or *LLC*) miss performance counters which trigger an interruption when the budget is totally spent. This is done in order to prevent any further accesses from the considered core before the end of the time slot. In Memguard, the sum of the allocated budgets is assumed to be less than the worst-case bandwidth of the DRAM subsystem to ensure that the guaranteed bandwidths can always be reached for all requesters. This performance guarantee theoretically simplifies the WCET computation by enabling trivial estimation of the duration of memory transactions. Indeed, guaranteed bandwidth helps to bound memory access time, thus providing a timing penalty that can be summed to a WCET computed in isolation. However, a global reclaim manager retrieves the unspent bandwidth (both when a core does not use all its

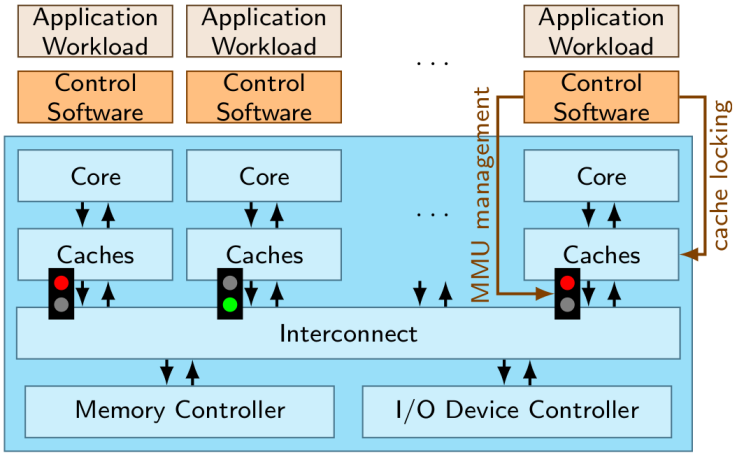


Figure 3.15: Architecture of MARTHY (extracted from [101])

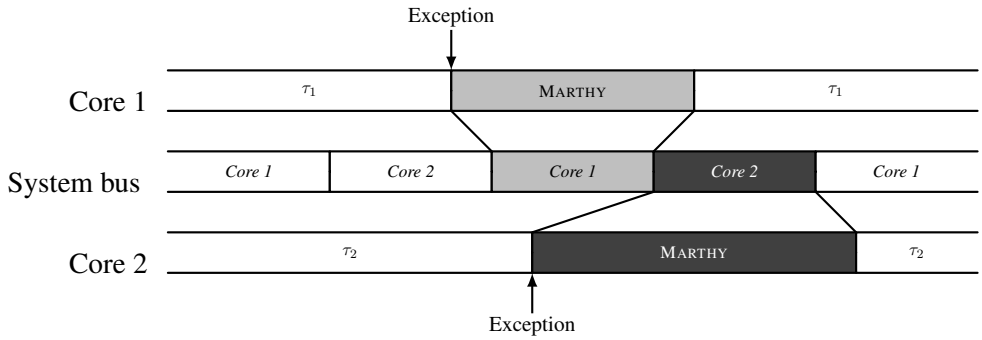


Figure 3.16: Example of execution with MARTHY with two tasks τ_1 and τ_2 running on two cores

budget or when the actual performance was better than the worst-case) from all cores in order to redistribute it for best-effort applications. The decision of one core regulator to donate a part of its budget to the global reclaim manager is nonetheless based on a speculative estimation of the future memory demands of the core. By doing so, the regulator takes the risk of an optimistic speculation that may ruin the bandwidth guarantees.

In [104], Kostrzewa *et al.* proposed a software-based admission control mechanism controlling the rate at which applications can access the NoC on a many-core processor. Their solution is based on an *access layer* distributed over the *Resource Managers* (or *RMs*) of the NoC nodes. Each RM dynamically regulates the injection rates of its local NoC node based on chip-wide information on the current NoC load. A synchronization protocol is provided to exchange information between RMs. This is done using either a specific Virtual Channel or a parallel control NoC as in the KALRAY MPPA®-256 [17]. By doing so, the access layer provides quality-of-service mechanisms to real-time applications by adjusting injection rates to ensure the respect of timing constraints. Moreover, it improves the overall performance thanks to the dynamic adaptation.

3.4.2.2 Modification of applications

Pellizzoni *et al.* proposed in [105] a *Predictable Execution Model* (or *PREM*) to mitigate the unpredictability of COTS processors. The key idea is to avoid the contention on shared resources

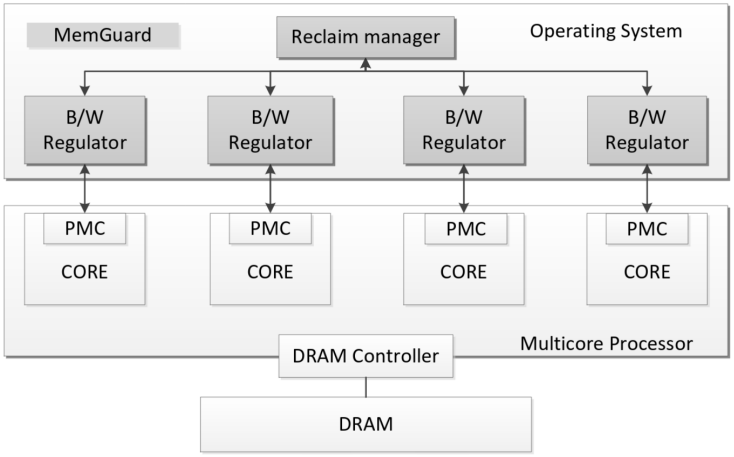


Figure 3.17: Architecture of Memguard (extracted from [103])

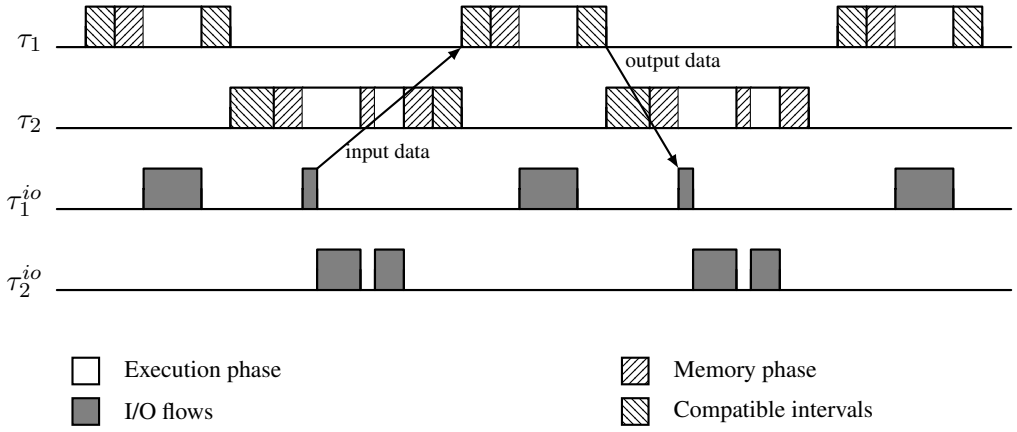


Figure 3.18: Example of schedule following PREM with two tasks τ_1 and τ_2

in order to become insensitive to the behaviors of low level non real-time arbiters. More precisely, a specific intrusive hardware bridge [106] is assumed to regulate the accesses of I/O peripherals to the DRAM subsystem. Then, all the source code of all the tasks must be annotated by the application designer in order to be modeled as a set of non-preemptable *intervals* of two types (each task can be associated with several intervals of any type). The *predictable* intervals are divided by the compiler into *memory phases* during which the core fetches data or code from the DDR-SDRAM into its cache memories and *execution phases* during which the core computes the actual code of the tasks without any cache miss. The *compatible* intervals are provided to support OS calls and are compiled and executed without modifications. The only assumption on compatible interval is that the time required for the system calls to be served can be bounded. All the intervals and their sub-phases are then scheduled together with the regulated I/O flows in order to avoid all the competition when accessing shared resources, thus ensuring predictable and isolated executions of the tasks as depicted in figure 3.18.

The key concepts of PREM have been extended by Durrieu *et al.* in [107] to port an avionic application from Thales on a multi-core COTS target [108]. They assume an *Acquisition, Execution, Restitution* task model (or *AER*) where execution and communications are decoupled. During

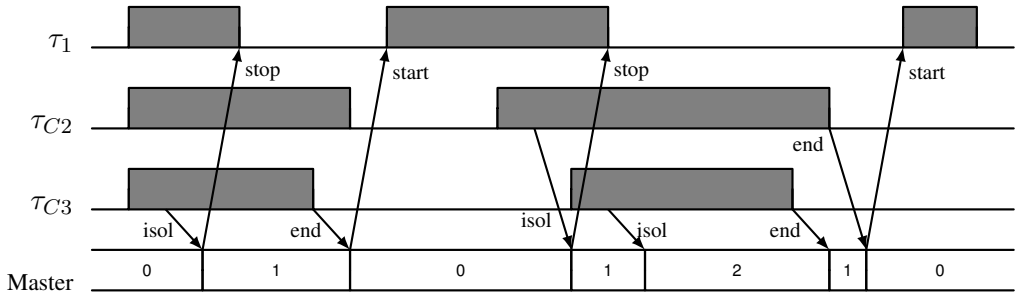


Figure 3.19: Example of distributed run-time WCET control with one non-critical task τ_1 and 2 critical tasks τ_{C2} and τ_{C3}

acquisition phases, the code and data required for the following execution phase are loaded into the private L2 cache of a core that has previously been configured in SRAM. Then, the actual code of the task can be computed without accessing any shared resources, thus enabling parallel executions of independent tasks. Finally, the data produced by the task is committed to the memory during the restitution phase. In order to improve the predictability of accesses to shared resources, only one acquisition or restitution phase is allowed to run at a time. One may note that the applicability of such an execution model in a multi-core environment relies on the ability to compute a global schedule of the accesses to shared resource. To tackle this problem, Durrieu *et al.* propose a target-dependent synchronization algorithm to be run at startup to reach a global notion of time shared by all cores. Overall, this approach offers interference-free execution phases and competition to shared resources are avoided by construction. Doing so greatly simplifies the computation of the WCET of execution phases and timing bounds for the access to shared resources.

In [109], Jegu *et al.* patented a method to enforce the predictability of a COTS multi-core processor. Their approach is close to the philosophy of PREM or AER since the decoupling of communication and execution is used to avoid concurrent accesses to the shared bus thanks to a global schedule and a specific control software.

In [110], Kritikakou *et al.* proposed a distributed runtime to control the WCET of mixed-critical tasks aiming at maximizing the resource utilization on a multicore processor. In this approach, the tasks are categorized as critical or non-critical and the WCET_{iso} of a critical task is defined as its WCET when executing in parallel with all the other critical tasks (but no non-critical). From there, the CFG of each critical task is assumed to be *instrumented*. That means that several parts of the CFG are enhanced with a specific control software capable of detecting if, at some point of the CFG, there is a risk for the running task to exceed its WCET_{iso}. Based on that, all the tasks (both critical and non-critical) run in parallel together with a *master* control software. Then, if a critical task detects a potentially too large execution time, it sends an *isol* request to the master. In response, the master stops all the co-running non-critical tasks to mitigate the interferences suffered by the critical tasks and waits for the completion of the critical task before re-starting the non-critical one. An example of this behaviour is depicted in figure 3.19.

In [111], Tabish *et al.* proposed a scratchpad-centric approach for executing real-time tasks on the NXP MPC5777M multi-core target [112]. They assume AER-like tasks split in load, execute and unload phases. The key idea of their contribution lies in the extensive exploitation of the scratchpad memories (or SPM) that are associated to each core. During load phases, a DMA unit

fetches the code and data of one task from the main memory to one half of the SPM. Then the core enters in an execution phase during which it only accesses its local SPM. During the execution, the DMA can fetch the code and data of the next task to run in the second half of the SPM. When reaching unload phase, the DMA writes the output of the completed task from the SPM to main memory, thus leaving half on the SPM ready for loading ta new task. In a multi-core configuration, the memory bus is scheduled in TDMA and DMA transfers can occur in pre-assigned slots. As for PREM and AER, this approach enables executions without interferences. Moreover contentions on I/O are avoided. Overall, it eases the WCET estimations and the computation of bounds on the shared resources access times.

3.5 Summary

In this chapter, we firstly overviewed the two main WCET estimation methods; namely static analysis and measurement-based techniques. Although the latter have shown promising progress recently, especially regarding probabilistic approaches, they still fail to provide bounds that are as sound as those provided with static analysis; thus limiting their usability in safety-critical avionics systems. We identified the computation of interference penalties on wormhole-switched NoCs and COTS DDR-SDRAM subsystems to be major problems to ensure time-predictability on many-core processors. We studied solutions to mitigate interferences on such shared resources through either custom-made hardware platforms or bespoke control software on COTS processors and we identified the latter to be better suited to an industrial context at the time of writing. Yet, the existing control software that have been proposed in the literature target only mono and multi-core processors, leaving the problem of mastering many-cores mostly unexplored. Solving this issue will be the first axis of contribution of this thesis.

Chapter 4

Off-line scheduling of parallel tasks

Contents

4.1 Off-line scheduling of DAG tasks 71
4.1.1 DAG model 71
4.1.2 DAG notions 72
4.1.3 Off-line model transformation 73
4.1.4 Off-line mapping of DAGs on multi-core 75
4.2 Off-line mapping on distributed architectures 78
4.2.1 Mapping on the KALRAY MPPA®-256 78
4.2.2 Mapping on other many-core architectures 80
4.2.3 Core and network co-scheduling 82
4.3 Summary 82

In this chapter, we investigate the methods to schedule parallel tasks off-line and we give a particular focus on approaches that can be applied to distributed architectures while meeting constraints 3 and 4 detailed in Chapter 2. We firstly overview in Section 4.1 the existing works that deal with *DAG tasks* [113] since they have many similarities with our application model. We then focus on methods that have been proposed to schedule DAG tasks off-line on multi- and many-core processors. We will analyze methods based on off-line transformations of the model and approaches using complete off-line scheduling of the tasks. Finally, we will detail the solutions that were proposed to tackle hard real-time mapping problems on distributed architectures.

4.1 Off-line scheduling of DAG tasks

The *Directed Acyclic Graph* (or *DAG*) model is often considered as the most general model for tasks with explicit parallelism [113]. Indeed most of other parallel task models such the Fork-join [114] and the Parallel Synchronous [115] models can be seen as special cases of the DAG model.

4.1.1 DAG model

A periodic DAG task is modeled as a set of sub-tasks whose execution order is constrained by precedence relations. Studying the DAG model is relevant in our case since its suits very well one part of our problem, namely the precedence constraints between sub-tasks. However, one may note that our model is more expressive than a pure DAG because communications between sub-tasks

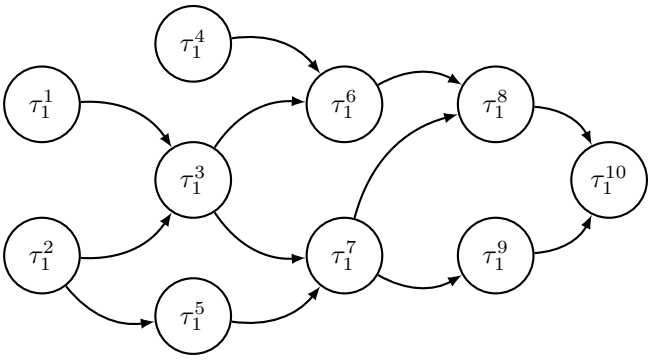


Figure 4.1: Example of a DAG task τ_1 with 10 sub-tasks

are made explicit as well as the transactions targeting external memories. A real-time DAG task is periodically activated and must complete before an implicit deadline. Usually, all sub-tasks are activated with their parent task and temporally constrained with the same deadline. More precisely, a taskset is modeled as $\tau = \{\tau_1, \ldots, \tau_n\}$ with $\tau_i =< S_i, P_i, T_i >$ a DAG task characterized by: $S_i = \{\tau_i^1, \ldots, \tau_i^{n_i}\}$ the set of sub-tasks, $P_i \subset S_i \times S_i$ the set of precedence relations between the sub-tasks and T_i the period of the task. Each sub-task τ_i^j is associated with its WCET C_i^j. An example of DAG task is depicted in Figure 4.1. A DAG task can be seen as a directed graph [116] $G =< N, E >$ where the *nodes* $N = S = \cup S_i$ and the precedence relations are the *edges* $E \subset N \times N$.

We will firstly introduce classical DAG notions to ease the understanding of the following sections. Then, we will study the founding contributions on DAG scheduling on mono-core processors before analyzing the extensions that were made more recently for multi-core chips.

4.1.2 DAG notions

DAGs inherit from all the theoretical foundations of Graph Theory [116]. Manipulation of DAGs are commonly achieved using classical mathematical notions on directed graphs. For commodity, we remind some of these notions and precise the notations below.

A *path* is a sequence of adjacent nodes defined as $p =< n_1, \ldots, n_k >$. p connects n_1 to n_k if and only if subsequent nodes are connected by edges in E, i.e,

$$\forall i \in [1, k-1], (n_i, n_{i+1}) \in E$$

The *parents* of a node $n_i \in N$ are the set of nodes connected to n_i by an edge and preceding it in the ordering. More precisely, n_j is a parent of n_i if and only if $(n_j, n_i) \in E$. Similarly, the *children* of n_i are the set of nodes connected to n_i by an edge and following it in the ordering. More precisely, n_j is a child of n_i if and only if $(n_i, n_j) \in E$. For commodity, let us define the following notations:

$$Parents(n_i) := \{\ n_j \mid \forall (n_j, n_i) \in E\ \}$$

$$Children(n_i) := \{\ n_j \mid \forall (n_i, n_j) \in E\ \}$$

Example 10 (Parents and children in DAGs)**.** *The parents of the sub-task τ_1^7 in Figure 4.1 are τ_1^3 and τ_1^5. Its children are τ_1^8 and τ_1^9.*

The *predecessors* of a node extend the notion of parents recursively. n_j is a predecessor of n_i if and only if there is a path linking n_j to n_i. Similarly, the *successors* of a node extend the notion of children recursively. n_j is a successor of n_i if and only if there is a path linking n_i to n_j. Let us define the following notations:

$$Preds(n_i) := Parents(n_i) \bigcup_{n_j \in Parents(n_i)} Preds(n_j)$$

$$Succs(n_i) := Children(n_i) \bigcup_{n_j \in Children(n_i)} Succs(n_j)$$

Example 11 (Predecessors and successors in DAGs)**.** *The predecessors of the sub-task τ_1^7 in Figure 4.1 are $\tau_1^1, \tau_1^2, \tau_1^3$ and τ_1^5. Its successors are τ_1^8, τ_1^9 and τ_1^{10}.*

The notions of *entry* and *exit* sub-tasks in a DAG refer respectively to sub-tasks having no predecessors and sub-tasks having no successors.

The *Critical Path* (or CP) in a DAG is its longest *sequential* path. The length of a path in a DAG task model is equal to the sum of the WCETs of sub-tasks composing it. The CP of a DAG task represents a lower-bound on the execution time of the task when fully parallelized. The ratio between the WCET of the task and the length of its CP is defined as the *maximum speedup* M [117].

$$M(\tau_i) = \frac{\sum_{\tau_i^j \in S_i} C_i^j}{\sum_{\tau_i^k \in CP(\tau_i)} C_i^k}$$

Example 12 (CP and speedup)**.** *Assuming that odd sub-tasks of Figure 4.1 ($\tau_1^1, \tau_1^3, \tau_1^5, \tau_1^7$ and τ_1^9) have a WCET of 5 and the even sub-tasks ($\tau_1^2, \tau_1^4, \tau_1^6, \tau_1^8$ and τ_1^{10}) have a WCET of 3, the critical path of the task is $CP(\tau_1) =< \tau_1^1, \tau_1^3, \tau_1^7, \tau_1^9, \tau_1^{10} >$ with a length of 23. The total WCET of τ_1 is 40. Thus, $M(\tau_1) = 40/23 \simeq 1.74$.*

DAG nodes can be sorted in a *topological order* to produce a linear sequence of sub-tasks that respect the precedence constraint.

Example 13 (Topological order)**.** *A topological order of the DAG of Figure 4.1 is:*

$$< \tau_1^1, \tau_1^2, \tau_1^4, \tau_1^5, \tau_1^3, \tau_1^7, \tau_1^6, \tau_1^8, \tau_1^9, \tau_1^{10} >$$

Topological sorting is of importance since its provides a correct mono-core schedule of a DAG task. Algorithms are known to produce topological ordering of any DAG in linear time.

4.1.3 Off-line model transformation

Chetto *et al.* presented in [118] an approach to schedule independent periodic sequential tasks together with sporadic groups of DAG tasks on *mono-core* processors. Chetto *et al.* proposed to modify the timing parameters of the constrained sub-tasks to transform them in an equivalent set of independent tasks that can be scheduled with classical algorithms such as EDF. The idea is to assign each sub-task in the DAG a modified release date based on the finish time of its parents and a modified deadline based on the deadline of its children. In [118], only one DAG task in

considered. Each sub-task τ_i^j is also provided together with an activation date (or *offset*) s_i^j and a relative deadline d_i^j. The idea is to replace the precedence relations P_i by changing the s_i^j and d_i^j parameters of each sub-task. We denote the modified parameters $\widetilde{s_i^j}$ and $\widetilde{d_i^j}$. Deadlines are adjusted as follows:

1. For all tasks with no successors : $\widetilde{d_i^j} := d_i^j$;
2. Select a task for which all children are already assigned an adjusted deadline;
3. For the selected sub-task τ_i^j:

$$\widetilde{d_i^j} := \min(\ d_i^j\ ,\ \min_{\forall \tau_i^k \in children(\tau_i^j)} (\widetilde{d_i^k} - C_i^k)\ )$$

4. Repeat step 2 and 3 while there are tasks with unadjusted deadlines.

In a symmetric manner, the activation dates are adjusted as follows:

1. For all tasks with no predecessors : $\widetilde{s_i^j} := s_i^j$;
2. Select a task for which all parents have adjusted activation dates;
3. For the selected task τ_i^j:

$$\widetilde{s_i^j} := \max(\ s_i^j\ ,\ \max_{\forall \tau_i^l \in parent(\tau_i^j)} \widetilde{s_i^k} + C_i^k)\ )$$

4. Repeat step 2 and 3 while there are tasks with unadjusted activation dates.

The process of adjusting timing parameter is achieved prior to execution of the tasks. Once all tasks have updated timing parameters, they can be considered as independent. Thus, they can be scheduled on-line using classical scheduling algorithm taking into account offsets and constrained deadlines.

Example 14 (Modification of timing parameters)**.** *Let us consider 10 sub-tasks* $\{\tau_1^1, \ldots, \tau_1^{10}\}$ *constrained by the precedence relations depicted in Figure 4.1. Table 4.1 shows an example of modified timing parameters for the 10 sub-tasks following the algorithm described in [118].*

The authors proved the optimality of this algorithm in the sense that the original dependent task set is schedulable if and only if the modified taskset is schedulable using a preemptive scheduling algorithm. However, scheduling tasks with offsets in a non-preemptive manner, even without precedence constraints remains NP-hard [119]. It is also Co-NP hard in the strong sense for preemptive tasks on one core [120]. And similarly, Ullman showed in [121] that, in a *multi-core* context, scheduling DAG tasks is also strongly NP-Hard, even using a preemptive algorithms.

In a similar approach, Qamieh *et al.* [122, 123] adapted the method of [118] to multi-core processors using the *DAG stretching algorithm*. The idea of is to append as many non-critical sub-tasks as possible to the critical path to form one sequential thread with a maximum utilization. By doing so, the number of sub-tasks that are executed in parallel with this main thread is minimized. The precedence relations of sub-tasks executed in parallel to the main thread are enforced using adjusted timing parameters as in [118]. The modified taskset is then scheduled online using GEDF. In [123],

Sub-task	s_1^i	C_1^i	d_1^i	T_1^i	$\widetilde{s_1^i}$	$\widetilde{d_1^i}$
τ_1^1	0	3	40	40	0	25
τ_1^2	0	4	40	40	0	25
τ_1^3	0	2	40	40	4	27
τ_1^4	0	6	40	40	0	28
τ_1^5	0	1	40	40	4	40
τ_1^6	0	3	40	40	6	31
τ_1^7	0	4	40	40	6	31
τ_1^8	0	6	40	40	10	37
τ_1^9	0	5	40	40	10	37
τ_1^{10}	0	3	40	40	16	40

Table 4.1: Example of adjusted timing parameters for DAG tasks using the algorithm described in [118]

Qamieh *et al.* showed in simulation with synthetic tasksets that, despite the goal of minimizing the parallelism during execution, this approach had better schedulability ratios than existing techniques for scheduling DAG tasks. Yet, it can be argued that minimizing the slack of the master thread increases the risk of suffering deadline misses if the WCET estimation of a sub-task was inaccurate.

Many of the other contributions on the multi-core scheduling of DAG tasks, including [113], [124] and [125] are based on pure on-line scheduling techniques and thus out of the scope of this thesis.

4.1.4 Off-line mapping of DAGs on multi-core

4.1.4.1 Real-time scheduling

The problem of mapping off-line dependent tasksets on multi-core processors has already been addressed in the real-time community. Grolleau and Choquet-Geniet [126] proposed a technique based on Petri nets to compute such mappings. They fully model the tasks and the execution target with Petri nets augmented by a constraint on the largest number of transitions crossed simultaneously as shown in Figure 4.2. This approach allows both preemption and migration of tasks between cores of the processor. The schedule is computed with the Petri nets and the mapping is done afterwards in post-processing in order to minimize the number of migrations. Overall, this approach provides a formal solution to the scheduling problem and can provide optimal results regarding different performance criteria (shortest response time, balance of idle periods, ...). However, the scalability of Petri nets on that kind of problem is questionable and the approach needs to be evaluated over large applications.

Unfortunately, several other formal approaches to this problem such as the work on Priced Timed Automata of Behrmann *et al.* [127] or the UPPAAL formulation of Baro *et al.* [128] also face major scalability issues.

In [129], Boniol *et al.* tackled the scalability issue by using an approach based on *Constraint Programming*. They rely on an AER-like execution model (Section 3.4.2) where tasks are run on a bus-based multicore processor. Each task is split in *execution* and *communication* slices. The mapping problem is thus turned into a slice assignment problem of where each core is modeled by a series of execution slots of equal lengths and the shared bus is also modeled by communication

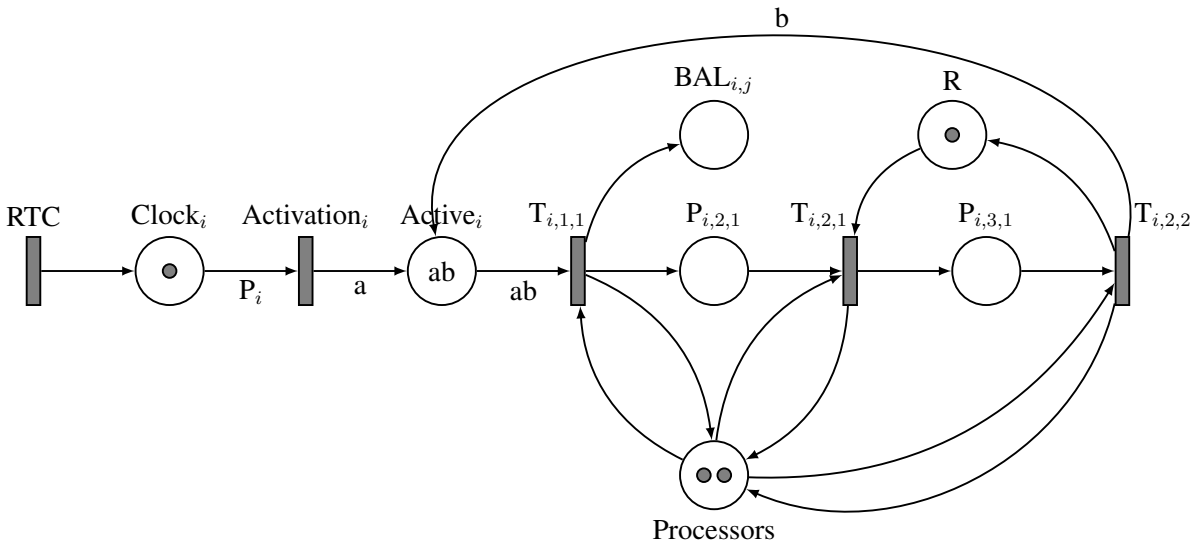


Figure 4.2: Example of task modeled using Petri nets

slots having the same length. The problem is then to assign execution slices to core slots and communication slices to bus slots. CPU utilization is constrained by checking that the sum of the WCET of execution slices assigned to a slot is not superior to the length of the slot. Precedences are enforced using constraints on the position of the slots to which execution slices are assigned. Similar constraints are provided to account for memory and communication limitations. Although not directly applicable to a many-core processor and thus to our problem, such an approach based on Constraint Programming seems to be capable of scaling up to industrial-sized problems. This demonstrated scalability was one of the main reasons for investigating solution based on Constraint Programming when designing our mapping tool as described in Chapter 8.

4.1.4.2 Makespan-optimization

Outside from the real-time community, the problem of scheduling off-line dependent tasksets on multi-core processors has been thoroughly studied. As a main difference, the tasks models that are commonly considered do not necessarily represent periodic tasks or include any deadline constraints. In general, the approaches rather strive for reducing the total completion time of a DAG task, commonly referred to as the *makespan-optimization* problem.

Definition 7 (Makespan)**.** *The makespan is the time difference between the beginning of the first running sub-task and the end of the last running sub-task in a specific schedule. The optimal (shortest) makespan of a task is equal to the length of its critical path.*

In [130], Kwok and Ahmad overviewed the plethora of static scheduling algorithms for allocating non-periodic DAG tasks on multi-core processor. Given the NP-completeness of the scheduling problem [131], most of the proposed solutions consider heuristics based on the *list scheduling* technique [132, 133, 134, 135, 136, 137, 138, 139, 140]. The idea is to keep a priority queue of the sub-tasks that can be scheduled and to iteratively pick τ_i^j the first element of the queue, place it on the core on which it can start the earliest and add to the queue the children sub-tasks of τ_i^j. Usually, the scheduling algorithms vary essentially in the way of assigning priorities to sub-tasks.

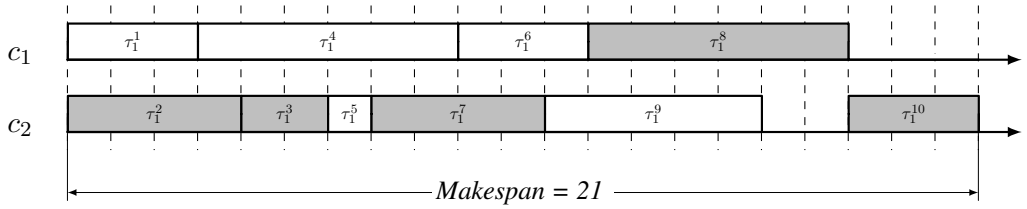


Figure 4.3: Example of schedule produced by HLFET on two cores. The sub-tasks of the critical path are colored in gray.

Some algorithms assign static priorities to sub-tasks such as *HLF* (or *Highest Level First*) [140], *LPT* (or *Longest Processing Time*) [141] and *CP* (or *Critical Path*) [142]. More recent approaches usually assign priorities *dynamically*. In these approaches, the priorities are not fixed a priori and all priorities in the scheduling queue are re-computed every time a sub-task is appended to it.

Two common timing attributes for assigning priorities to sub-tasks are the *b-level* and the *t-level* defined as:

t-level : the t-level of a sub-task τ_i^j is the longest path from an entry node to τ_i^j. The t-level can be seen as the earliest start time of a sub-task.

b-level : the b-level of a sub-task τ_i^j is the longest path from τ_i^j to an exit node. All b-levels are upper bounded by the length of the Critical Path of the DAG.

These timing attributes are used in many popular list-scheduling algorithms such as the *Highest Level First with Estimated Times* (or *HLFET*) algorithm [132], the *Insertion Scheduling Heuristic* (or *ISH*) [143] or the *Modified Critical Path* (or *MPC*) algorithm [144]. A description of HLFET is provided in Algorithm 1.

Algorithm 1 (HLFET)**.**

1. *Compute the* b-level *of each sub-tasks*
2. *Put ready sub-tasks into the priority queue. Highest priority is given to sub-tasks with largest b-level.*
3. *Schedule the sub-task with highest priority to the core on which it can start the earliest.*
4. *Update the scheduling queue with the sub-tasks that are now ready.*
5. *Repeat 3 and 4 until all sub-tasks are scheduled.*

Example 15 (Application of HLFET)**.** *Let us consider 10 sub-tasks* $\{\tau_1^1, \ldots, \tau_1^{10}\}$ *constrained by the precedence relations depicted in Figure 4.1. The WCETs of the sub-tasks are those defined in Table 4.1. Periods and offsets are ignored. Table 4.2 shows an example of b-level, start dates* ($\widetilde{s_1^i}$) *and core* (c_i) *assignment of sub-tasks when using HLFET on a dual-core target. The two cores are denoted* c_1 *and* c_2*. As depicted in Figure 4.3, the makespan of this schedule is 21.*

Overall, all those makespan-oriented scheduling algorithms consider single task problems without real-time constraints. Many heuristics have been proposed, mostly using a list-scheduling approach, in order to assign sub-tasks to cores. Yet, applying such approaches is impossible in a hard-real time context without profound modifications of the models and algorithms to manage

Sub-task	b-level	$\widetilde{s_1^i}$	Core
τ_1^1	18	0	c_1
τ_1^2	19	0	c_2
τ_1^3	15	4	c_2
τ_1^4	18	3	c_1
τ_1^5	14	6	c_2
τ_1^6	12	9	c_1
τ_1^7	13	7	c_2
τ_1^8	9	12	c_1
τ_1^9	8	11	c_2
τ_1^{10}	3	18	c_2

Table 4.2: Example of schedule produced by HLFET on two cores

multiple parallel tasks with deadline constraints. Moreover, these techniques usually don't apply to many-core architectures with distributed memory schemes since communications must be achieved explicitly by software and that both the cores and the NoC should be treated during the scheduling phase. For this reason, we explore in the next sections the contributions that were made regarding the scheduling of parallel tasks on such distributed architectures.

4.2 Off-line mapping on distributed architectures

In the previous section, we overviewed the approaches that have been proposed to map DAG tasks on multi-core processors. Yet, although providing good results, these approaches are usually not applicable to many-core targets such as the KALRAY MPPA®-256. Indeed, the communications on multi-cores rely on shared memory while using NoC-based many-core processors requires to manage explicit communications using message-passing techniques when distributing an application over the whole chip. In this section, we overview the work that has been done to map real-time applications on the KALRAY MPPA®-256, on other many-core processors and also more general (but still relevant) work on time-triggered macroscopic-sized distributed systems.

4.2.1 Mapping on the KALRAY MPPA®-256

In [47], Giannopoulou *et al.* propose a *mixed-criticality* scheduling policy for the KALRAY MPPA®-256 based on the *Flexible Time-Triggered Scheduling* policy (or *FTTS*) originally introduced in [145]. A FTTS schedule consists of succession of *frames* of fixed duration. The sequence of frames is repeated over a *scheduling cycle* whose length equals the hyperperiod of the taskset. Each frame is composed of several *sub-frames* of variable length during which tasks can execute. In a frame, tasks of high criticality levels are assigned to the firsts sub-frames while lower-criticality tasks are assigned ending sub-frames. By doing so, high-criticality tasks can be executed easily. Lower-criticality tasks coming after can be either fully executed, executed in degraded mode or just canceled depending on the remaining time in the current frame. Doing so enables to guarantee the safe execution of high criticality tasks while introducing some flexibility to efficiently use the hardware resources. Yet, the underlying assumption behind the FTTS is that the WCET of tasks are in the same order of magnitude to minimize idle states within sub-frames.

Example 16 (FTTS scheduling policy)**.** *Figure 4.4 depicts 2 cycles of an FTTS schedule. Tasks with high priorities are colored in dark gray. Low-priority tasks are in light gray. During the first*

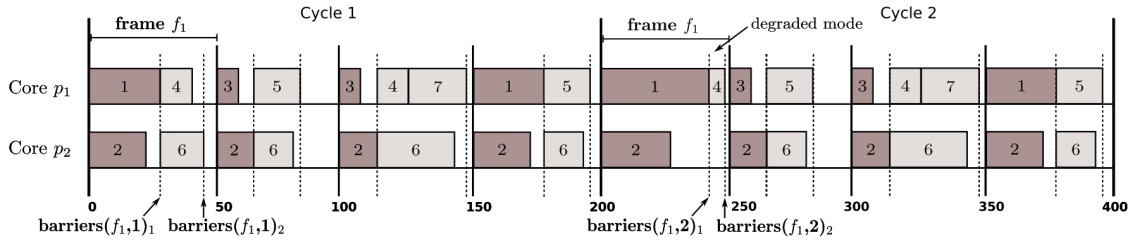


Figure 4.4: Example of FTTS schedule on 2 cycles (extracted from [47]).

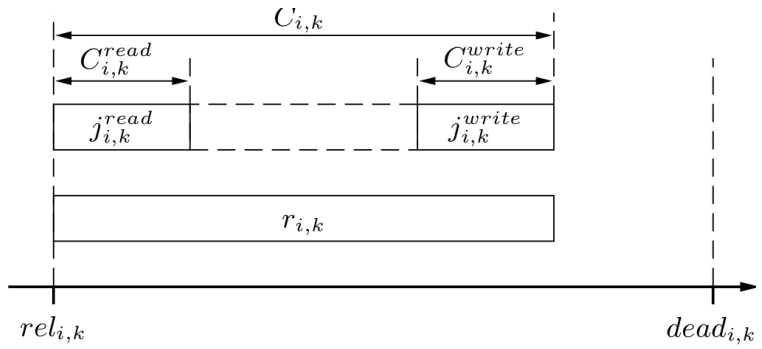


Figure 4.5: Read, write and execute phases of Runnables (extracted from [146]).

frame of the second cycle, task 1 executes for a long time, thus leading to the execution of task 4 in degraded mode to keep the total execution time within the span of frame 1. One may note that the division into no overlapping sub-frame is implemented using barriers.

In addition, in [47], an analysis of the memory interferences inside compute cluster is proposed together with a method to compute WCTT of packets on the NoC using Network Calculus. Moreover, a mapping heuristic is detailed to optimize the utilization of cores, local memory banks and the NoC. Unfortunately, this work has not been implemented on any real target and still lacks experimental evaluation. Especially, the complexity of the run-time supporting the execution of frames and sub-frames is not detailed.

In [146], Becker *et al.* proposed a method to schedule tasks inside a single cluster of the KALRAY MPPA®-256. They proposed an ILP formulation of the mapping problem together with sub-optimal heuristics enabling the approach to scale from small programs to actual industrial-sized applications. The authors eliminated inter-core interferences during execution thanks to privatization of local memory banks to each core and assumed a *read-execute-write* semantics of the application similar to the AER execution model previously described in Section 3.4.2. They consider only automotive applications following an AUTOSAR model where each task τ_i (also denoted as a *Runnable*) is a sequential piece of code characterized by a period T_i , a WCET C_i and memory footprint S_i. As shown in Figure 4.5, using the read-execute-write semantics, the WCET of a task is decomposed into three phases $C_i = C_i^{rd} + C_i^{ex} + C_i^{wr}$:

- the read phase of duration C_i^{rd} during which the code and data of the runnable are copied from the external RAM into a local cluster's memory bank;

- the execution phase of duration C_i^{ex} during which the code of the task is executed by a PE;

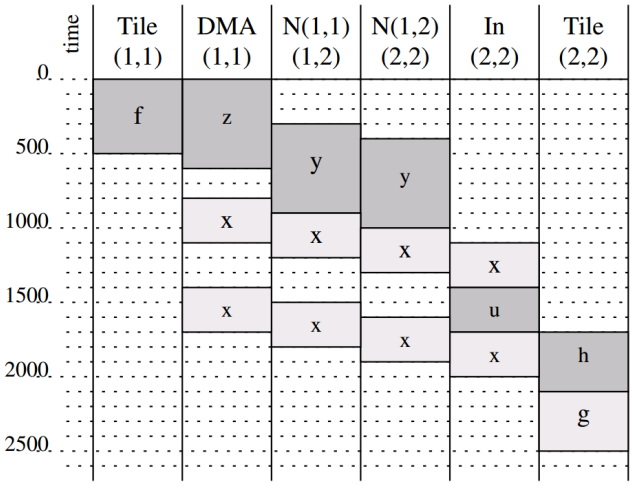


Figure 4.6: Example of scheduling table produced by LoPhT (extracted from [147]).

- and the write phase of duration C_i^{wr} during which the data produced by the task are committed to the external RAM.

In [146], the authors modeled each task by three jobs corresponding to the three phases. Then, they proposed different solutions to the problem of assigning execution jobs to cores and read and write jobs to non-overlapping time frames. In addition, communication between tasks are ensured using a dedicated local memory bank which statically stores all the data that will be exchanged. In doing so, the communication model of the application remains implicit and the applicability of the approach is thus limited to single-cluster applications. In addition, bounds on C_i^{rd} and C_i^{wr} are computed while assuming no competing accesses to the external RAM thus limiting the applicability of the approach to the execution of at most 2 applications (1 per DDR-SDRAM controller).

4.2.2 Mapping on other many-core architectures

Carle *et al.* [147] proposed an efficient allocation and scheduling tool called *LoPhT* for computing static mapping of dataflow applications on NoC-based many-core processors. They provide several heuristics computing global scheduling tables for tasks execution on cores and for communications on NoC resources in a time-triggered fashion. An example of such a scheduling table is depicted in Figure 4.6. The approach is evaluated over an image processing application composed of 32 tasks and exhibits not only a correct execution regarding real-time bounds but also more generally improved performance and latency in comparison with hand-made parallelization. Unfortunately, several issues still need to be tackled for this approach to be applicable for our problem. Firstly only the case of one single DAG task is addressed and no support is provided for tasks with multiple periods. Secondly, accesses to external RAM are not taken into account. And finally, the feasibility still needs to be demonstrated with an implementation on a real target.

In [148], Puffitsch *et al.* presented a method to map dependent tasksets on many-core platforms. They propose a general execution model enforcing predictable execution of applications, and they instantiate it for three different hardware platforms: the Intel SCC [14], the Texas Instrument TMS320C6678 [108] and the Mellanox TILEmpower-Gx36 [149]. In addition, they provide a CP-based approach to map "*reasonably large*" tasksets with extended precedence constraints on the

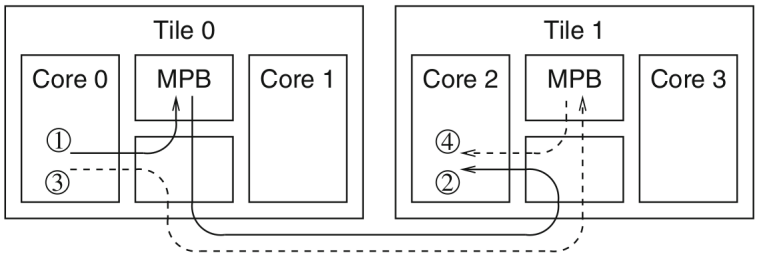


Figure 4.7: Example of *pull* (steps ① and ②) and *push* (steps ③ and ④) operations on the Intel SCC [14] (extracted from [148]).

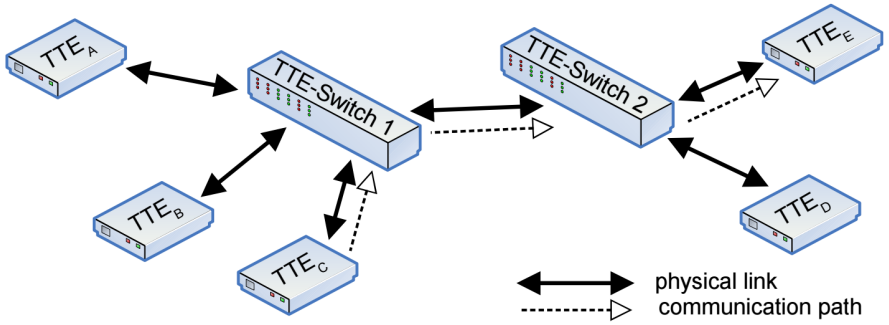


Figure 4.8: Example of TTEthernet network with 5 end-systems and 2 switches (extracted from [150]).

three targets, thus enabling the utilization of a formal approach rather than sub-optimal heuristics for industrial-sized problems. In [148], the authors assume time-bounded NoC communications relying on the notion of *Message Passing Areas* (of *MPAs*) inspired by the *Message Passing Buffers* of the Intel SCC. In this model, tasks can either *push* data in the remote MPA of another task or *pull* data from a remote MPA into its local memory as shown in Figure 4.7. The time required for these operations is considered as bounded and a practical bound is obtained using stressing benchmarks on the three platforms. Finally, the scalability of the approach is demonstrated using realistic benchmarks comprising hundreds of tasks.

Overall, the approach presented here enables a predictable execution of large real-time applications on many-core processors and is thus of major interest for our problem. Unfortunately, several additional issues need to be addressed in order to adapt it to safety-critical applications executing on the KALRAY MPPA®-256. Firstly, the execution model that is assumed in [148] requires profound modifications to be applicable to KALRAY MPPA®-256. On the KALRAY MPPA®-256, data are exchanged over the NoC thanks to DMA transactions configured by software. In this context, the *pull* operation would require heavy software support, and probably an important overhead that should be accounted in the mapping problem. In addition, bounding communication durations by considering the absolute worst case interference pattern on the NoC seems unadapted on the KALRAY MPPA®-256 because of the possibility of deadlocks (cf. Section 3.3.1). And finally, it may be argued that measurement-based approaches for computing WCTTs would not be sufficient to certify safety-critical software for the same reasons motivating the choice of static analysis over measurement-based techniques when computing WCETs (cf. Section 3.1).

4.2.3 Core and network co-scheduling

Although applied to macroscopic-sized networked systems such as the one depicted in Figure 4.8, the work of Craciunas *et al.* [150] shares many common aspects with our problem. They propose to compute simultaneously the schedules of tasks on end-systems and the schedules of communications over the virtual channels of a *TTEthernet* network [151] using *Satisfiability Modulo Theories* (or *SMT*) and *Mixed Integer Programming* (or *MIP*) solvers. When focusing on many-core processors, the scheduling problem falls in a somewhat similar category where both tasks' and communications' schedule are inter-dependent. Unfortunately, the approach presented in [150] only partially solves our problem. The authors focus on a pure *scheduling* issue to answer the question of *when* tasks and communications are executed. The *mapping* problem to decide *where* should each task be assigned is not relevant in their context since the function of each end-system is assumed to be known a priori. In our case, no such pre-assignment of tasks to cores is comprised into the model. Therefore, a pre-mapping technique is required to apply the method. However, one may note that splitting the global problem into two sub-problems of mapping and scheduling solved sequentially does not allow to cover the whole solution space.

4.3 Summary

In this chapter, we overviewed the various methods that where proposed to map and schedule dependent tasksets on distributed and many-core architectures. We identified in the DAG task model many similarities with our application model. Unfortunately, most of previous works regarding this model either focus on on-line scheduling techniques or are hardly directly applicable to a real-time context, to a many-core architecture or to an industrial context. We overviewed several other techniques focusing on hard-real time scheduling on many-core processors, including approaches targeting the KALRAY MPPA®-256. While these approaches show promising results regarding the utilizability of many-core processors in real-time systems, none considers simultaneously an application model similar to ours together with the aerospace and industrial constraints we previously described in Chapter 2. The second axis of contribution of this thesis is thus a novel mapping and scheduling technique solving both issues at once.

Part III

Contributions

Chapter 5

Execution model for the KALRAY MPPA®-256

Contents

5.1 Bounds on shared resources access time 85
5.1.1 SRAM bank access time 85
5.1.2 WCTT on the NoC 86
5.1.3 DRAM access time 87
5.1.4 Summary 88
5.2 Mitigation of the interferences 88
5.2.1 Definition of the execution model 89
5.2.2 Motivations 90
5.3 Summary 92

In this chapter, based on the detailed description of the KALRAY MPPA®-256 of Chapter 2, we provide mathematical expressions enabling the computation of interference penalties when accessing shared resources such as the local SRAM banks, the Network on Chip or the external DDR-SDRAM. We assess the cost of an approach considering worst-case pathological competitors to safely bound the WCET of an application independently from its co-runners. Finally, we provide an execution model which mitigates this cost by restricting the behaviour of applications when sharing resources.

5.1 Bounds on shared resources access time

In this section, we provide models of the shared resources such as the SRAM, the NoC the DDR-SDRAM. The idea is firstly to identify the possibilities of bounding access time to those resources without making assumptions on the applications behaviour; and secondly, to estimate the amount of time that could be saved by reducing the access time margins by making some assumptions. The models presented here are derived from the composition rules that we originally presented in [71].

5.1.1 SRAM bank access time

Accesses to SRAM banks in compute clusters are ruled by a complex arbitration policy. Fortunately, the access protocol for SRAM is simple and no refresh is required. Assuming no DMA

reception and a service time of $T_{access}^{SRAM} = 10 \times t_{ck}$ (with t_{ck} the duration of a clock cycle) for a single memory access of 8 bytes (9 cycles of latency and 1 cycle per double-word consecutively fetched), the maximum duration of a memory transaction of S_{trans} bytes initiated by a PE, the RM, the DSU or the DMA in emission can be upper-bounded with equation 5.1 when the number of competitors accessing the same bank is greater than 1.

$$T_{trans}^{SRAM}(S_{trans}, N_{comp}^{SRAM}) = \left\lceil \frac{S_{trans}}{W_{bus}^{SRAM}} \right\rceil \times N_{comp}^{SRAM} \times T_{access}^{SRAM} \tag{5.1}$$

with: W_{bus}^{SRAM} the width of the memory bus (8 bytes on the KALRAY MPPA®-256) and N_{comp}^{SRAM} a *virtual* number of competitors which depends on the initiator (a PE, the RM or the DMA).

The N_{comp}^{SRAM} value depends on the initiator. For example, a cache miss from a PE should be considered with $N_{comp}^{SRAM} = 64$. Indeed, the first Round-Robin arbiter at cache level will, in the worst case, forward to the next level of arbitration one out of two requests. The second arbiter will forward one out of sixteen requests and the last level one out of two. Eventually, in the worst case, one out of sixty-four requests issued to a bank will be initiated by the PE's cache under consideration. With a similar approach, $N_{comp}^{SRAM} = 6$ for the RM, the DSU and the DMA Tx.

Example 17 (Cost of accessing the local SRAM)**.** *Let us assume a PE reading 64 bytes from one local SRAM bank. The transaction's address is aligned on a multiple of 8 bytes. With* $S_{trans} = 64$ *and* $N_{comp}^{SRAM} = 64$*, this transaction can take in the absolute worst case* $T_{trans}^{SRAM} = 5120$ *clock cycles. If the PE is guaranteed to access the bank without competitors, the equation 5.1 does not apply any more and the transaction's duration will last for* 17 *cycles. Indeed* $N_{comp}^{SRAM} = 1$ *and* T_{access}^{SRAM} *is payed only for the first double-word access (the others will follow consecutively without repaying for* T_{access}^{SRAM} *every time).*

The only way to *safely* bound SRAM access time of PEs requires to consider the absolute worst bank-level competition if no assumption can be made on workloads ran by other PEs. As explained in Example 17, the cost of worst case competition can be prohibitive. Moreover, in case of DMA receptions, T_{trans}^{SRAM} should be extended with the maximum stalling time because of DMA Rx transactions. This stalling time depends on the NoC parameters and the memory areas assigned to Rx channels. Yet, any bound on the DMA Rx interference penalty can be summed to T_{trans}^{SRAM} safely thanks to the property of full timing compositionality of the k1b cores.

Anyway, the cost of bounding SRAM access time can be very large if the worst-case competition needs to be assumed. Reducing the cost of such an assumption will be one of the goal of our execution model.

5.1.2 WCTT on the NoC

Bounding traversal time on the NoC depends both on the packetization procedure and the arbitration operated in switches. We analyze these two problems in the following sub-sections.

5.1.2.1 Packetization

With N_{bytes}^{flit} the number of bytes in one NoC flit, the number of payload flits needed to send a buffer of S_{trans} bytes over the NoC is:

$$N_{flit}(S_{trans}) = \left\lceil \frac{S_{trans}}{N_{bytes}^{flit}} \right\rceil$$

So, assuming N_{flit}^{pkt} to be the maximum number of flits in a NoC packet, the number of packets required to send S_{trans} bytes is:

$$N_{pkt}(S_{trans}) = \left\lceil \frac{N_{flit}(S_{trans})}{N_{flit}^{pkt}} \right\rceil$$

Thus, with N_{flit}^{head} the number of flits in a NoC packet header and N_{flit}^{bbl} the number of empty flits lost in the bubble separating two consecutive packets, we can derive the total number of flits (including *wasted* flits lost between packets, if any) needed to send S_{trans} bytes as:

$$N_{flit}^{total}(S_{trans}) = N_{flit}(S_{trans}) + N_{pkt}(S_{trans}) \times N_{flit}^{head} + (N_{pkt}(S_{trans}) - 1) \times N_{flit}^{bbl}$$

5.1.2.2 Time required to cross the NoC

The time required for all the flits of a transaction of S_{trans} bytes to cross a NoC route of N_{switch} switches without conflicts is given in Equation 5.2 with Δ_L the latency of a NoC link and Δ_S the latency of a NoC switch.

$$T_{trans}^{NoC}(S_{trans}, N_{switch}) = (N_{switch} + 1) \times \Delta_L + N_{switch} \times \Delta_S + N_{flit}^{total}(S_{trans}) \tag{5.2}$$

If we do not make assumptions on the routes or the injection rates of packets, bounding the WCTT on the NoC *with conflicts* can be impossible because of the possibility of deadlocks as explained in Section 3.3.1.1. What we emphasize here is the need for precise assumptions on the NoC access patterns to be able to use it under real-time constraints. As an example, we refer the reader to the contributions of Dupont de Dinechin *et al.* [46] and Giannopoulou *et al.* [47] for WCTT computation using Network Calculus where NoC accesses were assumed to be rate-limited at source and to follow pre-assigned routes.

Overall, bounding WCTTs on the NoC cannot be done without assumption on its utilization by applications. Providing users with a framework enabling predictable NoC usage will be the second goal of our execution model.

5.1.3 DRAM access time

5.1.3.1 Model of the arbiter

The KALRAY MPPA®-256's MPFE handles a complex dynamic priority assignment policy to avoid starvations. For simplification purpose, we will assume a specific MPFE configuration with starvation counters equal to 0 for all masters. In this case, all masters always have the same priority (the highest) and are thus arbitrated in a pure Round-Robin fashion.

With W_{burst}^{DDR} the width of a DDR burst, a transaction of S_{trans} bytes will require the following number of column access requests to be fully executed:

$$N_{req}^{DDR}(S_{trans}) = \left\lceil \frac{S_{trans}}{W_{burst}^{DDR}} \right\rceil$$

Yet, assuming pure Round-Robin for the MPFE, the number of arbitration rounds required for all the $N_{req}^{DDR}(S_{trans})$ to be forwarded to the reorder core is:

$$N_{round}^{DDR}(S_{trans}, N_{comp}^{DDR}) = N_{req}^{DDR}(S_{trans}) \times N_{comp}^{DDR}$$

A request entering the reorder core is guaranteed by the hardware to be served after at most $2N_{req}^{pool} - 1$ other requests. So, with T_{req}^{DDR} the worst case service time for a DDR request, the total amount of time (without considering refreshes yet) needed to complete a full transaction can be bounded by Equation 5.3.

$$T_{trans}^{DDR}(S_{trans}, N_{comp}^{DDR}) = (N_{round}^{DDR}(S_{trans}, N_{comp}^{DDR}) + 2N_{req}^{pool} - 1) \times T_{req}^{DDR} \tag{5.3}$$

5.1.3.2 Evaluation of T_{req}^{DDR}

As explained in Section 3.2.2, a complex access protocol is used to deal with DDR-SDRAM systems. The service time of a DDR-SDRAM request highly depends on the current state of the chip, and thus depends on the history of accesses that changed this state. The worst case access time of a DRAM request occurs in case of a row-miss read request following a write request. In this case, the controller must: 1) wait for the write completion (t_{WR}); 2) issue a *PRE* command (t_{RP}) on the currently opened row; 3) activate the correct row (t_{RCD}) and 4) issue the *RD* command ($t_{CAS} + t_{burst}$). The worst case service time for a request is given in Equation 5.4.

$$T_{req}^{DDR} = t_{WR} + t_{RP} + t_{RCD} + t_{CAS} + t_{burst} \tag{5.4}$$

Example 18 (Cost of accessing the DDR-SDRAM)**.** *Let us assume 192 bytes of data to be written in a bank of the DDR-SDRAM. With the values of Table 5.1, and assuming 4 DMAs competing for accessing the same bank,* $N_{round}^{DDR} = 12$*. With the DDR timing values of Table 3.1,* $T_{req}^{DDR} = 67.5ns$*. Thus,* $T_{trans}^{DDR} \approx 1.8\mu s$*. If only one DMA was accessing the bank without any competitor, the row activation would need to by payed only once, thus leading to* $T_{trans}^{DDR} = 75.5ns$*.*

As explained in Example 18, accessing the DDR-SDRAM in the worst-case competition scenario can lead to very inefficient use of the memory. Indeed, competitors accessing different rows of the same bank will force the controller to repeatedly precharge and activate rows, thus reducing the time used to effectively transfer data. Reducing the overhead involved by such competition while keeping bounds on memory transactions sound will be the third goal of our execution model.

5.1.4 Summary

Hosting several applications on a single target while making them independent from each other requires for each of them to assume worst-case competitors when accessing shared resources. In the previous sections, we have seen that such an assumption can lead to very high costs to bound the durations of accesses to the local SRAM, the NoC or the DDR-SDRAM. In order to keep co-hosting costs reasonable, we propose to artificially isolate applications by enforcing specific behaviours on shared resources. By mastering concurrent applications at the resource level, we aim at avoiding the corner cases for which accessing resources is too expensive, thus enabling better usage of the hardware overall.

5.2 Mitigation of the interferences

In order to mitigate the interferences when accessing shared resources, we propose an execution model which constrains the behaviour of applications in exchange of improving their predictability

PARAMETER	MEANING	VALUE
	SRAM	
t_{ck}	Duration of a clock cycle	$1/400$MHz = 2.5ns
T^{SRAM}_{access}	Duration of a single SRAM access	10 cycles
W^{SRAM}_{bus}	Width of the SRAM access bus	8 bytes
S^{SRAM}_{bank}	Size of a local SRAM bank	128 KiB
	NoC	
Δ_L	Latency of a NoC link (not flying)	See [17]
Δ_S	Latency of a NoC switch	See [17]
N^{flit}_{bytes}	Number of bytes in a NoC flit	4 bytes
N^{pkt}_{flit}	Number of flits (max.) in a NoC packet	Configurable. <62
N^{head}_{flit}	Default number of flits in a packet header	2
	DDR3-SDRAM	
N^{pool}_{req}	Size of the reordering pool	8 requests
T^{DDR}_{req}	Worst case service time for a DRAM request	Depends on module
W^{DDR}_{col}	Width of a DRAM column / burst size	64 bytes

Table 5.1: Notations and values of the KALRAY MPPA®-256 hardware parameters

and enabling better access to shared resources. The idea is to provide temporally isolated environments, denoted as *partitions*, to applications.

Definition 8 (Partition)**.** *A partition is an execution environment in which the temporal behaviour of an application does not depend on the behaviour of applications not belonging to the same partition.*

Our execution model imposes specific access patterns to sensible resources that are likely to be shared by several partitions in order to enforce interference-free executions of the isolated applications. By doing so, we aim at executing partitions in a *time-composable* manner.

5.2.1 Definition of the execution model

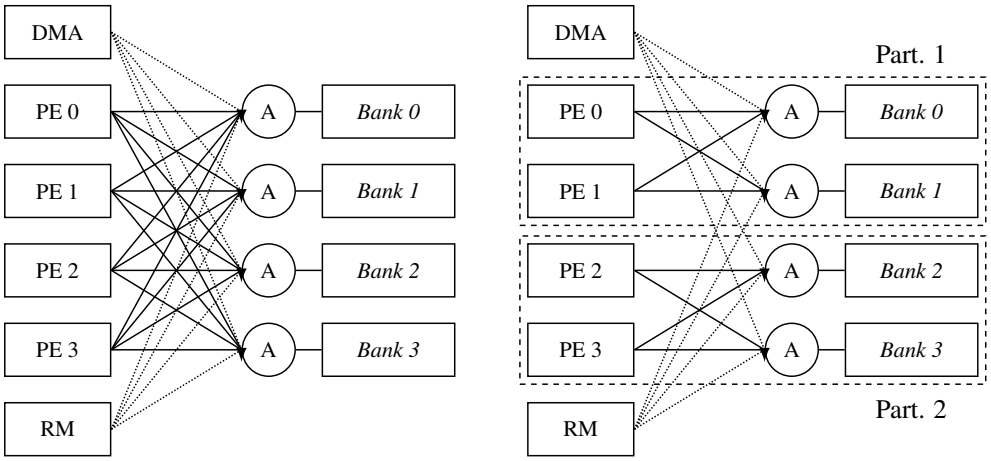
We propose four rules constraining accesses to the three main levels where interferences can occur on the KALRAY MPPA®-256, namely the local SRAM, the NoC and the DDR-SDRAM.

Rule 1. *Any PE (respectively any local SRAM bank) inside any compute cluster can be used by at most one partition.*

Rule 2. *Communications over the NoC must respect a pre-computed Time-Division Multiplexing schedule and avoid conflicts by either taking non-overlapping routes or accessing the shared route at different time. The NoC schedule must ensure that when a communication uses a route, it is completely reserved at the time of utilization.*

Rule 3. *The memory areas to be sent over the NoC must be defined off-line.*

Rule 4. *Any bank of the external DDR-SDRAM can be shared by two or more partitions if and only if they never access it simultaneously.*



(a) Without Rule 1 (b) With Rule 1

Figure 5.1: Example of application of the Rule 1 of the execution model on simplified compute cluster architecture with 2 partitions

5.2.2 Motivations

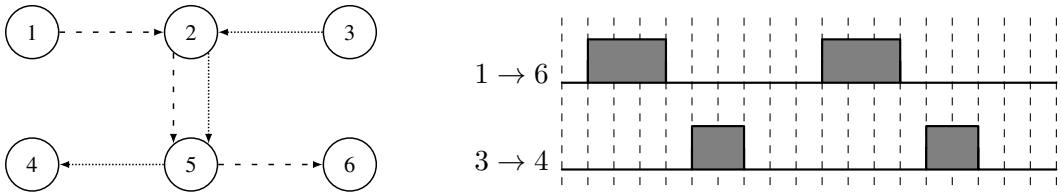
5.2.2.1 Rule 1

With Rule 1, we ensure that some code running on a PE will not suffer from local SRAM interferences caused by PEs of other partitions. By doing so, the N_{comp}^{SRAM} parameter of Equation 5.1 can be adjusted to account only for the PEs generating *friend* traffic, that is, PEs running code of the same partition. This does not only help to lower the local SRAM interference penalties when computing WCETs, but this also brings complete isolation between partitions. Yet, local SRAM accesses from the RM and the DMA can still be conflicting with the PEs. DMA accesses will occur only to send or receive data that belongs to a partition. Conflicts with PEs of the same partition do not break the property of temporal isolation. Similarly, as we will detail in Chapter 7, the RM will be executing a custom hypervisor which may access partition-reserved memory addresses only to achieve partition-related work.

Example 19 (Application of Rule 1)**.** *An example of application of the Rule 1 is shown in Figure 5.1. A simplified compute cluster architecture is presented in Figure 5.1a with 4 PEs, 4 banks of memory, the RM and the DMA. Figure 5.1b shows the same cluster when the Rule 1 is applied to split the compute cluster into two partitions. PEs 0 and 1 are not able to access the banks 2 and 3 allocated to the partition 2. Symmetrically, PEs 2 and 3 cannot access banks 0 and 1 reserved for the partition 1. In this case, any task running on the PE 0 (for example) may suffer only from "friend" memory interferences when in competition with the PE 1, the RM or the DMA. In this example, the partitioning has arbitrarily been chosen as symmetric (2 cores - 2 banks per partition) but any configuration could have been possible (1 core - 3 banks, ...) depending on the needs of applications.*

5.2.2.2 Rule 2

Enforcing a Time-Driven schedule of the NoC enables an easy and efficient estimation of WCTTs of packets. Indeed, assuming that no conflict can occur on the NoC, the traversal time of packets can be estimated accurately using simple models such as Equation 5.2. However, these gains are



(a) Simplified NoC (b) Communication schedule

Figure 5.2: Example of application of the Rule 2 of the execution model on simplified NoC with 2 communications competing for a shared NoC resource ② → ⑤

obtained at the costs of inflexible communication scenarios and additional efforts for pre-computing schedules that actually exhibit the no-overlapping property.

The approaches considering asynchronous and rate-limited sources such as [46] and [47] are often argued to provide more flexibility and good guaranteed bandwidths. However, our choice of considering a TDM schedule and synchronous sources over Network Calculus and asynchronous sources is motivated by the two following reasons:

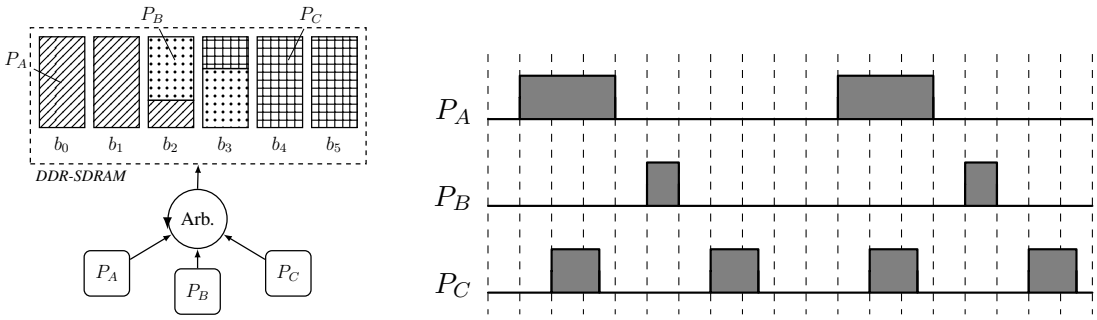
1. We aim at providing fully temporally isolated execution environments to partitions. Approaches with asynchronous sources often rely on the timing *compositionality* of the system to account for the worst case communication conflict. By doing so, the actual time required for a packet to cross the network, even if safely bounded, does depend on the behaviour of the co-running applications. By using a TDM schedule, we aim at providing a property of timing *composability* which is better suited to ensure strict isolation.

2. We aim at being able to efficiently execute multi-cluster applications in multi-cluster partitions to fully benefit from the massively parallel computational power of the target. To do so, our ability to ensure short guaranteed *latencies* for data exchange between clusters will be of major importance. TDM schedules have been shown in [152] to provide better results for this particular criteria while Network Calculus on asynchronous sources seems preferable when *bandwidth* guarantees are needed.

Example 20 (Application of Rule 2)**.** *A simplified NoC topology with 6 nodes is depicted in Figure 5.2a where two communications compete to access the ② → ⑤ NoC resource. Figure 5.2b shows an example of correct TDM schedule of the two communications where temporal partitioning ensures the conflicts avoidance.*

5.2.2.3 Rule 3

Although meeting this rule is not absolutely necessary to ensure full temporal isolation, it has several interesting qualities. Firstly, knowing a priori the buffers that will be sent during the strictly periodic slot imposed by Rule 2 enables to *verify* off-line the coherency of a partition. The size of the buffer to be sent can be compared with the width of the TDM slots using Equation 5.2 to check whether the time slot is sufficiently large for all the data to be sent. Secondly, as we will show in Chapter 7, Rule 3 greatly simplifies the implementation of the hypervisor in charge of enforcing the respect of the execution model. In particular, it helps in narrowing the WCET of the hypervisor which we will demonstrate to be of major importance for performance issues. And finally, knowing where and when DMA Rx transactions will occur enables accurate estimations of the stalling time



(a) DDR-SDRAM accesses

(b) DDR-SDRAM access schedule

Figure 5.3: Example of application of the Rule 4 of the execution model on simplified DDR-SDRAM arbiter with 3 masters P_A, P_B and P_C competing to access the memory

that PEs will suffer when accessing their local SRAM banks, thus solving the problem identified in Section 5.1.1.

5.2.2.4 Rule 4

With Rule 4 we eliminate the possibility of DDR-SDRAM masters conflicting to access different rows of the same bank. Thanks to this, the estimation of T_{req}^{DDR} can be adapted to take into account that a master issuing a series of requests targeting contiguous memory areas in the same row will have to pay for the activation of this row only for the *first* request and not for *all* of them. This provides both performance isolation by exploiting the parallelism of banked DDR-SDRAM but also improves the overall memory throughput by helping the memory controller issue inexpensive requests which will not change rows inside banks due to competitors accessing other rows .

Example 21 (Application of Rule 4)**.** *Figure 5.3a shows an example of 3 partitions P_A, P_B and P_C accessing the external DDR-SDRAM composed of 6 banks $b_0, \ldots, b_5$. In particular, one may note that the bank b_2 is shared by P_A and P_B while b_3 is shared by P_B and P_C. Figure 5.3b shows an example of correct schedule of memory transaction where the accesses of P_A and P_C are allowed to overlap in time since they do not share any bank. On the other hand, the transaction from P_B are never overlapping with P_A and P_C to ensure competition-free accesses to banks b_2 and b_3.*

5.3 Summary

In this chapter, we analyzed the architecture of the KALRAY MPPA®-256 in details with a particular focus on arbitration mechanisms for the access to shared resources. Based on this analysis, we proposed, when it was possible, mathematical expressions to compute worst-case interference penalties for the access to the local SRAM banks, the NoC and the DDR-SDRAM. We showed that, in most cases, considering the pathological worst-case pattern of accesses to the resources leads to prohibitive penalties while, in the remaining cases, computing such bounds was simply not possible without eliminating the worst-case thanks to assumptions on the system model. Based on these conclusions, we proposed four rules composing an execution model that mitigates or even eliminates undesired interferences between applications. This execution model relies on pure spatial partitioning inside compute clusters and on hybrid partitioning for the NoC and the DDR-SDRAM

to provide temporally isolated *partitions* in which applications can run. Overall, our approach relies on off-line reservation of resources by partitions. We formalize this using a notion of *resource budget* that is allocated to a partition. Budgets are key elements for the framework proposed in this thesis. This first contribution has been published in [71]. The next chapter provides a formal definition of the notions of partitions and budgets and exposes the workflow relying on it.

Chapter 6

Integration framework

Contents

6.1 Input parameters 95
6.1.1 Platform model 95
6.1.2 Resource budget 96
6.2 Off-line computations 97
6.2.1 Validate 97
6.2.2 Allocate 98
6.3 On-line support 100
6.4 Summary 100

In this chapter, we propose a framework enabling the integration of multiple applications on the KALRAY MPPA®-256 following the execution model defined in Chapter 5 and thus, in a time-composable manner. Overall, the whole framework relies on the property of *temporal isolation* that is provided to partitions. As shown in Figure 6.1, we propose to integrate partitions using a *work-flow* going from the applications' models down to an executable that runs on the target in a predictable manner. In this work-flow, several off-line computations need to be achieved in order to build scheduling tables that will be enforced at run-time on the KALRAY MPPA®-256 with a hypervisor. We will firstly present what the inputs of this work-flow are and we will explain and formalize the notion of *resource budget* of a partition. Then, we will overview the two different operations that are computed off-line based on the models of partitions. Finally, we will detail the purpose of having on-line support for the partitions using a hypervisor.

6.1 Input parameters

The purpose of the work-flow shown in Figure 6.1 is to integrate temporally isolated partitions on the KALRAY MPPA®-256. The inputs of this workflow include a model of the hardware platform and a set of partitions. Each partition is composed of an application and a *resource budget*. We detail how these inputs are structured below.

6.1.1 Platform model

The hardware target that is considered in this thesis is the KALRAY MPPA®-256. We already detailed its architecture in Section 2.2 and provided models of its shared resources in Section 5.1. For

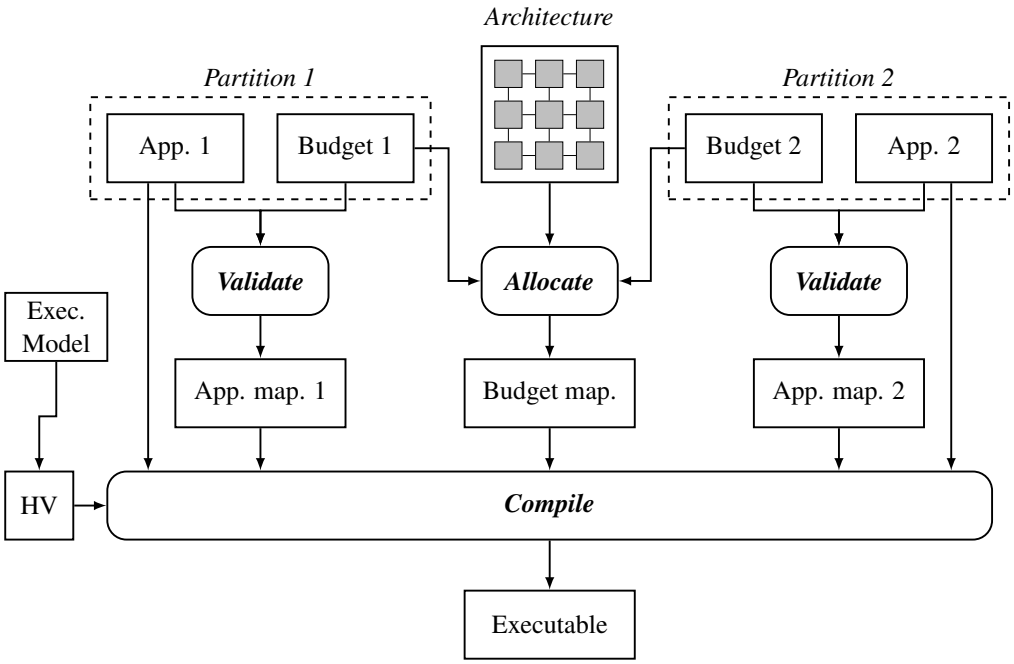


Figure 6.1: Integration of partitions on the KALRAY MPPA®-256 work-flow

convenience, we formalize the KALRAY MPPA®-256 as a tuple $< C_K, I_K, N_K, R_K >$ where:

- $C_K = \{c_K^0, \ldots, c_K^{15}\}$ represent the 16 compute clusters.
- $I_K = \{i_K^0, \ldots, i_K^{15}\}$ represent the 16 I/O RMs.
- $N_K = \{n_K^0, \ldots, n_K^{31}\}$ represent the 32 NoC nodes.
- $R_K = \bigcup R_K^{ij}$ is the set of all NoC routes with R_K^{ij} the set of routes connecting the NoC node n_K^i to the NoC node n_K^j.

6.1.2 Resource budget

As shown in Figure 6.1, a partition consists of an application provided together with a *resource budget*. We assume the applications to fit in the model of Section 2.1. A resource budget consists of the amount of resources required by an application to be executed. More formally, the resource budget of a partition P_i is defined as a tuple $< \mathcal{N}_i, \mathcal{I}_i, \mathcal{C}_i, \mathcal{B}_i >$ where:

1. $\mathcal{N}_i = \{pn_1^i, \ldots, pn_n^i\}$ is a finite set of *Partition Nodes* (or *PNs*). A PN represents a need in computational power for a partition inside a compute cluster. Each PN is associated with a number of PEs and a number of local SRAM banks. More formally:
   - $N_c : \mathcal{N}_i \mapsto \mathbb{N}$ associates each PN with the number of PEs that needs to be reserved for it in a compute cluster.
   - $N_b : \mathcal{N}_i \mapsto \mathbb{N}$ associates each PN with the number of local SRAM banks that need to be allocated to it in a compute cluster.

2. $\mathcal{I}_i = \{io_1^i, \ldots, io_o^i\}$ is a finite set of *I/O Nodes* (or *IONs*). An ION is an access point to the external DDR-SDRAM. It is materialized by a RM core located on an I/O cluster. It answers remote reads and remote writes requests between PNs and external memory banks.

3. $\mathcal{C}_i = \{pc_1^i, \ldots, pc_p^i\}$ is a finite set of *Partition Communications* (or *PCs*). A PC is a directed communication channel for PNs placed on different compute clusters (or between a PN and an ION). It is materialized as a strictly periodic access to the NoC for sending message from a PN/ION to a remote PN/ION. More formally:

   - $src : \mathcal{C}_i \mapsto \mathcal{N}_i \cup \mathcal{I}_i$ associates each PC with its source PN or ION.
   - $dst : \mathcal{C}_i \mapsto \mathcal{N}_i \cup \mathcal{I}_i$ associates each PC with its destination PN or ION.
   - $T : \mathcal{C}_i \mapsto \mathbb{N}$ provides the *period* of activation of the PC.
   - $C : \mathcal{C}_i \mapsto \mathbb{N}$ provides the *duration* of the PC's NoC access slot.
   - $O : \mathcal{C}_i \mapsto \mathbb{N}$ provides the *offset* of the PC, also denoted as the first activation date.
   - $R : \mathcal{C}_i \mapsto \mathbb{N}$ provides the maximum number of NoC nodes of the route taken by the PC. It can be used to impose a constraint of maximal latency.

4. $\mathcal{B}$ represents a number of external DDR-SDRAM banks.

All together, a partition includes the actual application and a set of *abstract* resources formalized under the notion of *budget*. Doing so enables application designers to express their needs for hardware resources and to request them early in the development process. This sort of resource reservation policy helps design, develop, test and qualify applications without having the knowledge of co-running partitions. Once a partition has its boundaries defined formally, the utilization of all other resources on the target becomes of little interest since it cannot have an impact on the application running inside the partition thanks to our execution model. Yet, the execution model requires both off-line and on-line support to be enforced. We detail how this is implemented respectively in sections 6.2 and 6.3.

6.2 Off-line computations

As shown in Figure 6.1, our work-flow includes two different off-line phases. During the *Validate* phase, the models of applications are checked over their allocated budgets. By doing so, we verify if the resources allocated to a partition are sufficient to execute the application and to meet all its constraints (both functional and non-functional). During the *Allocate* phase, the abstract resources of the budgets are matched over concrete resources on the hardware platform. In other words, the *Allocate* phase assigns PNs to clusters, PCs to NoC routes and IONs to RMs on I/O clusters in such a way that partitions do not overlap. Both the *Validate* and *Allocate* phases are currently implemented using Constraint Programming and are fully automated. We detail the operations achieved during these two phases below.

6.2.1 Validate

During the *Validate* phase, applications are checked over their budgets. Using the application model defined in Section 2.1, a budget will essentially be considered as valid if a schedule of all sub-tasks on PNs and a mapping of data on PCs can be found. More precisely, the expected inputs and outputs of the *Validate* phase are defined as follows

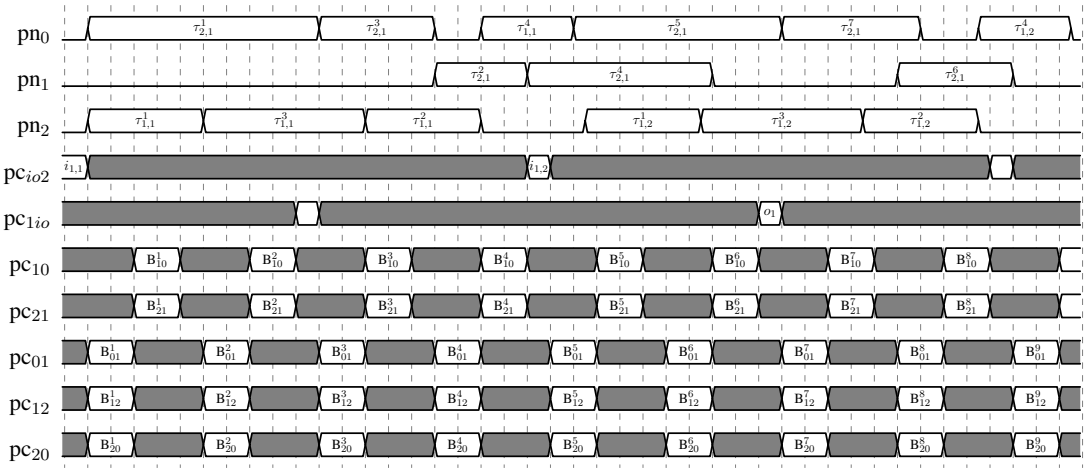


Figure 6.2: Example of schedule computed during the *Validate* phase.

Inputs: The *Validate* phase is achieved within the boundaries of a partition. Consequently, it takes as input the two components of a partition, namely the application model and the resource budget.

Example 22 (Inputs of Validation phase)**.** *Let us consider a partition in the application of Example 1 (on page 28) which should be executed. The application is composed of the 11 sub-tasks shown in Figure 2.1 with the parameters of Table 2.1. We allocate to this partition a budget composed of three PNs* pn_0, pn_1 *and* pn_2 *with* $N_c(pn_i) = 1$ *and* $N_b(pn_i) = 15$ $\forall i \in [0, 2]$*. The budget also features five PCs* $pc_{10}, pc_{21}, pc_{01}, pc_{12}$ *and* pc_{20} *where* pc_{ij} *links* pn_i *to* pn_j *with* $T(pc_{ij}) = 500$ *and* $C(pc_{ij}) = 200$ *for all PCs. Moreover* $O(pc_{01}) = O(pc_{12}) = O(pc_{20}) = 0$ *while* $O(pc_{10}) = O(pc_{21}) = 200$*. Finally, the budget includes one ION accessible through two PCs* pc_{1io} *and* pc_{io2} *with respective offsets* 900 *and* 1900 *and the same periods and durations* 2000 *and* 100*.*

Outputs: The Validation phase checks that an application fits inside its budget by computing a schedule of all sub-tasks on PNs and a mapping of data to PCs. If such a schedule exists, then the budget is considered as *valid*. Otherwise, either the application or the budget needs to be adjusted.

Example 23 (Outputs of Validation phase)**.** *With the partition of Example 22, the output schedule of the application inside its budget is shown in Figure 6.2. The communication slots of the PCs are used to send the data as follows:* $B^2_{20} = \{\delta_{b,1}\}$*;* $B^4_{20} = \{\delta_{d,1}, \delta_{e,1}\}$*;* $B^7_{20} = \{\delta_{b,2}\}$*;* $B^8_{20} = \{\delta_{d,2}, \delta_{e,2}\}$*;* $B^4_{21} = \{\delta_{d,1}\}$*;* $B^8_{21} = \{\delta_{d,2}\}$*;* $B^3_{01} = \{\delta_{g,1}, \delta_{h,1}\}$*;* $B^7_{01} = \{\delta_{m,1}, \delta_{n,1}\}$*;* $B^5_{10} = \{\delta_{i,1}\}$*;* $B^6_{10} = \{\delta_{j,1}\}$*;* $B^7_{12} = \{\delta_{l,1}\}$*. Since a correct schedule has been found, the budget of Example 22 can be considered as valid.*

In our approach, the process of computing mappings and schedules during the *Validate* phase is fully automated and is applicable to the mapping of *large* applications on complex budgets. The detail of our implementation using Constraint Programming is given in Chapter 8.

6.2.2 Allocate

During the *Allocate* phase, PNs, PCs and IONs are allocated to concrete hardware resources on the platform. The challenge during this phase is to find an allocation of the resources to partitions such

that the execution model is respected. It means for example that no PE should be allocated to more than 1 PN or that no 2 PCs overlap in space and time. In other words, the *Allocate* phase uses the following inputs and outputs.

Inputs: The inputs of the *Allocate* phase include the resource budgets of all partitions that should be integrated on the platform and a model of the hardware target. The formalism for resource budgets is the same as for the Validate phase. The model of the hardware is assumed to be equivalent to the one used in Chapter 5 and is formalized as in Section 6.1.1.

Outputs: The outputs of the *Allocate* phase consist of mapping of PNs on clusters, PCs on NoC routes, IONs on I/O RMs and the assignment of a temporal offset to partitions. More formally, the outputs of the Allocate phase are characterized by a set of mappings $M =< M_N, M_C, M_I, M_P >$ where:

- $M_N : \mathcal{N} \mapsto C_K$ associates each PN of each partition to a compute cluster;
- $M_C : \mathcal{C} \mapsto R_K$ associates each PC of each partition to a NoC route;
- $M_I : \mathcal{I} \mapsto I_K$ associates each ION of each partition to an I/O RM;
- $M_P : P \mapsto \mathbb{N}$ assigns a temporal offset to each partition.

with $\mathcal{N} = \bigcup \mathcal{N}_i, \mathcal{C} = \bigcup \mathcal{C}_i, \mathcal{I} = \bigcup \mathcal{I}_i$ and $P = \bigcup P_i$.

Example 24 (Outputs of Allocate phase)**.** *Let us consider the partition of example 22. A correct mapping of the partition's budget on the* KALRAY MPPA®*-256 is the following:*

- *for PNs:* $M_N(pn_0) = c_K^2$, $M_N(pn_1) = c_K^0$ *and* $M_N(pn_2) = c_K^8$;
- *for PCs:* $M_C(pc_{10}) = n_K^0 \rightarrow n_K^2$, $M_C(pc_{01}) = n_K^2 \rightarrow n_K^0$, $M_C(pc_{12}) = n_K^0 \rightarrow n_K^8$, $M_C(pc_{21}) = n_K^8 \rightarrow n_K^0$ *and* $M_C(pc_{20}) = n_K^8 \rightarrow n_K^{10} \rightarrow n_K^2$;
- *for IONs:* $M_I(io) = i_K^0$, $M_C(pc_{1io}) = n_K^0 \rightarrow n_K^{16}$ *and* $M_C(pc_{io2}) = n_K^{16} \rightarrow n_K^0 \rightarrow n_K^8$;
- *and the temporal offset of the partition is 0.*

Overall, the allocate phase maps PNs to compute clusters with constraints on the number of cores and the number of SRAM banks required. It also maps PCs to NoC routes, clearly depending on the assignment of the PNs to clusters, so that no NoC resource can be simultaneously used by two or more PCs. This kind of overlapping avoidance is also made possible using timing offsets for the partitions. Finally, the allocate phase assigns IONs to RMs on I/O clusters in such a way that the external memory is sufficiently large to handle the footprints of all partitions assigned to it and that the PCs targeting each ION can meet rule 4 of the execution model.

Although relatively complex, our current implementation of the Allocate phase using Constraint Programming is showing promising first results. Unfortunately, the approach still requires further investigations and evaluations to be considered as mature. For this reason, regarding the off-line computations of our work-flow, this thesis will focus only on the *Validate* phase which can be arguably be defined as already seemingly complex. Based on our current results, the implementation of a sound and robust *Allocate* phase appears clearly as a good perspective for interesting future work.

6.3 On-line support

Once mappings of applications inside their budgets have been computed and once each budget is matched over a set of concrete hardware resources, some form of run-time support is required to enforce the respect of partitions boundaries. To do so, we propose to instantiate our execution model into a low-level software component denoted as the *hypervisor*. The goal of the hypervisor is to guarantee that, while applications are running, the allocation of resources to partitions is always ensured. In particular, erroneous behaviours of applications, either because of bugs or malicious software, should be kept within the boundaries of their partitions. Among other issues, the hypervisor will be in charge of:

1. enforcing the spatial partitioning between PNs assigned to the same compute cluster;
2. managing the NoC in order to enforce the global TDM schedule that guarantees the avoidance of overlapping PCs;
3. configuring the DMAs to send the correct buffers during each PC activation;
4. setting up various hardware configurations on both compute and I/O clusters;
5. and containing applications in their partitions boundaries, even in case of erroneous behaviours.

Being capable of implementing such a hypervisor is clearly a major issue regarding our work-flow. We will demonstrate in Chapter 7 that, although complex, the implementation of such a hypervisor is feasible and that the expected property of temporal isolation between partitions can be obtained in practice.

6.4 Summary

In this chapter, we proposed a work-flow to integrate applications on the KALRAY MPPA®-256. We formalized the inputs of this work-flow with models of the applications and resource budgets of partitions. We explained how the *Validate* phase is supposed to check whether an application can fit into its partition budget. Moreover, we overviewed the purpose of the *Allocate* phase which assigns concrete hardware resources to partitions. Finally, we explained how our hypervisor is meant to support the execution of the partitions at run-time. In the next chapters, we will explain in further details how implementing such a hypervisor can be done and how Constraint Programming can be used to automate the *Validate* phase.

Chapter 7

Implementation of the execution model

Contents

7.1 Hypervisor description 101
7.1.1 Assumptions 102
7.1.2 Enforcing the execution model 103
7.2 Experimental validation 108
7.2.1 Experimental setup 108
7.2.2 Scenario 1: no interference 112
7.2.3 Scenario 2: NoC interferences 113
7.2.4 Scenario 3: DDR-SDRAM interferences 114
7.3 Discussion on design trade-offs 115
7.3.1 Memory footprint 115
7.3.2 WCET of the hypervisor 118
7.4 Summary 119

In this chapter, we provide extensive details on the implementation of our execution model on the KALRAY MPPA®-256. We describe the architecture of our hypervisor and emphasize the means used to enforce the rules of the execution model. We also describe how the soundness of this implementation was experimentally evaluated against the ROSACE case study before finally discussing the major design trade-offs for the hypervisor and their consequences on performances that can be expected by applications designer.

7.1 Hypervisor description

In this section, we present the architecture of our hypervisor and all methods used to enforce temporal isolation on the KALRAY MPPA®-256. The hypervisor runs in bare-metal on the target in order to manage all resources without unexpected behaviours due to non-mastered software components. The RM of each running cluster executes a similar instance of the hypervisor. Each instance is however configured accordingly to the PNs running locally on the cluster. In our current implementation, the hypervisor consists in a task that is periodically activated simultaneously on all RMs of all clusters. In addition, the hypervisor includes the boot procedure for the PEs, their *Interrupt Service Routines* and the micro-code of DMAs.

7.1.1 Assumptions

The hypervisor is mostly composed of a periodic task executed by RMs and of the privilege code of PEs. Yet, the hypervisor also relies on several assumptions that are, for most of them, related to general hardware configurations. We detail these assumptions below.

7.1.1.1 Hardware configuration

The enforcement of the temporal isolation between partitions is achieved with various methods detailed in Section 7.1.2. When describing these methods, we will assume that the following hardware configurations have been applied by the hypervisor at boot-time:

Local SRAM configuration: As explained in Chapter 2, the addressing mode of local SRAM banks inside compute clusters can be set to either *interleaved* or *blocked* mode. By default, the KALRAY MPPA®-256 uses the interleaved mode to evenly spread the traffic across all local memory banks. Yet, it appears that such a behaviour makes a lot more complex implementation of memory partitioning. Indeed, doing so requires to *stride* executables in 64 bytes long pieces of code, to use `goto` instructions to enforce jumps from strides to strides, to build a correct memory mapping where pieces of code are aligned with memory addresses matching a single bank and to use complex MMU configurations to avoid accesses to inappropriate memory addresses. To avoid this complexity, we will assume in the rest of the dissertation that the memory addressing scheme inside compute clusters has been configured to the *blocked* mode to guarantee that contiguous memory addresses represent contiguous physical memory areas.

NoC configuration: As explained in Chapter 2, the DNoC interface of each cluster is equipped with hardware packet shapers in order to limit traffic injection in the NoC. Previous contributions were made to use those packet-shapers to enforce (ρ, σ) limitation when applying Network Calculus to the KALRAY MPPA®-256 [46, 47]. In our case, conflicts on the NoC are avoided altogether thanks to rule 2 of the execution model that imposes a global NoC TDM schedule. Thus, limitation through packet shapers is not required. For this reason, we will assume in the rest of the dissertation that such mechanisms are disabled by the hypervisor at boot time.

External DDR-SDRAM: For the accesses to DDR-SDRAM, we make two assumptions. Firstly, similarly to the local SRAM addressing mode, the external DDR-SDRAM is addressed by default using an interleaved scheme. Again, this is done to improve the average memory throughput by evenly distributing the traffic to all DDR-SDRAM banks. Again, striding problems occur when one tries to partition the memory. To account for this memory addressing scheme, DMA micro-codes must jump from one memory stride to the other to limit their traffic to a single bank. To avoid complex DMA micro-codes, we will assume in the rest of the dissertation that the interleaved DDR-SDRAM addressing scheme is disabled. Secondly, as explained in Chapter 2, the DDR-SDRAM arbiter of the KALRAY MPPA®-256 uses a complex two-step arbitration policy. We assume the MPFE to be configured with starvation counters equals to 0 for all masters as in Section 5.1.3 in order to simplify the arbitration policy to simple Round-robin. In addition, we assume the Reorder core to be disabled and to follow a simple FIFO policy. Doing so greatly simplifies the modeling of the arbitration scheme. One may note that thanks to the Rule 4 of the execution model, the situations where several masters compete to access different memory areas of the same bank are avoided by design. Consequently, it can be argued that enabling or disabling reordering should have

very limited impact on performances in practice. Indeed, the effect of reordering should be small when bank-competition is avoided. The main reason for making this assumption is not performance but rather to make hardware modeling simpler.

7.1.1.2 Global time-synchronization

On the Bostan version of KALRAY MPPA®-256, the timers of the DSUs are fully synchronous by design. Thanks to this, synchronizing all the instances of the hypervisors running on all clusters can be done easily. Once all compute clusters are started, an I/O cluster can broadcast a message containing a timestamp value referring to a future date. Then, all hypervisors just need to wait for this date to be reached by their local timestamps to start running. When achieved correctly, this enables all hypervisors to be fully synchronized. In the rest of the dissertation, we will assume that the activation of hypervisors is fully synchronous on all RMs.

7.1.2 Enforcing the execution model

In this section, we will detail the methods applied to enforce the 4 rules of our execution model and how temporal isolation can be provided to partitions at run-time.

7.1.2.1 Rule 1

As explained in Chapter 5, the rule 1 of the execution model states that strict spatial partitioning must be enforced between partitions inside compute clusters. To do so, the following techniques are applied.

MMU configurations: Each PE has an MMU enabling both virtualisation and memory protection. By setting the MMUs' configurations correctly, one can split the memory in bank-aligned memory chunks, thus avoiding bank sharing. In addition, the virtual addressing enables the code of applications to be independent from their actual position in physical memory. To do so, one LTLB entry can be used to forbid access to the 2MiB range of physical addresses and the remaining LTLB entries are used to set up the virtual address space of the partition from 2MiB upwards. Doing so makes feasible the allocation of any set of contiguous banks using only the 8 fully associative LTLB entries of PEs.

Example 25 (Memory partitioning using MMUs)**.** *Figure 7.1 shows an example of two PEs assigned to two different partitions. Their address space is split in two parts. The range of physical addresses is forbidden. Each PE can access only virtual addresses from* `0x20000` *upwards. The virtual address spaces are mapped on different memory banks to enforce partitioning.*

Execution modes of cores: The *k1b* cores support two execution modes: *privilege* and *user*. The PEs execute the code of applications only when running in user mode. Changing from user to privilege mode through system calls or other exceptions is forbidden to avoid MMU reconfigurations or access to cluster peripherals (DMA, ...). Only the RM running the hypervisor stays in privilege mode to manage the different resources of the cluster.

Exception handlers: The exception handlers of PEs are defined by the hypervisor at boot time. In the current implementation, all PEs share common exception handlers simply turning off the calling PE. Although this may seem very restrictive, one may note that an exception occurring due to a TLB miss can be identified as a segmentation fault in our frozen-MMU configuration

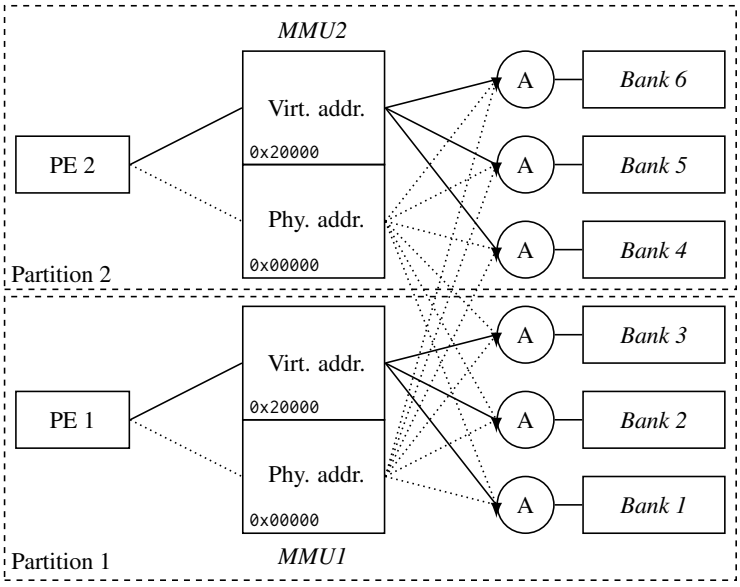


Figure 7.1: Example of local SRAM partitioning using virtual memory protection

strategy. Any TLB miss should be considered as an erroneous behaviour for which turning off the callee can be appropriate. Moreover, handling system call exceptions should never be necessary. Communications with out-of-cluster elements are achieved by DMA transactions initiated by the hypervisor. In such a passive communication policy, applications should not require to treat I/O actively. So, system calls should, in this configuration, be considered as erroneous behaviours. Finally, exceptions because of hardware traps also involve an incorrect execution pattern (division by zero, ...) for which shutting down the PE responsible for the problem is appropriate. Introducing softer exception management schemes may be necessary for certain kind of applications. We consider their implementation as a perspective of amelioration of our work.

Management of stacks: On the KALRAY MPPA®-256, the location of the PEs' stacks is fixed at boot time. Since the boot procedure of PEs is part of the hypervisor, and since the whole memory mapping is known a priori, stacks of PEs are assigned correct locations into their partitions' address spaces.

Using this four techniques, applications running on PEs can address only memory locations within their partitions' boundaries. Moreover, no erroneous behaviours (either because of bugs or malicious software) should impact co-running partitions since executions in privilege mode using exceptions are strongly terminated. Thus, the principle of spatial partitioning is strongly enforced at run-time within compute clusters. Moreover, in our current implementation, the hypervisor's code and data are placed in a reserved local memory bank to avoid any interferences with applications. This ensures that interferences between applications cannot propagate through the hypervisor.

7.1.2.2 Rule 2

The rule 2 of the execution model states that NoC communications must respect a global TDM schedule. To do so, our hypervisor manages NoC *scheduling tables* for each outgoing PCs of its cluster.

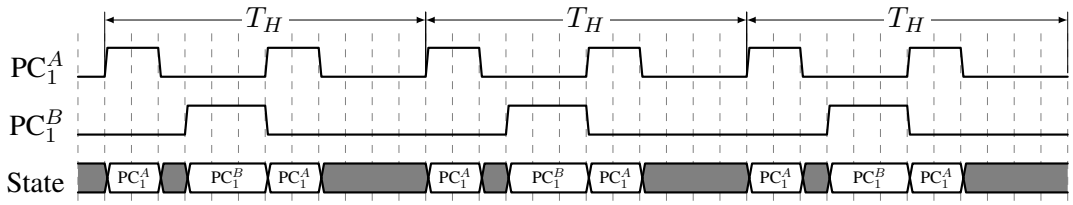


Figure 7.2: Example of application of a NoC scheduling table

Structure of scheduling tables: Assuming a set of outgoing PCs $[PC_1, \ldots, PC_n]$, the hypervisor will repeat a scheduling table with a hyperperiod T_H of length:

$$T_H = \underset{i=1}{\overset{n}{lcm}}(T_i)$$

where $lcm(a, b)$ is the least common multiple of a and b. The schedule is flattened in memory as an array representing the succession of PCs and idle states with their associated duration over a complete hyperperiod. At each activation, the hypervisor checks if it enters a new PC state. If it does, it starts the DMA to achieve the according NoC transfers.

Example 26 (NoC scheduling table)**.** *The Table 7.1 represents a scheduling table for 2 PCs: PC_1^A and PC_1^B with their associated durations. PC_1^A and PC_1^B respectively have a duration of* 2 *and* 3 *and a period of* 6 *and* 12*. Thus, as shown in Figure 7.2, the NoC scheduling table has an hyperperiod $T_H = lcm(6, 12) = 12$.*

Table 7.1: Example of scheduling table of PC_1^A and PC_1^B

Occupant (of the NoC interface)	PC_1^A	*Idle*	PC_1^B	PC_1^A	*Idle*
Duration (in nb. of hypervisor act.)	2	1	3	2	4

The choice of storing the PCs configurations using flat scheduling tables rather than just their timing parameters helps in decreasing the work load of the hypervisor. Indeed, at each activation, the hypervisor simply looks up in the table for the current DMA configuration and applies it. By doing so, we trade memory overhead for time complexity. As we will explain in Section 7.3.2, narrowing the WCET of the hypervisor is of major importance for performance reasons, thus motivating this choice.

DMA micro-code: As explained in Chapter 2, the DMAs of the KALRAY MPPA®-256 can execute custom micro-code. In our case, the micro-code is considered as a part of the hypervisor and is not configurable by applications. Our DMA micro-code is able of sending a queue of buffers autonomously. It means that, once started by the hypervisor at the beginning of a PC state, the DMA can send multiple non-necessarily contiguous memory areas from its cluster to one remote cluster. The time complexity required for configuring the DMA micro-code to send n non-contiguous memory chunks grows in $\mathcal{O}(n)$. In order to mask this time, we use two copies of the same DMA micro-code. At run-time, when the DMA is processing one queue of buffers using one of the micro-code copies, the hypervisor can configure the second copy of the micro-code in prevision of the next PC to be started. Thus, when this new PC begins, the hypervisor can quickly start its associated transfers by changing the address of the micro-code executed by the DMA and start it. Once this is done, one of micro-code copies is made available to be configured in prevision of the next PC.

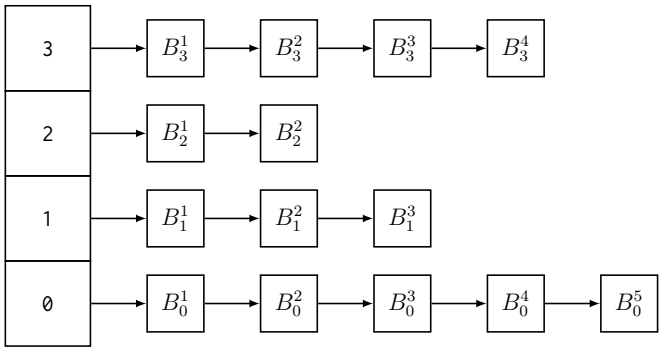


Figure 7.3: Example of list of buffer queues associated to a PC

Off-line verification: In our approach, verifying that the amount of data during one PC slot does not run over the following PC slot is made with off-line verifications only. Indeed, because of the rule 3 of the execution model, the queues of buffers that must be sent at each PC slot are defined off-line. Moreover, thanks to the rule 2, we are able to assume that no collisions ever occur on the NoC. So, using the models of the NoC introduced in Chapter 5, it is possible to bound the time required to send each queue of buffers. With these time bounds, it becomes possible to check that all PC slots are long enough for all their associated buffers to be sent without risking overruns. Then, the only work left to our hypervisor is to start DMAs at the right time to ensure respect of the schedule.

Overall, flattening the scheduling tables, using two DMA micro-codes and checking PC slots' loads off-line enable to enforce the respect of the global TDM schedule at run-time while keeping the work-load of the hypervisor relatively low, or at least masked in order to keep hypervisor activations intervals short.

7.1.2.3 Rule 3

As explained in Chapter 5, the rule 3 of the execution model imposes to define off-line which chunks of memory need to be sent during each PC slot. Since our DMA micro-code is capable of sending multiple non-contiguous buffers autonomously, we associate to each PC slot a queue of buffers to be sent.

Buffer queues: Each PC is associated with a circular list of buffer queues. Before the beginning of a PC slot, all the memory buffers to be sent during this slot are appended to the DMA's queue in order to be sent as soon as the PC slot starts.

Example 27 (List of buffer queues)**.** *Figure 7.3 depicts an example of list of queues of buffers that could be associated to a PC. During the first activation of this PC, buffers B_0^1 to B_0^5 should be sent by the DMA. During the second activation, buffers B_1^1 to B_1^3 are sent. Similarly, buffers B_2^1 to B_2^2 and B_3^1 to B_3^4 are sent respectively during the third and fourth activation of the PC. Then, during the fifth activation, buffers B_0^1 to B_0^5 are sent again and so on.*

Management of distributed memory: Each of the buffers depicted in Figure 7.3 consists of three elements: a local relative offset, a total size and a remote relative offset. Here, an offset represents a memory address that is relative to the base address of the considered PN. In both

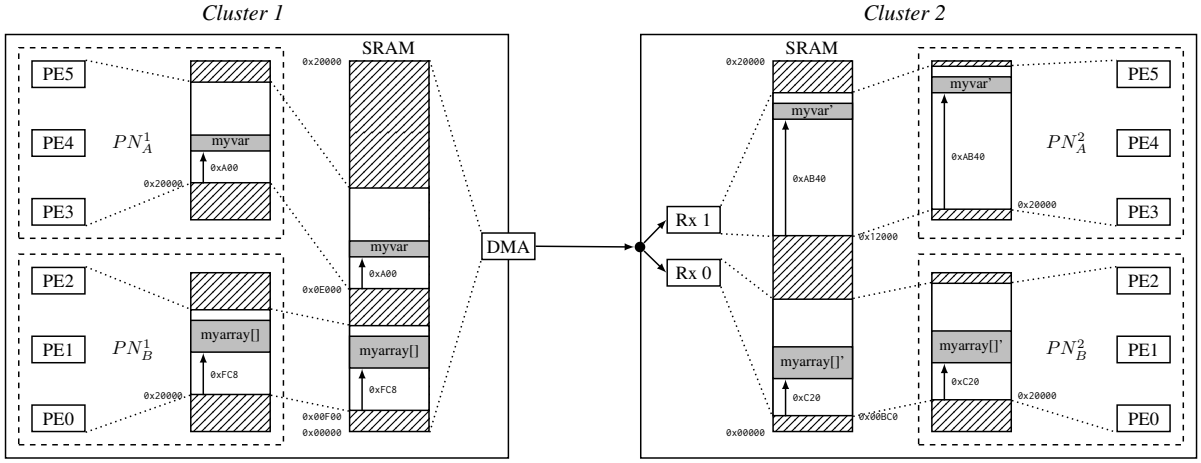


Figure 7.4: Example of relative offsets for managing distributed memory

virtual and physical addressing, the relative offset of each symbol of the PN is the same. When the hypervisor configures the DMA to send a buffer, it converts the local memory offset of the buffer to be sent into a physical address using the base physical address of the PN. Then, the NoC packet header is configured to send the data to a specific reception channel at the remote offset. Assuming that the remote hypervisor configured this reception channel to the physical address space of the remote PN, then, the data are written at the specified remote offset in the distant partition.

Example 28 (Relative offsets for distributed memory)**.** *Figure 7.4 shows an example of two partitions A and B, each of which has two PNs (PN_A^1 and PN_A^2 for partition A and PN_B^1 and PN_B^2 for partition B). The two partitions are distributed over two clusters. The buffer denoted by the symbol* myvar *in the virtual address space of PN_A^1 needs to be sent over to PN_A^2 into the buffer* myvar'*. As shown in Figure 7.4, the local offset of* myvar *is* `0xA00` *both in virtual and physical addressing. Thus, the DMA can be configured to read from the physical address* `0x0EA00` *and to send packets at destination of the reception channel* Rx 1 *of the cluster 2 with a remote offset* `0xAB40`*. Assuming that the reception channel* Rx 1 *has been configured by the remote hypervisor to be aligned with the base physical address of PN_A^2 (*`0x12000` *in the example), then* myvar *will be written into the address space of PN_A^2 at the specified offset. A similar process is used to send* myarray[] *from PN_B^1 to* myarray[]' *in PN_B^2.*

From a practical point of view, managing memory offsets by hand is usually not a simple task. To avoid such complexity, offset are usually handled by application designer with symbols such as "myvar" and "myvar'" in the previous example. Indeed, specifying the name of a variable to copy from one PN to another variable in another PN is usually simpler than specifying their memory addresses by hand. Although not a problem for local symbol, having reference to remote symbols is not that simple on the KALRAY MPPA®-256 since the code for each cluster is compiled into an independent binary. Thus, at compile time, the exact memory location of remote symbols is not known. To tackle this issue, during compilation, all references to remote symbols are temporarily replaced by arbitrary values to make compilation feasible. Once all the binary executables of all clusters are compiled, it is then possible to get the symbol table from each cluster's binary and to patch the executables

so that arbitrary remote offsets are replaced by the actual post-compilation and post-linking memory addresses of the remote symbols.

Overall, these two methods enable to manage queues of buffers for all PCs activations. Making this possible is important since it enables to verify off-line that PCs will always be sufficiently long to send all their pending data.

7.1.2.4 Rule 4

As explained in Chapter 5, the rule 4 of the execution model imposes that no external DDR-SDRAM banks can be accessed simultaneously by two partitions. Since PNs can access the external RAM only by using PCs, the management of I/O related PCs must be done carefully.

Assumptions on scheduling tables: When computing the NoC scheduling table, a specific constraint needs to be met in order to enforce the respect of rule 4. Since no PCs initiated by two partitions can overlap if accessing the same memory bank, the choice of the temporal offsets for these PCs is done while propagating the overlapping avoidance constraint from NoC resources up to DDR-SDRAM banks. If the resulting scheduling table meets this constraint, no additional support is required at run-time to meet rule 4.

Bounds on durations of memory transactions: The implementation of the rule 2 involves verifying that the amount of data sent during each PC does not run over its duration. Applying such off-line verification involves not only to model the NoC using equations of Section 5.1.2.2 but also models of the DDR-SDRAM arbiter for PC targeting the external RAM. As mentioned in Section 7.1.1.1, we assume that the MPFE treats requests in a pure Round-Robin fashion and that the reordering pool is disabled. Consequently, since bank contentions are avoided by construction, bounding the duration of DDR-SDRAM transactions is largely simplified and largely tightened because the cost of activating a row with $t_{RP} + t_{RCD}$ is payed only once for each PC slot (added to the costs of refreshes that must be payed anyway). The cost of switching from RD to WR still needs to be accounted since rule 4 does not avoid RD and WR overlapping. The rationale behind this choice is that we eliminate the most expensive operations (namely several request targeting different rows of the same bank) while not being too restrictive in the possibility to allocate co-running I/O PCs. Yet, if one considers the cost of RD and WR switching as too expensive, the rule 4 can simply be augmented by an additional constraint on the direction of co-running PCs. Although making the computation of PC offset more complicated, it may lead to better memory throughput under favorable configurations.

7.2 Experimental validation

The enforcement of the execution model requires not only support from the hypervisor at run-time, but also off-line pre-computation to build scheduling tables and to verify PCs with respect to hardware models. In order to verify the soundness of the approach, we evaluated it experimentally on the KALRAY MPPA®-256.

7.2.1 Experimental setup

The goal of our experimental evaluation is to confirm the capability of our execution model, and more precisely its implementation through the hypervisor and the off-line computation, to tem-

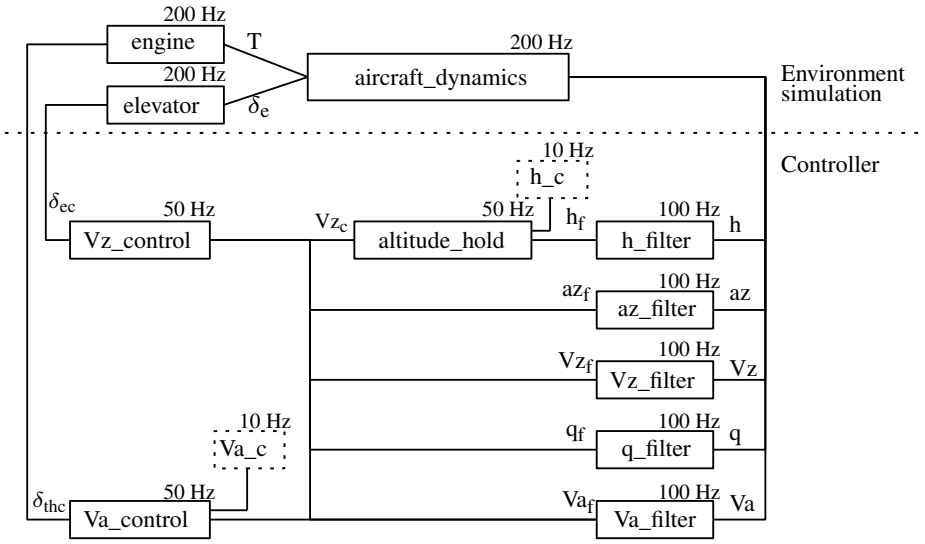


Figure 7.5: Longitudinal flight controller of the ROSACE case study

porally isolate several co-running partitions. To do so, we used the ROSACE controller [153] as a reference application, and we analyzed its behaviour while running in isolation and with co-running partitions.

7.2.1.1 The ROSACE case study

Pagetti *et al.* presented the ROSACE case study in [153]. Although of modest size, the longitudinal flight controller of ROSACE is representative of real avionics applications since it involves complex multi-periodic execution patterns.

Description of the application: The flight controller of the ROSACE case study is shown in Figure 7.5. It consists of two major parts: 1) the *simulation* environment which represents the physical model of the aircraft under consideration; and 2) the *Controller* which includes three sub-controllers and five filters. All blocks of the simulation part runs at 200Hz while the filters and controllers are respectively executed at 100Hz and 50Hz.

Implementation choices: Since relatively small, both the ROSACE controller and the simulation environment fit inside only one compute cluster. So, we placed the whole ROSACE case study into one PN occupying 3 local SRAM banks and 5 PEs. We used directly the C code generated by the PRELUDE compiler as in [153] to generate 5 co-running threads. Figure 7.6 shows the schedule of the five threads for 4 activations of the ROSACE tasks at 200Hz. In order to study configurations with multiple PNs, we augmented the ROSACE application with a *Set Point Generator* (or *SPG*) which provides commands (h_c and V_{a_c}) on-line. The SPG simulates commands from the pilot and enables to run complex scenarios of descents and ascents. We placed the SPG into a second PN and interconnected the ROSACE and the SPG PNs with two PCs with respective frequencies 10Hz and 200Hz. In addition, we connected the SPG's PN to an ION using a 200Hz PC. It is used to log the output value from ROSACE in the external RAM for post-processing. The budget of this partition with the 2 PNs and the 3 PCs is shown in Figure 7.7.

The activation of the tasks of ROSACE as shown in Figure 7.6 are aligned on the end of PC_2. One master thread (τ_1) wait for the end-of-PC notification from the RM and starts running.

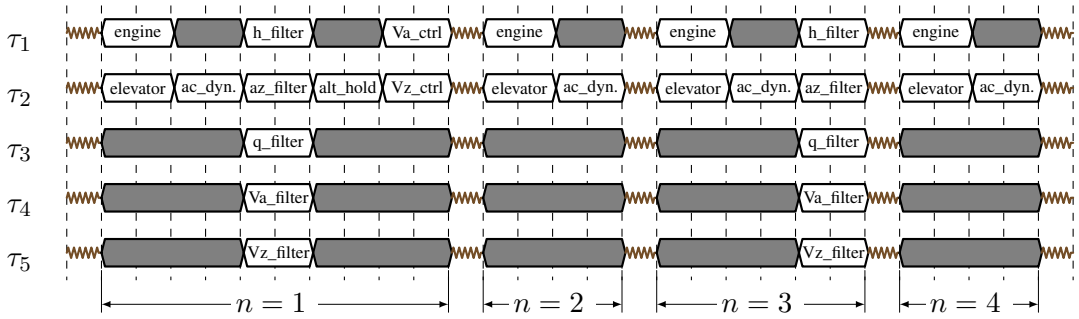


Figure 7.6: Schedule of ROSACE blocks in 5 threads

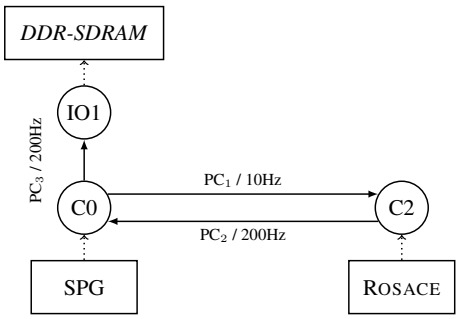


Figure 7.7: Budget of the ROSACE's partition with 2 PNs and 3 PCs

Then, the 5 PEs are synchronized using simple busy-waiting locks implemented using shared (local) memory. The PC_1 is activated at the same time as 1 out of 20 activations of PC_2.

The implementation of ROSACE involves intra-cluster communications (between the 5 threads), inter-cluster communications (between ROSACE and the SPG) and memory accesses to log the outputs values in external RAM. In doing so, we defined a reference implementation that will be analyzed in isolation and against concurrent partitions.

7.2.1.2 The ImgInv co-running application

In order to evaluate the sensibility of ROSACE to co-running partitions, we implemented a second application, called *ImgInv* which computes the *inverse* of a grayscale image as shown in Figure 7.8. The choice of ImgInv to run our experiments if essentially motivated by its configurability which makes the evaluation of many partition's configurations simple. Firstly, the number of PEs on which the application is parallelized can be configured easily. The inversion algorithm is composed of independent loops which can be spread among several processing elements without heavy modifications in the algorithm's structure. Moreover, the speedup that can be expected from this parallelization in practice can be close to the number of PEs running the code if the memory is adequately managed to avoid bottlenecking. Secondly, the memory footprint of ImgInv can be controlled efficiently. The code size of ImgInv is relatively small (few KiB), thus making the image storage prominent. Thus, the local memory required by ImgInv to run can be adjusted by simply changing the size of images. Thirdly, the NoC budgets of ImgInv can also be controlled using different picture size and various periods. And finally, the implementation if ImgInv is rather simple.

Architecture of the application: ImgInv runs cyclically. It has three image buffers of equal size. At one execution cycle, the PEs execute the inversion algorithm on one buffer containing an



(a) Before inversion (b) After inversion

Figure 7.8: Example of application of ImgInv

image. Simultaneously, one of two other buffers receives (from the NoC) a new image to be inverted. Meanwhile, the third buffer which contains an already inverted image is being sent by the DMA over the NoC. At the following execution cycle, the PEs will execute the inversion algorithm on the previously received image. The image that has just been inverted will be sent. And a new image will be received in the third buffer.

Implementation choices: Our implementation of ImgInv is designed to completely fit inside one PN. It runs on square 8-bits grayscale images. The size of images (512x512, 256x256 or 128x128) must be configured off-line in order to define the length of buffers (and thus the budget of the PN) prior to execution. The emission and reception of buffers is achieved through two PCs, one for emissions and the other for receptions. The durations of the PCs are correlated to buffers' lengths. Several instances of ImgInv can be placed in a single partition to exchange data. For example, the emission PC of one ImgInv PN can be used as reception PC for another ImgInv instance.

In general, we will use ImgInv with different configurations (different image sizes, one or multiple instances per partition, ...) in a partition beside ROSACE. We expect from these experiments to exhibit a property of temporal isolation, meaning that ROSACE's behaviour will stay insensitive to the configurations of ImgInv.

7.2.1.3 Observation means

The validation of the expected temporal isolation property requires to be able to finely observe the execution of ROSACE. To do so, we used three different means of observation which are detailed below.

Post-processing of output values: In our implementation of ROSACE, the output values from the controller (h, a_z, V_z, q, V_a) are logged in the external RAM by the SPG. The development environment of the KALRAY MPPA®-256 enables to dump the content of memories after executions. So, after running ROSACE on a specific scenario, the execution logs can be read and post-processed to be verified. Not spotting execution errors on such logs does not prove that the execution was correct, however, it can be argued that not spotting errors is encouraging. Thus, it may increase our confidence in the correctness of the implementation. On the other hand, spotting errors proves the non-correctness of the execution and may provide valuable

informations for investigation. In our experiments, this post-processing of the logged value was used to verify the functional correctness of our ROSACE implementation against the proven-correct values given in [153].

CNoC-based log server: In our implementation of the execution model, the CNoC is left unused. Since it enables to send small asynchronous messages, we used it to implement a simple log feature. We reserved one compute cluster to run a *Log server*. The Log server continuously waits for the reception of CNoC messages and stores them in its local memory on reception. Then, a PC is reserved for the Log server to periodically flush its received messages into external RAM. Overall, the Log server enables applications to log information at run-time asynchronously without requiring a PC. Although providing flexibility, this log mechanism requires careful considerations to be used adequately. The CNoC reception buffers are not designed to store multiple messages. Thus, if a CNoC message is received while a previous message is still pending, the newly received one may be dropped. So, logs from the Log sever should always be considered together with the number of lost messages (that can be counted on the KALRAY MPPA®-256) to verify if they can be trusted as is or not. And secondly, sending CNoC messages takes time. Using the log server is thus intrusive and modifies the temporal behaviour of applications under consideration. This requires particular attention in a time-triggered environment.

DDR-SDRAM interposer: As a third observation mean, we used a DDR-SDRAM protocol analyzer (also called a DDR-SRAM *interposer*) to increase our confidence in the soundness of the implementation. Such hardware tool is physically placed between the memory module and the motherboard. It is capable of sniffing commands going from the DDR-SDRAM controller to the memory modules in a *non-intrusive* manner. It means that DDR-SDRAM commands can be logged at run-time, and this is done transparently from software. The captured data contains many information on the commands (RD, WR, ACT, PRE, REF, ...) with the addresses targeted and also cycle accurate timestamps. Depending on the interposer, the value of data (during RD and WR) may or may not be logged as well. Anyway, if the memory spaces reserved to partitions do not overlap, only the address targeted by a memory request is required to identify the originating partition. Doing so enables to check that memory banks are accessed by partitions only when they are supposed to do so and that the rule 4 of the execution model is respected by checking that memory transactions of partitions sharing banks do not overlap in time.

Overall, these means of observation when considered separately may not be totally sufficient to prove the correctness of an implementation. Yet, we argue that using all three simultaneously, although not providing a formal proof of correctness, definitely increases our confidence in the validity of the approach. In the rest of the dissertation, we will consider this level of confidence as sufficient for validation purpose.

7.2.2 Scenario 1: no interference

The first experiment involves running the ROSACE partition alone on the target. The implementation follows exactly what is shown in Figure 7.7. The ROSACE's PN is placed on cluster 2 and the SPG's PN on cluster 0. The ION is reserved on the north I/O cluster. In this setup, we validated the functional behaviour of ROSACE using the output values logged in the external RAM. Moreover, we used the Log server to save the execution times of all blocs of ROSACE. Figure 7.9 shows these

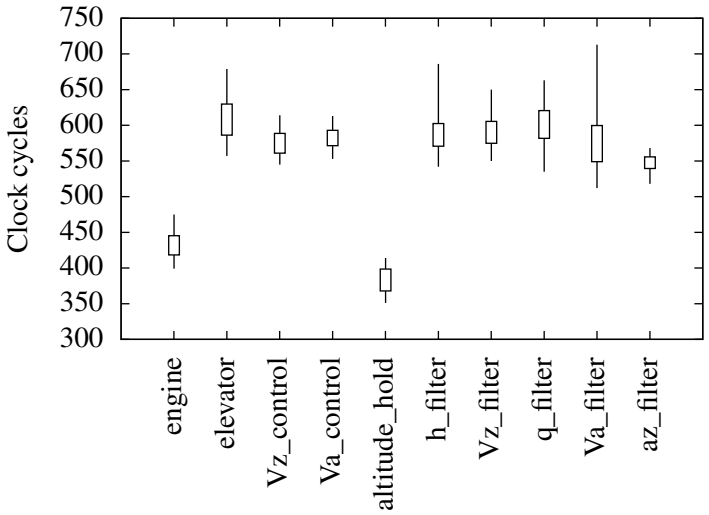


Figure 7.9: Measures of the execution times of ROSACE

execution times[1]. This candlestick chart represents the minimum and maximum values using lines and the standard deviations (centered on the mean) using boxes. The execution of each bloc is repeated at least 10,000 times.

This setup serves as a reference for other scenarios. The goal of other experiments will be to demonstrate that, with or without co-running partitions, the execution times of ROSACE always remain the same.

7.2.3 Scenario 2: NoC interferences

The second experiment involves two instances of ImgInv in a partition that runs beside ROSACE. As shown in Figure 7.10, each instance of ImgInv is placed alone in a PN. The two PNs are placed on clusters 3 and 4. The two instances are similar. Each uses 8 cores and 7 local memory banks to store the code and data (512x512 images in this case). The two ImgInv instances are linked by two PCs running at 500Hz. The reception PC of one instance corresponds to the emission PC of the other and vice versa. The purpose of each ImgInv is to re-invert the image that has just been inverted by the other instance. Although this has not real functional interest, it enables us to generate NoC traffic on resources that are also used by the PCs between ROSACE and the SPG which remain in similar configuration as in scenario 1.

Under this configuration, we observe that the execution times of ROSACE are very close to those observed in isolation. Indeed, the best and worst observed execution times of all basic blocs of ROSACE are *exactly* the same with or without the ImgInv co-runners. Moreover, the average execution times of each bloc are very close in both cases (they differ by less than 1%). In addition, we temporarily modified all hypervisors to log the time at which they receive DNoC packets. By doing so, we were able to observe that 100% of the DNoC packets were received within their intended time slot. This means that no packet has ever been late because of delays incurred by the

[1]The execution times of the "aircraft_dynamics" bloc have an average of approximately 59,000 cycles. We do not plot them for clarity.

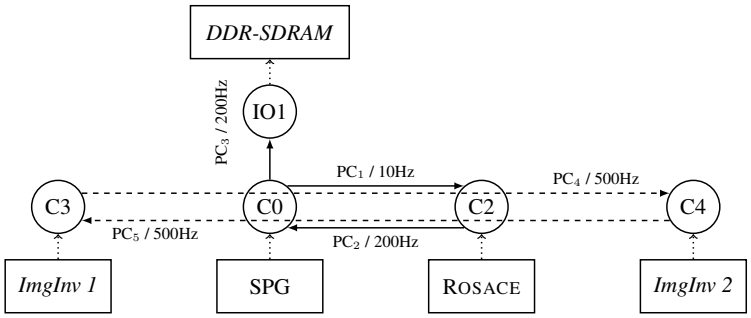


Figure 7.10: Configuration for the scenario 2 with two instances of ImgInv sharing NoC resources with ROSACE

communications of other partitions thus showing the respect of the overlapping avoidance on the NoC.

7.2.4 Scenario 3: DDR-SDRAM interferences

During the third experiment, we introduced competition for accesses to the external DDR-SDRAM. As shown in Figure 7.11, only one instance of ImgInv is used. The PN configuration of this instance is similar to the one of the scenario 2 (8 PEs and 7 banks). Two PCs are used to send and receive images to/from the external DDR-SDRAM. We arbitrarily placed the buffers accessed by the ImgInv partition in the same DDR-SDRAM bank as ROSACE in order to generate interferences. As for scenario 2, the execution times of ROSACE remain very similar to those in Figure 7.9. Again the best and worst observed execution times are the same no matter the behaviour of co-running partitions and the average of execution times also differ by less than 1%. We re-used the Log sever to save the reception dates of DNoC messages and again, 100% of those packets were received within their intended slots.

Overall, these experiments evaluate the capability of our approach to enforce temporal isolation between partitions at run-time. We observed that the execution times within a partition seem to remain insensitive to the pressure put on resources that are massively shared on many-core architectures such as the NoC or the DDR-SDRAM. This insensitivity is precisely what we expected when trying to enforce temporal isolation between partitions.

Although this experimental evaluation does not provide a formal proof of correctness, it has the benefit of showing the complexity of the development effort required to master complex architectures such has the KALRAY MPPA®-256. This involves the management of boot codes for all cores, the configuration of many hardware resources such as the local and external memories, the synchronization of all clusters to obtain a global notion of time, the management of exceptions handlers, the configuration of MMUs, the programming of DMAs, the management of large scheduling tables, the on-line and off-line operations to take distributed memory into account and to verify budgets before execution (among other problems). In addition, making the system observable also requires efforts to post-process values written in DDR-SDRAM, to implement log features using the CNoC or to analyze the traces of DDR-SDRAM accesses. Yet, despite all this complexity, we have shown by tackling these problems in practice that the management of resources at a very low level is feasible and that the temporal behaviour of the final system seems to yield very promising timing properties regarding temporal isolation.

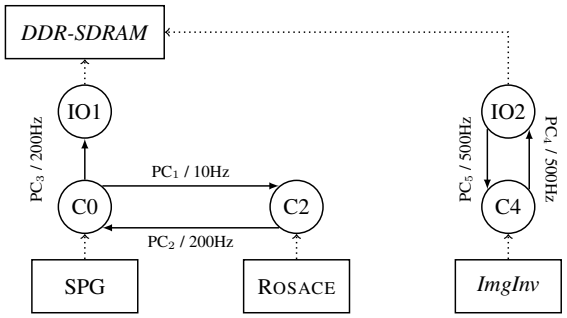


Figure 7.11: Configuration for the scenario 3 with one instance of ImgInv sharing a DDR-SDRAM bank with ROSACE

7.3 Discussion on design trade-offs

In the previous sections, we detailed the architecture of the hypervisor, how the rules of the execution model are enforced at run-time and 3 experimental case-studies exhibiting the expected property of temporal isolation. Yet, we did not mention the various design trade-offs that need to be accounted when developing a hypervisor and the impact on applications' performances. In this section, we analyze these trade-offs with a specific focus on the footprint of scheduling tables in the local memories of clusters and on the various elements having an impact on the WCET of the hypervisor.

7.3.1 Memory footprint

As explained previously, one of the 16 local SRAM banks is reserved for the hypervisor in each cluster. In this bank, both the code and data used by the hypervisor must stored. The code includes the boot codes (1 for the RM and a generic one for all PEs), the exception handlers (few bytes only), the RM's code that is running cyclically and the DMA's micro-codes. Overall, the total size of the code can be kept within few dozens of KiB. Thus, all the remaining space of the SRAM bank (approximately 100KiB) can be used to store the data. Among all data used by the hypervisor, the scheduling tables and the buffer queues appear to be the most memory-consuming.

7.3.1.1 Footprint of scheduling tables

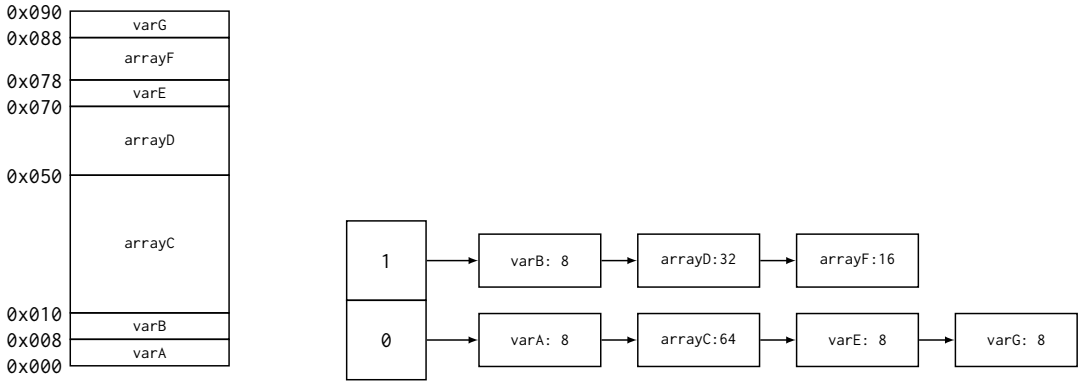
As explained in Section 7.1.2.2, our approach relies on flattened scheduling tables storing all the successive communication states in a hyperperiod. With n the number of PCs for emission of message, the hyperperiod T_H is defined as:

$$T_H = \underset{i=1}{\overset{n}{lcm}}(T_i)$$

we can deduce the total number of PC activation in one hyperperiod as:

$$N_t = \sum_{i=1}^{n} \frac{T_H}{T_i}$$

In the general case, the number of elements in the array N_a representing this scheduling table is lower bounded by N_t. In the worst case, there may be idle states between each PC activation. In this case, the total number of elements to store in the array of the scheduling table is $N_t \times 2$. Thus,



(a) Memory mapping (b) Buffer queues

Figure 7.12: Example of buffer queues for non-optimized memory mapping

$N_t \leq N_a \leq 2N_t$. Clearly, the ratio of the periods can have a major impact on N_a [154, 155]. Let us show that with an example.

Example 29 (Size of scheduling tables)**.** *Let us consider two PCs with respective periods* $T_1 = 12000$ *and* $T_2 = 24001$*. The hyperperiod of this configuration will be* $T_H = 288012000$*. In this case, the number of PC activations is* $N_t = T_H/T_1 + T_H/T_2 = 36001$*. So, in the worst case, the number of elements in the scheduling table array may be* $N_a \approx 72000$*. Let us now modify to* $T_2 = 24000$*. In this case, the hyperperiod of the system is* $T_H = 24000$ *and* $N_t = 3$*. The array will now contain at most* $N_a = 6$ *elements.*

The previous example shows that PC configurations that look very similar may have very different *efficiency* regarding memory utilization. Let us define an indicator to measure the efficiency of a configuration with:

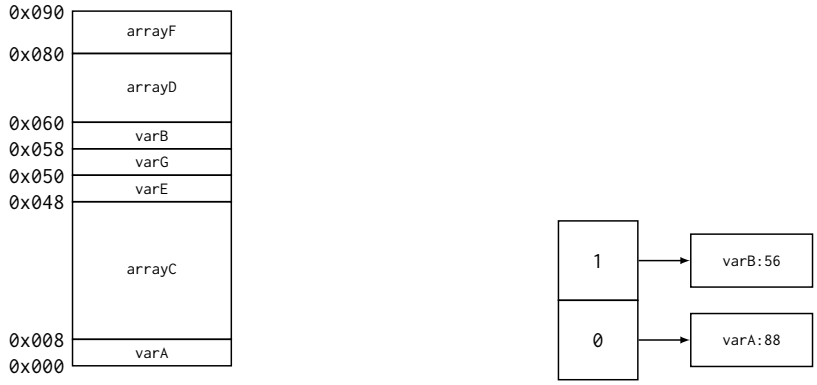
$$M = \frac{n}{N_a}$$

In the best case, $M = 1$ and so the number elements in the array is exactly equal to the number of PCs. In this case, all the periods of PCs have to be equal and the DMA utilization $U = \sum C_i/T_i = 1$. In all other cases, $M < 1$. When the number of elements in the array is largely superior to the number of PCs, $M \approx 0$ denotes an inefficient utilization of the memory.

Example 30 (Efficiency of scheduling tables storage)**.** *Let us again consider two PCs with periods* $T_1 = 12000$ *and* $T_2 = 24001$*. In the worst case,* $N_a = 72002$*. This configuration leads to a very inefficient use of memory with* $M = 2/72002 \approx 2.8 \times 10^{-5}$*. Let us now modify* $T_2 = 24000$*. In this case, the memory utilization* $M = 2/6 \approx 0.33$ *is largely better in worst case.*

7.3.1.2 Impact of non-contiguous memory areas

As explained in Section 7.1.2.3, the data to be sent at each PC activation must be defined off-line as a list of buffer queues. The hypervisor browses the list circularly by moving from a buffer queue to the next at each activation of the corresponding PC. These queues must be defined off-line and all their elements represent non-contiguous chunks of memory. Depending on the memory mapping, the length of the queues can vary widely.



(a) Memory mapping (b) Buffer queues

Figure 7.13: Example of buffer queues for optimized memory mapping

Example 31 (Example of buffer queue in function of memory mapping)**.** *Let us assume 7 variables and arrays that need to be sent from a compute cluster:* `varA` *on 8 bytes,* `varB` *on 8 bytes,* `arrayC` *on 64bytes,* `arrayD` *on 32 bytes,* `varE` *on 8 bytes,* `arrayF` *on 16 bytes and* `varG` *on 8 bytes. For functional reasons, on one out of two activations of an outgoing PC, the following list of variables needs to be sent :* `varA`*,* `arrayC`*,* `varE` *and* `varG`*. During the other PC activations, the 3 other elements are sent. Assuming the memory mapping shown in Figure 7.12a, the list of buffer queues that need to be stored is shown in Figure 7.12b. Since the memory mapping is not optimized for making buffers as large as possible, the memory overhead for storing the list of buffer queues is large. The same example with an optimized memory mapping is shown in Figure 7.13.*

Placing data contiguously helps make the storage of buffer queues more efficient. However, one may also note that sending several data in a bigger chunk of memory requires the receiver to now about it in order to de-concatenate the data correctly.

7.3.1.3 Good practices

In order to avoid the inefficient utilizations of memory as detailed in previous sections, one may restrict the application's to respect some rules to avoid problematic corner cases.

Choice of periods: The choice of the PCs' periods has a major impact on the size of the array storing the scheduling table. In particular, mastering the periods ratio seems to be a key element to avoid very inefficient uses of memory ($M \approx 0$). When two PCs have prime periods, the hyperperiod can become very large and lead to very small M. To avoid this problem, one may restrict the choice of PCs period to ensure good divisibility. To do so, the choice of periods may be limited only to *harmonic* periods.

Definition 9 (Harmonic periods)**.** *Let us consider a set of* n *periods* $\{T_1, ..., T_n\}$ *sorted in increasing order, meaning that* $\forall i \in [1, n-1],\ T_i \leq T_{i+1}$*. The periods are defined as* harmonic *if and only if:*

$$\forall i \in [1, n-1],\ \exists k \in \mathbb{N}^*,\ T_{i+1} = k \times T_i$$

Hopefully, making such an assumption is usually not a problem in industrial multi-periodic real-time systems where the periods of tasks often respect this constraint by design.

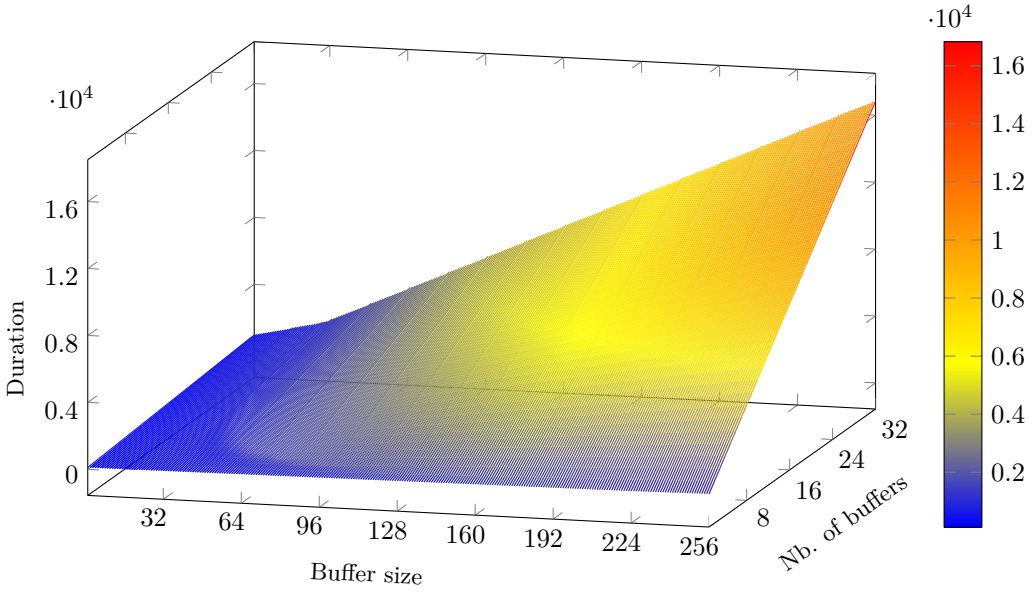


Figure 7.14: Execution times of the DMA micro-code in function of the buffers queues.

Size of data chunks: The mapping of the buffers to be sent during PCs has an impact on the memory overhead required to store the queues of buffers treated by the hypervisor. Clearly, concatenating several data before sending them helps compress this list of buffer queues. However, one may note that being efficient in the storage of the list of buffer queues actually requires to have a small number of large data chunks to be sent. Yet, this can be done not only by optimizing the application's memory mappings to concatenate its data but also by designing the application so that there are not many small data but few larger data instead. Consequently, it can be argued that taking this into account at design time (of applications) may be of great help to keep the hypervisor's memory overhead low. To do so, one may decide to impose to application's designer to keep the number of exchanged data under a pre-defined trigger.

Figure 7.14 shows the results of benchmarks during which we measured the time required by the DMA engine to transfer a queue of non-contiguous buffers. For each experiment, all buffers are assumed to be of the same length. The experiments were repeated while varying the sizes of buffers between 1 and 256 double-words. The number of buffers in the queue also varies between 1 and 32. In all cases, we measure the time required for the DMA to process the whole queue of transfers autonomously. We can observe that the number of buffers to be sent by the DMA seems to have more impact on the total transfer duration than the sizes of the buffers, thus confirming experimentally the soundness of the statement given above.

7.3.2 WCET of the hypervisor

The WCET of the hypervisor is of major importance for performance reasons. Indeed, as explained previously, the duration and the periods of PCs are expressed in number of hypervisor activations. So, keeping the WCET of the hypervisor low enables to activate it at a fast pace and thus to avoid coarse overestimations of PCs durations. In the context of this thesis, our WCET estimates are simply deduced from measures of execution times for simplification. In the context of a real aerospace design, the WCET of the hypervisor should be computed using safer estimation techniques. We

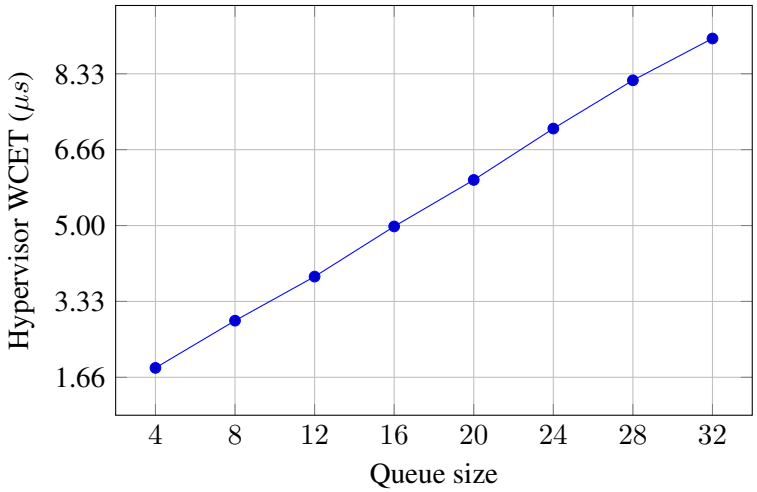


Figure 7.15: Measured WCET of the hypervisor versus the size of the DMA buffer queue

argue that computing WCET using static analysis should be both easy and efficient given the hardware configuration. Indeed, the hypervisor is the only task executed by the RM which accesses only its reserved SRAM bank. Doing so enables to reduces the WCET estimation problem to a time-compositional VLIW mono-core processor accessing a SRAM only shared with a DMA (when reading its micro-code). Computing the WCET in this situation should be very favorable.

As explained previously, configuring a DMA for sending n buffers autonomously takes $\mathcal{O}(n)$ time. So, there is a clear correlation between the WCET of the hypervisor and the number of buffers in a DMA queue. Since the size of this queue can be defined by software, it is an important design trade-off to choose an appropriate value. A small number of elements in this queue will narrow the WCET of the hypervisor while limiting its autonomous capabilities. On the other hand, managing a large queue will be time-consuming but let the DMA run on its own for a long time. The best choice depends on applications and deciding the optimal value before trying it experimentally does not seem to be an easy task. In our current implementation, we arbitrarily chose to support short WCETs and thus short buffer queues. The rationale behind this choice is that NoC latency seems to be very important to efficiently parallelize application on several clusters. Being able to manage short PCs is very likely to help in keeping latencies low.

Figure 7.15 shows the results of experimental benchmarks measuring the WCET of the hypervisor for different configurations of DMA the buffer queue. In our current implementation, we chose the queue size so that the hypervisor's WCET remains under $5\mu s$. In the experiments of Section 8.3, we will evaluate the impact of this choice on the performances that applications can expect.

7.4 Summary

In this chapter, we detailed the architecture of the hypervisor with a specific focus on the methods used to enforce the respect of the execution model's rules. We have overviewed the hardware configurations that can be used to mitigate or even eliminate intra-cluster and inter-cluster interferences between partitions. Then, we evaluated the approach over an avionics case study and an image processing application. Based on that, we showed that the expected temporal isolation is obtained in practice. Finally, we discussed the major design trade-offs for the hypervisor and user

applications including the size and the number of data in applications and the configurations of the DMA regarding the size of its buffer queue.

Overall, we have shown that enforcing temporal isolation between partitions using low level management of the KALRAY MPPA®-256 is complex but practically feasible. In this chapter, we assumed the mapping of applications to be already defined in order to focus on the elimination of interferences. Yet, computing such mapping can be challenging given the complexity of the hardware and the size of industrial applications. In the next chapter, we will provide details on our approach to tackle this issue using constraint-programming.

Chapter 8

Validation of the budgets

Contents

8.1 Budget choice 121
8.1.1 Assumptions 122
8.1.2 Minimum constraints 123
8.2 Budget validation 124
8.2.1 Modelling framework 124
8.2.2 Problem formulation 127
8.2.3 Constraints 128
8.3 Experimental results 133
8.3.1 Goals 133
8.3.2 Overview of the case study 134
8.3.3 Analysis of results 135
8.4 Summary 141

In this chapter, we detail how the *Validate* phase of our integration work-flow (Figure 6.1 page 96) is implemented using *constraint programming*. The purpose of the whole *Validate* phase is to check whether an application fits inside the resource budget attached to its partition or not. To do so, we propose to compute a mapping of the application on the resources described in the budget while meeting functional and non-functional constraints. Throughout the chapter, we will assume only applications following the model of Section 2.1 and budgets formalized using the notations introduced in Section 6.1.2.

At first, we will analyze how resource budgets should be chosen and what are our assumptions regarding them. Then, we will describe how the mapping problem can be modeled using constraint programming and the notion of *Conditional Time-Intervals*. Finally, we will experimentally evaluate the scalability of the approach using a real-world case-study before discussing the various performance trade-offs that can be identified from the results. In order to ease reading, Table 8.1 reminds the notations used in the chapter that were defined throughout the dissertation.

8.1 Budget choice

In our approach, budgets of partitions are assumed to be given a priori. The choice of a specific budget over another has consequences on the implementation and performances of applications.

SYMBOL	MEANING
	Application model
τ_i	The i-th task of the application
T_i	The period of task τ_i
S_i	The set of sub-tasks composing task τ_i
τ_i^j	The j-th sub-task of task τ_i
C_i^j	The WCET of sub-task τ_i^j
M_i^j	The total memory footprint of sub-task τ_i^j (including code, static and read-only data)
P_i	The set of precedence relation between the sub-tasks of task τ_i
δ_k	The k-th data of the application
m_k	The size of δ_k
$prod(\delta_k)$	The sub-task producing δ_k
$cons(\delta_k)$	The set of sub-tasks consuming δ_k
	Hardware parameters
S_{bank}^{SRAM}	The size of a local SRAM bank inside a compute cluster (128KiB)
Δ_S	Latency of a NoC switch
Δ_L	Latency of a NoC link
	Budget
$\mathcal{N}$	The set of PNs in the budget of the partition under consideration
$N_c(pn)$	The number of PEs reserved for PN pn
$N_b(pn)$	The number of local SRAM banks reserved for PN pn
$\mathcal{C}$	The set of PCs in the budget of the partition under consideration
$src(pc)$	The source PN or ION of the PC pc
$dst(pc)$	The sink PN or ION of the PC pc
$T(pc)$	The period of the PC pc
$C(pc)$	The duration of the PC pc
$T(pc)$	The offset of the PC pc
$R(pc)$	The maximum number of NoC nodes on the path taken by PC pc

Table 8.1: Reminder on notations defined throughout the dissertation

For example, when an application does not fit in the local memory of a single compute cluster by a small overhead, it is not simple to decide whether its budget should include one PN that takes a complete cluster and a second PN that takes just a small part of another cluster, or 2 PNs so that each takes a little more than half a cluster.

In this section, we will detail our assumptions on budgets and discuss what are their consequences on the implementation of applications. Then, we will provide guidelines to choose budgets and maximize their chances of being valid.

8.1.1 Assumptions

In our current approach, we make several assumptions on the budgets that are allocated to partitions. We detail these assumptions and their motivations below.

8.1.1.1 No code fetch

We assume that applications do not fetch code from the external DDR-SDRAM. Therefore, an application needs to completely fit inside the local memories of compute clusters to be executed. When one does not fit in a single cluster, it should be split over several ones and use PCs to exchange data between the PNs.

Clearly, making such an assumption is restrictive. Our decision to do so is motivated by two reasons.

1. the problem of loading code from the external RAM has already been studied in the literature. For example, the scheduling problem solved by Becker *et al.* [146] includes not only the allocation of *Runnables* to PEs but also the scheduling of the reads and writes to external RAM. Here, we address a different problem in order to provide a complementary contribution to the state of the art.

2. avoiding the code fetch from external RAM forces large applications to be split over several compute clusters. By doing so, the problem of tightly managing the NoC when applications are widely distributed cannot be avoided. Since the main difference between many-core and multi-core processors lies in the distributed architecture of the former, addressing in priority all the NoC-related issues appears as especially relevant given that this thesis focuses on many-core-specific problems only. Counter-intuitively, our main goal is not really to demonstrate how the power of many-core processors can be used to speed-up applications, it is rather to figure out the solutions that will help us parallelizing future applications on several clusters when there will be no other choice to meet performance requirements.

8.1.1.2 Hypervisor

As explained in Chapter 7, the implementation of the hypervisor imposes constraints on how memory can be used in compute cluster and on how the period and the duration of PCs can be expressed. In each compute cluster, one local SRAM bank is reserved for the hypervisor. Consequently, PNs cannot require more than $N_b(pn) = 15$ memory banks. Moreover, since PCs periods and length must be multiple of the hypervisor's period, we will assume that all PCs-related are measured directly using this multiple number. If the hypervisor's period is $5\mu s$ and the period of a PC should be $20\mu s$, then $C(pc) = 4$ for example.

8.1.2 Minimum constraints

In our approach, the budget of a partition is assumed to be given a priori. Yet, defining such a budget appears to be difficult and sometimes a counter-intuitive task. In order to help the application designer in the process of choosing a budget, we provide some necessary conditions for a budget to be valid. Although meeting all those conditions does not necessarily involve that a budget is valid, it can improve our confidence in the probability of it to be so.

8.1.2.1 Local memory constraints

Since we assume that no code is ever fetched from the external RAM, the whole application must fit within the local SRAM banks of its PNs. Thus, we can deduce from the application's memory footprint a necessary condition on the overall budget to verify that it at least includes sufficient storage space. Although this condition is not sufficient, it is necessary and any valid budget must verify it. With $\mathcal{N}_i$ the set of PNs in the budget, S the set of sub-tasks composing the application,

S_{bank}^{SRAM} the size of a local SRAM bank (table 5.1 page 89) and M_i^j the memory footprint of a sub-task τ_i^j (Section 2.1), condition 8.1 must be met.

$$\sum_{pn \in \mathcal{N}_i} N_b(pn) \times S_{bank}^{SRAM} \geq \sum_{\tau_i^j \in S} M_i^j \tag{8.1}$$

In our current implementation, one of the 16 local memory banks is reserved for the hypervisor. Thus a budget always verifies $N_b(pn) \leq 15$ for all its PNs. Consequently, the absolute minimal number of PNs required for a budget to be valid must verify condition 8.2

$$|\mathcal{N}_i| \geq \left\lceil \frac{\sum\limits_{\tau_i^j \in S} M_i^j}{15 \times S_{bank}^{SRAM}} \right\rceil \tag{8.2}$$

8.1.2.2 Computational power constraints

As for the memory, a necessary condition concerns the computational power. With C_i^j the WCET of a sub-task τ_i^j and T_i the period of a task τ_i, the utilization ratio U of the application is defined as:

$$U = \sum_{\tau_i^j \in S} \frac{C_i^j}{T_i}$$

Since the number of cores on PEs on which the application can be scheduled must be greater than U, condition 8.3 must be met.

$$\sum_{pn \in \mathcal{N}_i} N_c(pn) \geq \lceil U \rceil \tag{8.3}$$

On the KALRAY MPPA®-256, since there are 16 PEs per compute cluster, $N_c(pn) \leq 16$ for all PNs. So, the absolute minimal number of PNs required for a budget to be valid must verify condition 8.4.

$$|\mathcal{N}_i| \geq \left\lceil \frac{U}{16} \right\rceil \tag{8.4}$$

8.2 Budget validation

Given a budget, the purpose of the *Validation* phase is to check whether an application can be scheduled while meeting all its functional and non-functional constraints. We propose to do so using an approach based on *constraint programming* (or *CP*) in order to compute the schedule of the application off-line as stated by the Constraint 3. Using CP enables to take into account complex models coming from the specificities of the many-core target and the limitations imposed by the execution model using a formal approach.

8.2.1 Modelling framework

Several approaches in the literature have been proposed to map dependent tasksets on multi- and many-core processors using ILP [146, 156, 148]. Unfortunately, most of these approaches often face major scalability issues when applied on industrial-sized applications. Typically, heuristics

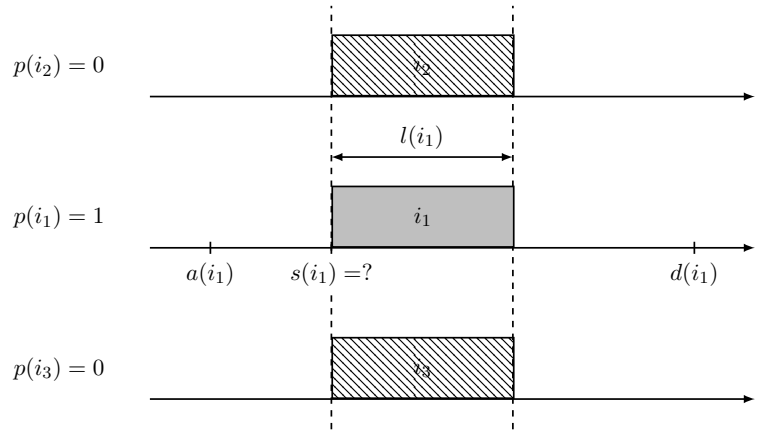


Figure 8.1: Example of three conditional time-intervals i_1, i_2 and i_3 with only one present in the final solution

are proposed in order to tackle these issues while providing sub-optimal results. In this thesis, we propose to use a CP-based approach using a different modelling framework that has proven its usefulness in practice in order to cope with the same scalability issues that we faced when developing the preliminary ILP-based version of our case study.

In the rest of the chapter, we will use the notion of *Conditional Time-Intervals* [157, 158] introduced into IBM ILOG CP Optimizer [159] since version 2.0. For commodity, we provide an introduction to this scheduling framework in the following sections.

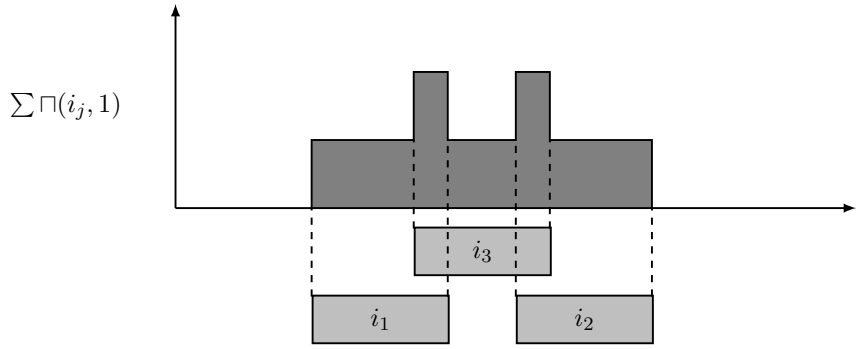
8.2.1.1 Conditional time-intervals

An *Interval* variable i represents an activity of finite duration that should be scheduled by the solver. An interval i features several attributes such as its start date $s(i)$ or its duration $l(i)$. Typically, a scheduling problem will require to find start dates of interval given their durations and other additional constraints. Each interval i is defined within a specific time window $[a(i), d(i)]$ where $a(i)$ and $d(i)$ respectively represent the activation date and the deadline of the interval. It means that any schedule computed by the solver must verify $s(i) \geq a(i)$ and $s(i) + l(i) \leq d(i)$.

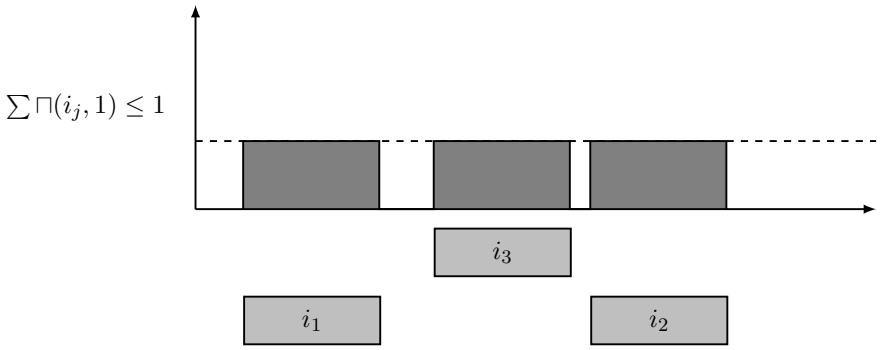
An interval is said to be *conditional* (or *optional*) if it also features a *presence* attribute $p(i)$. Such intervals do not necessarily require to be present in the final solution. $p(i)$ is assigned 0 or 1 to encode this. Basically, all constraints on intervals that are eventually not present in the solution are ignored.

Overall, such a formulation enables a natural modelling of problems where activities are not only scheduled over time but are also allocated to resources. To do so, each activity can be represented by a set of conditional intervals (one interval for each resource) and constrained so that only one of these intervals is present in the final solution as shown in Figure 8.1.

Example 32 (Conditional time-intervals)**.** *Let us assume a simple multi-core scheduling problem. Given a set of task to be scheduled on m identical cores, a typical formulation of the problem using intervals would include m conditional intervals for each task. Each set of intervals would be constrained so that only one is present in the final solution, thus encoding the task-to-core mapping problem. Other constraints should be applied on the start dates of intervals to ensure a correct schedule regarding temporal constraints.*



(a) Cumulative functions on i_1, i_2 and i_3 without constraints



(b) Cumulative functions on i_1, i_2 and i_3 with constraints

Figure 8.2: Example of three intervals associated with pulse cumulative functions

8.2.1.2 Cumulative functions

A *cumulative function* represents the usage of a resource by activities over time as the sum of their individual contributions. They can be used in order to constraint the utilization of a resource to remain in a specific envelope representing the resource's capabilities. While other types of cumulative functions exist (see [157, 158] for further details), we will use only *pulse* functions throughout this chapter. A pulse function $\sqcap(i, h)$ increments the usage of a resource by h at the beginning of interval i and decrements it by h when it completes as shown in Figure 8.2a.

8.2.1.3 Usual constraints on intervals

Interval variables can usually be constrained using a variety of different constraints. Among all of those implemented within IBM ILOG CP Optimizer we use only the following subset:

Presence constraints: optional intervals have an attribute $p(i) \in [0, 1]$ which denotes their presence ($p(i) = 1$) or their absence ($p(i) = 0$) in the final solution. Constraints on $p(i)$ can be used to enforce the presence or absence of specific intervals depending on configurations. Especially, *alternative* constraints enable to force only one interval to be present over a set of intervals. For commodity, we will use the $\oplus$ symbol to denote alternative constraints defined as follows:

$$\oplus(I) = 1 \Leftrightarrow \sum_{i \in I} p(i) = 1$$

with $I = \{i_1, \ldots, i_n\}$ a set of optional intervals;

Precedence constraints: intervals can be constrained by precedence relations in order to enforce a specific order for their activation. We note the precedence constraint operator $\rightarrow$ with the following semantics:

$$i_1 \rightarrow i_2 \Leftrightarrow s(i_1) + l(i_1) \leq s(i_2)$$

Cumulative function constraints: cumulative functions can be constrained in order to maintain the usage of a resource within a specific envelope. Figure 8.2b shows an example of three intervals associated with pulse cumulative functions under a constraint. Here, using pulse functions of height 1 with a constraint on the cumulative function to be less than 1 actually implements a non-overlapping constraint.

8.2.2 Problem formulation

We consider an application $\mathcal{A} =< \tau, \delta >$ following the model of Section 2.1 and a budget $< \mathcal{N}, \mathcal{I}, \mathcal{C}, \mathcal{B} >$ as defined in Section 6.1.2.

8.2.2.1 Job-level solving

Since intervals apply on jobs and since our task model allows for different periods in the task set, we will unfold the scheduling over a complete hyperperiod of length:

$$T_H = \underset{\tau_i \in \tau}{\text{lcm}}(T_i)$$

We will refer to the k-th activation of task τ_i as the k-th *job* of τ_i denoted $\tau_{i,k}$. Similarly, we will refer to the k-th activation of the sub-task τ_i^j as the k-th *sub-job* of τ_i^j denoted $\tau_{i,k}^j$. We denote the set of all sub-jobs in a hyperperiod S.

Since in our model, precedence relations are only applied to sub-tasks of the same parent task, we duplicate them to operate at the sub-job level. Moreover, given the fact that data always have only one producing sub-task, we assume that each data is over-written every time its producing sub-task is activated. The k-th production of δ_x by sub-job $\tau_{i,k}^j$ is denoted $\delta_{x,k}$.

8.2.2.2 Management of I/O requests

In our application model, sub-tasks are assumed to deal with I/O explicitly. Requests to read and write to I/O subsystems are modeled as input and output data in the sub-task. Consequently, I/O requests are not managed differently from data, the only exception being that data are placed on PN to PN PCs while I/O requests are assigned to PN to ION (or ION to PN) PCs.

In the rest of the chapter, and for readability, we will detail how scheduling data on the PCs is done but we will not explicitly account for the I/O requests. Yet, the management of I/O requests is strictly equivalent to data except that the set of eligible PCs to map them is not the same.

8.2.2.3 Decision variables

In our scheduling problem, we have two types of decision variables.

Sub-jobs on PNs: Each sub-job $\tau_{i,k}^j$ is associated with $|\mathcal{N}|$ optional intervals. We note $j(\tau_{i,k}^j, pn_l)$ the interval variable representing the allocation of sub-job $\tau_{i,k}^j$ on the PN pn_l. If $j(\tau_{i,k}^j, pn_l)$ is present in the final solution, it means that $\tau_{i,k}^j$ is executed by PN pn_l. The length of each sub-job interval $l(j(\tau_{i,k}^j, pn_l))$ is equal to the WCET C_i^j of its corresponding sub-task. The

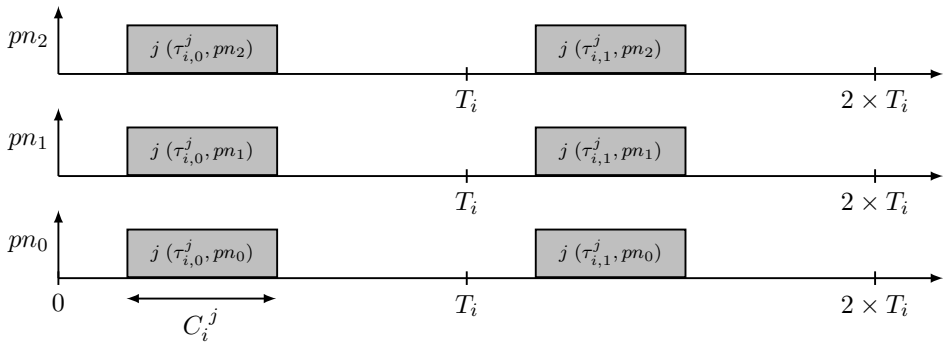


Figure 8.3: 6 interval variables representing 1 sub-task τ_i^j with 2 sub-jobs that may be scheduled on 3 PNs

activation date and deadline of all intervals are defined by the time window $[k \times T_i; (k+1) \times T_i]$ during which its corresponding sub-job lives. This is represented in Figure 8.3.

Data on PCs: Each data instance $\delta_{x,k}$ is associated with $|\mathcal{C}|$ optional interval variables. We note $d(\delta_{x,k}, pc_l)$ the interval representing the allocation of the k-th instance of δ_x on PC pc_l. If $d(\delta_{x,k}, pc_l)$ is present in the final solution, it means that $\delta_{x,k}$ is sent from the PN $src(pc_l)$ to the PN $dst(pc_l)$ via pc_l. Counter-intuitively, we do not set the length of these intervals to be the time for the data to be sent over the NoC. The length of a data interval $l(d(\delta_{x,k}, pc_l))$ is set equal to the length $C(pc_l)$ of one slot of the PC on which it is sent. The start date $s(d(\delta_{x,k}, pc_l))$ of each data interval should be aligned with a slot of its corresponding PC pc_l in order to encode the assignment of the data transfer during this specific slot. The activation date and deadline of a data interval are equal to the living time window $[k \times T_i; (k+1) \times T_i]$ of its producing sub-job.

8.2.3 Constraints

8.2.3.1 Sub-jobs mapping constraints

Each sub-job must be executed on only one PN. We enforce this behaviour using constraint 8.5.

$$\forall \tau^j_{i,k} \in S, \oplus(\{j(\tau^j_{i,k}, pn) \mid \forall pn \in \mathcal{N}\}) = 1 \tag{8.5}$$

If different sub-jobs of the same sub-task could execute on different PNs, the code and data of the sub-task would need to be duplicated on several PNs, thus putting a higher pressure on a potentially sensitive resource: the local SRAM. Moreover, for coherency reason, NoC communication would be required to transfer static data of the sub-task from one cluster to another, again putting pressure on a potentially sensitive resource: the NoC. To avoid these problems, we eliminate sub-task migrations over a hyperperiod by enforcing all sub-jobs of a sub-task to be assigned to the same PN using constraint 8.6.

$$\forall \tau^j_{i,k} \in S,\ \forall pn \in \mathcal{N},\ p(j(\tau^j_{i,k}, pn)) = p(j(\tau^j_{i,(k+1)\%\frac{T_H}{T_i}}, pn)) \tag{8.6}$$

Figure 8.4 shows an example of sub-jobs non-migrating from one cluster to another with intervals on other PNs are absent.

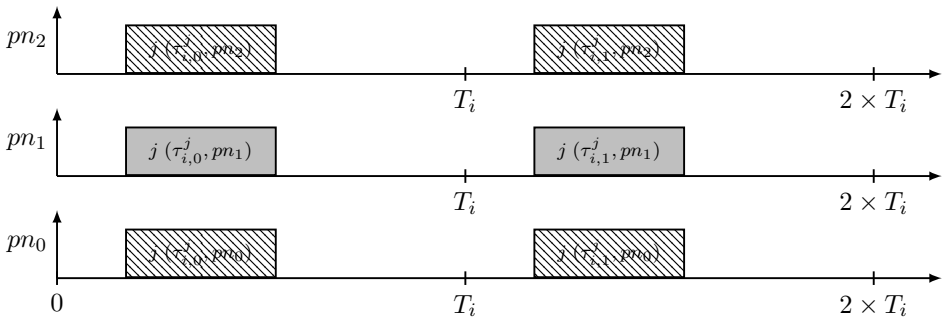


Figure 8.4: Example of sub-job interval presence following the constraint on migration of sub-jobs between PNs

8.2.3.2 PN utilization constraints

Since the number of PEs used by one PN is obviously limited, the number of sub-jobs simultaneously running inside a PN should be limited as well. We enforce that using cumulative functions as in constraint 8.7.

$$\forall pn \in \mathcal{N}, \sum_{\forall \tau^j_{i,k} \in S} \sqcap\,(j(\tau^j_{i,k}, pn), 1) \leq N_c(pn) \tag{8.7}$$

Constraint 8.7 imposes that, at any time, the number of sub-jobs running concurrently inside one PN never exceeds the number of PEs of this PN. We argue that such a condition is necessary and sufficient to ensure that all sub-jobs can always be executed by a PE without requiring preemption.

Proof. The problem of assigning to each sub-job one PE among $N_c(pn)$ is equivalent to a $N_c(pn)$-coloring problem of the associated *Interval Graph*. Here, an Interval Graph is a non-weighted non-directed graph having sub-jobs for nodes and edges linking all pairs of sub-jobs that overlap in time. Such an Interval Graph is well known to be *chordal* [160] and it inherits from all the properties of *perfect* graphs. The chromatic number of a perfect graph is equal to the size of its largest clique [161]. The largest clique of our Interval Graph equals the maximum number of intervals that can overlap at any point in time, which is $N_c(pn)$ by design. Consequently, our Interval Graph is $N_c(pn)$-colourable, meaning that all sub-jobs can be assigned to $N_c(pn)$ PEs without overlapping. □

In addition, the sub-tasks allocated to PNs should meet the memory specifications of their PNs. This is enforced using constraint 8.8. One may note that, in constraint 8.8, only the memory footprint of first sub-job of each sub-task is accounted. Since migration of sub-jobs is forbidden by constraint 8.6, all sub-jobs of a sub-task will use the same copy of the code and data.

$$\forall pn \in \mathcal{N}, \sum_{\tau^j_i \in S} p(j(\tau^j_{i,0}, pn)) \times M^j_i \leq N_b(pn) \times S^{SRAM}_{bank} \tag{8.8}$$

8.2.3.3 Data mapping constraints

Mapping data on PCs depends on the location of the producing sub-task and the consuming sub-task(s). If all are located within the same PN, no NoC transfer is required for the communication since sub-tasks will communicate via shared memory inside the compute cluster. As long as one

consuming sub-task is assigned to a different cluster from the producing sub-task, then the data must be sent over a PC connecting the two PNs. Constraint 8.9 imposes that for the first produced data interval.

$$\forall pc \in \mathcal{C},\ \forall \delta_x \in \delta,$$
$$\Big(\ p(j(prod(\delta_{x,0}), src(pc)))\ \wedge \sum_{\tau^j_{i,0} \in cons(\delta_{x,0})} p(j(\tau^j_{i,0}, dst(pc))) \geq 1\Big) = p(\delta_{x,0}, pc) \tag{8.9}$$

In addition, since sub-jobs do not migrate between clusters, every time a data is produced it should be sent over the same PCs. Consequently, we impose with constraint 8.10 that all the data interval must have a presence attribute equal to those of other data intervals representing the same data.

$$\forall pc \in \mathcal{C},\ \forall \delta_{x,k} \in \delta,\ p(d(\delta_{x,k}, pc)) = p(d(\delta_{x,k+1}, pc)) \tag{8.10}$$

As stated in Section 8.2.2.3, start dates of data intervals must be aligned with a slot of their corresponding PC. We enforce this using constraint 8.11.

$$\forall pc \in \mathcal{C},\ \forall \delta_{x,k} \in \delta,\ s(d(\delta_{x,k}, pc))\ \%\ T(pc) = O(pc) \tag{8.11}$$

8.2.3.4 PC utilization constraints

The amount of data that can be transferred during a PC slot must be limited to meet hardware constraints, and especially the NoC bandwidth and the DMA capabilities. Our execution model provides a property of collision avoidance on the NoC, thus drastically simplifying the estimation of NoC transfer durations. Indeed, using the equations of Section 5.1.2, we can derive the number of flits required to send each data δ_x of the application as $N^{total}_{flit}(m_x)$ where m_x is the size of δ_x. In addition, we can derive the number of flits that can safely be sent during one PC slot from equation 5.2 as:

$$N^{PC}_{flit}(pc) = C(pc) - (R(pc) + 1) \times \Delta_L - R(pc) \times \Delta_S$$

Thus, using cumulative functions, we can limit the number of data allocated to each PC slot so that the number of flits to be sent is less than the $N^{PC}_{flit}(pc)$. This is enforced using constraint 8.12 and illustrated in Figure 8.5.

$$\forall pc \in \mathcal{C},\ \sum_{\delta_{x,k} \in \delta} \sqcap\Big(d(\delta_{x,k}, pc), N^{total}_{flit}(m_x) + N^{gap}_{flit}\Big) \leq N^{PC}_{flit}(pc) \tag{8.12}$$

The N^{gap}_{flit} term of equation 8.12 represents the number of empty flits lost in the gap when the DMA jumps from one contiguous memory area to another. By accounting it for all data, we are making the conservative assumption that data are placed in non-contiguous memory areas, thus requiring DMA jumps between all of them. Clearly, computing the position of data in memory together with the NoC schedule could help decrease the overhead incurred by these jumps. Indeed, by concatenating data, larger chunks of memory can be sent efficiently over the NoC, as we recommended in Section 7.3.1.3. Unfortunately, doing so greatly increases the complexity of the problem which may thus face limitations regarding scalability. Extending our current model to include data positioning into the optimization problem in an efficient manner represents an interesting opportunity of future work.

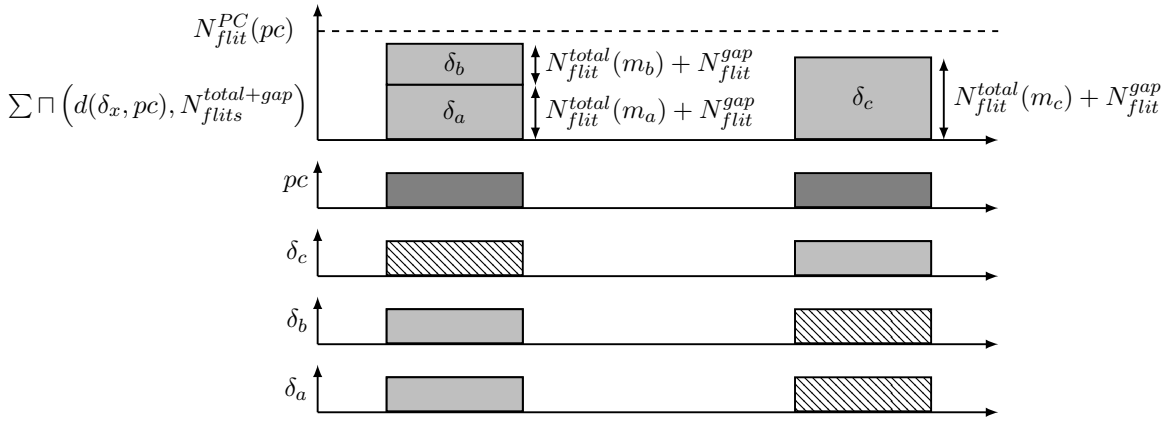


Figure 8.5: Limitation of the amount of data sent during the activation of a PC using cumulative functions on data intervals

Since the queues of DMAs can handle only a fixed number of non contiguous buffers, the number of data to be sent in each PC slot must be limited, especially because we assume non-contiguous data in memory. This is enforced using constraint 8.13.

$$\forall pc \in \mathcal{C}, \sum_{\delta_{x,k} \in \delta} \sqcap(d(\delta_{x,k}, pc), 1) \leq N_{bufs}^{DMA} \tag{8.13}$$

8.2.3.5 Precedence constraints

The application model imposes the respect of a set of precedence relations while executing sub-tasks having the same parent tasks. As explained in Section 2.1, only pairs of sub-tasks exchanging data can be constrained by precedence relations. Consequently, constrained data can be classified in two categories:

- *forward* data are produced by the parent node of the precedence relation and consumed by the child node. More formally, assuming a precedence (τ_i^j, τ_i^l) where τ_i^j must execute before τ_i^l, δ_x is said to be a forward data if $(\tau_i^j = prod(\delta_x)) \wedge (\tau_i^l \in cons(\delta_x))$;
- *backward* data are produced by the child node of the precedence relation and consumed by the parent node. In this case, the data produced by a sub-job of the child sub-task is consumed during the next sub-job of the parent sub-task. More formally, assuming a precedence (τ_i^j, τ_i^l) where τ_i^j must execute before τ_i^l, δ_x is said to be a backward data if $(\tau_i^j \in cons(\delta_x)) \wedge (\tau_i^l = prod(\delta_x))$.

With forward data, the freshest value semantics imposed by our application model requires that the child sub-task does not start before it receives the data produced by the parent sub-task. When both sub-tasks are assigned to the same PNs, communication is ensured using shared memory. Consequently, the data produced by the parent sub-task is committed to memory when it completes, and the child sub-task can start with no delay. This is enforced using constraint 8.14.

$$\forall(\tau_{i,k}^j, \tau_{i,k}^l) \in P_{i,k},\ \forall pn \in \mathcal{N},\ j(\tau_{i,k}^j, pn) \rightarrow j(\tau_{i,k}^l, pn) \tag{8.14}$$

However, if the two sub-tasks are assigned different PNs, then a NoC transfer is required for the produced data to be delivered to the child data. In order to enforce this precedence between the

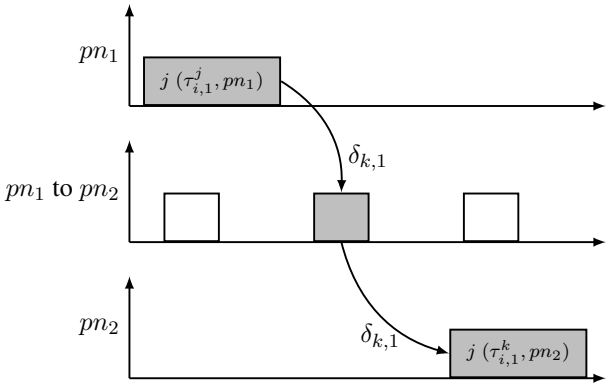


Figure 8.6: Data interval enforcing the precedence between sub-jobs

intervals of the two sub-jobs, the data interval during which the data is sent over the NoC is used as a pivot. As shown in Figure 8.6, two precedence constraints are used to ensure that the data is sent after the completion of the parent sub-task (constraint 8.15) and before the start of the child sub-task (constraint 8.16).

$$\forall pc \in \mathcal{C},\ \forall \delta_{x,k} \in \delta,\ j(prod(\delta_{x,k}), src(pc)) \rightarrow d(\delta_{x,k}, pc) \tag{8.15}$$

$$\begin{aligned} &\forall (\tau^j_{i,k}, \tau^l_{i,k}) \in P_{i,k},\ \forall pc \in \mathcal{C},\ \forall \delta_{x,k} \in \delta, \\ &\quad (\tau^j_{i,k} = prod(\delta_{x,k})) \wedge (\tau^l_{i,k} \in cons(\delta_{x,k})) \Rightarrow d(\delta_{x,k}, pc) \rightarrow j(\tau^l_{i,k}, dst(pc)) \end{aligned} \tag{8.16}$$

With backward data, the data consumed by the parent sub-job $\tau^j_{i,k}$ is the data produced by the previous child sub-job $\tau^l_{i,k-1}$. When the two sub-tasks are assigned the same PN, constraint 8.15 is sufficient to enforce the correct behaviour. Indeed, the data produced by $\tau^l_{i,k-1}$ is sent after the completion of the sub-job. Since its deadline equals the deadline of the producing sub-job, it needs to be sent before the next activation of the task and thus, necessarily before the activation of $\tau^j_{i,k}$. If the two sub-tasks are assigned different PNs, we need to ensure that the parent sub-job ends before the data produced by the child sub-job is sent to ensure that it never consumes a "too fresh" data. This is enforced using constraint 8.17.

$$\begin{aligned} &\forall (\tau^j_{i,k}, \tau^l_{i,k}) \in P_{i,k},\ \forall pc \in \mathcal{C},\ \forall \delta_{x,k} \in \delta, \\ &\quad (\tau^j_{i,k} \in cons(\delta_{x,k})) \wedge (\tau^l_{i,k} = prod(\delta_{x,k})) \Rightarrow j(\tau^j_{i,k}, dst(pc)) \rightarrow d(\delta_{x,k}, pc) \end{aligned} \tag{8.17}$$

8.2.3.6 Determinism constraints

In our application model, sub-tasks can exchange data, even without being constrained by precedence relations. So, if a producing and a consuming sub-job run concurrently, the ordering of the production and consumption of data may not be deterministic. Indeed, since the time at which the data is produced and the time at which it is consumed depend on the internal structure of sub-tasks and on their execution time, the ordering between production and consumption is not known a priori and may vary at run-time. In this thesis, we consider that the flexibility of the application model enables to choose any ordering between production and consumption of data as long as it stays consistent between execution. To do so, when the two sub-tasks are assigned different PNs,

we avoid the overlapping between the data interval during which the data is sent and the sub-job interval of the consumer using constraint 8.18.

$$\begin{aligned}&\forall \delta_{x,k} \in \delta,\ \forall \tau_{i,l}^{j} \in cons(\delta_{x,k}),\ \forall pc \in \mathcal{C},\\ &\quad (prod(\delta_{x,k}), \tau_{i,l}^{j}) \notin P_k \wedge (\tau_{i,l}^{j}, prod(\delta_{x,k})) \notin P_k\\ &\quad \Rightarrow \sqcap(d(\delta_{x,k}, pc), 1) + \sqcap(j(\tau_{i,l}^{j}, dst(pc)), 1) \leq 1\end{aligned} \tag{8.18}$$

Similarly, if the producing and consuming sub-tasks are assigned the same PN, they should not overlap. This is enforced using constraint 8.19.

$$\begin{aligned}&\forall \delta_{x,k} \in \delta,\ \forall \tau_{i,l}^{j} \in cons(\delta_{x,k}),\ \forall pn \in \mathcal{N},\\ &\quad (prod(\delta_{x,k}), \tau_{i,l}^{j}) \notin P_k \wedge (\tau_{i,l}^{j}, prod(\delta_{x,k})) \notin P_k\\ &\quad \Rightarrow \sqcap(j(prod(\delta_{x,k}), pn), 1) + \sqcap(j(\tau_{i,l}^{j}, pn), 1) \leq 1\end{aligned} \tag{8.19}$$

8.3 Experimental results

We argue that our CSP formulation of the *Validation* phase can be used for industrial-sized applications and achieve good performances with realistic case studies. To demonstrate that, we evaluated the approach over an industrial case-study provided by Airbus. In the next sections, we will firstly clarify what should be understood by "good performances" and what our assumptions are when running the experiments. And secondly, we will analyze several results from experimental studies and identify the key trade-offs regarding performance.

8.3.1 Goals

The goal of our experimental study is twofold. Firstly, we aim at showing that our formal approach is capable of managing large applications as required in an industrial context. Secondly, we aim at demonstrating that having an automated and efficient formulation of the problem enables design-space explorations regarding both budget choices and hypervisor implementations. In order to investigate all those topics, we will run multiple experiments while varying three different parameters of the case study.

8.3.1.1 Increasing work-load

In order to demonstrate the ability of our formulation to deal with large and heavy application, we will not only try to schedule our industrial case study but we will also modify it in order to artificially increase the work-load to be processed. By doing so, we show how our CSP formulation can manage not only existing applications but also more demanding applications as it may be necessary in the future.

In order to simulate work-load increase, we will modify the periods of tasks in the application model. We use a parameter, denoted k_{div} which grows with the resulting work-load. We will reduce the period of tasks as $\widetilde{T_i} = \frac{256-k_{div}}{256} \times T_i$ and find out if a schedule can still be found. With $0 \leq k_{div} \leq 255$, the modified periods can vary between their original value T_i and $T_i/256$. Since all other timing parameters of the application, including the WCET of sub-tasks, are left unmodified, reducing the period increases the utilization of tasks and simulates a more demanding application overall.

8.3.1.2 Narrowing the budget

Our formulation of the problem assumes a pre-defined budget in which the application must be scheduled. Since choosing such a budget does not appear to be an intuitive task, we will investigate how modifications on an original budget can impact the resulting performances. Firstly, we will investigate three different PNs configurations. With n_{pn}^{orig} the original number of PNs in the budget, we will investigate 3 budgets with $n_{pn}^{orig} \leq |\mathcal{N}| \leq n_{pn}^{orig} + 2$ in order to identify how increasing the number of PNs can impact our capability to find correct schedules. With a similar idea, we investigate how modifications on PCs can impact schedulability. With $C_{orig}(pc)$ the original length of a PC, we investigate three PCs configurations with durations within the $[C_{orig}(pc), C_{orig}(pc)+2]$ range for each of them.

8.3.1.3 Choosing the DMA configuration

We explained in Section 7.3.2 how the number of buffers treated by the DMA can impact the WCET of the hypervisor and thus the overall performance. Although this value should be fixed in a real avionics development, we use the opportunity of these experiments to also identify the trade-off regarding the number of DMA buffers that suits best the needs of our applications. To do so, we use different values of N_{bufs}^{DMA} in the $[4, 32]$ range to identify which configuration provides the schedulability up to the highest utilizations.

8.3.2 Overview of the case study

As a case study, we used a *large* industrial application from Airbus. This application fits in the model of Section 2.1. It features several tasks with harmonic periods. In a hyperperiod, the total number of sub-jobs and data instances is close to 100,000. When turned into optional intervals, the number of decision variables and constraints in the CSP grows to several millions.

In order to run this application, we make several assumptions on the geometry of its associated budget. These assumptions are detailed below.

8.3.2.1 One PE per PN

In our experiments, we always limited the number of cores of each PN to be $N_c(pn) = 1$. This choice is motivated by the two following reasons:

1. assuming only one PE inside each PN enables to eliminate the inter-PE interferences at the local SRAM level. Only interferences coming from the DMA still need to be taken into account. Consequently, WCET of sub-task can be computed accordingly in an efficient manner;

2. the goal of this thesis is to explore many-core-specific problems overall. By avoiding intra-cluster parallelization, we increase the need for an efficient inter-cluster parallelization involving a tight management of the NoC. Although our CSP formulation is capable of managing both intra- and inter-cluster parallelization, we will focus only on the latter in order to identify the problems and trade-offs that incur with disruptive hardware architectures using NoCs.

8.3.2.2 Prompt symmetric PCs

During our experiments, pairs of PNs in a budget are assumed to be linked using *prompt* and *symmetric* PCs. We clarify these notions below.

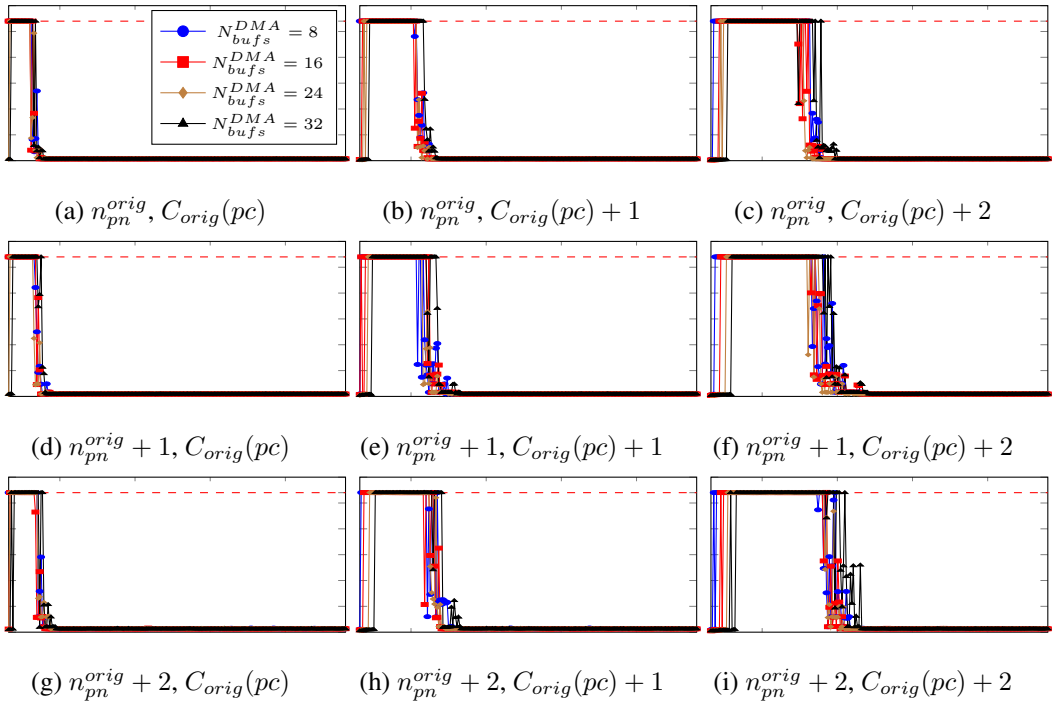


Figure 8.7: Comparison of computation times in function of the load increase with different budgets. The x axis represents the length of the application's hyperperiod (small value means high utilization) and the y axis represents the time used by the solver (timeout at 10,800 seconds)

Symmetric PCs means that in all budgets, all PCs are assumed to have identical periods and durations. In addition, all pairs of PNs are connected by PCs in both ways. These assumptions simplify the expression of budgets and enable an easy generation of many budgets following the goals mentioned in Section 8.3.1.

Prompt PCs means that the original duration of a PC (before being adjusted as explained in Section 8.3.1.2) is *close* to the smallest possible PC duration. The duration of PCs will typically be in the order of magnitude of the hypervisor's period and the PCs' periods will also be as short as possible.

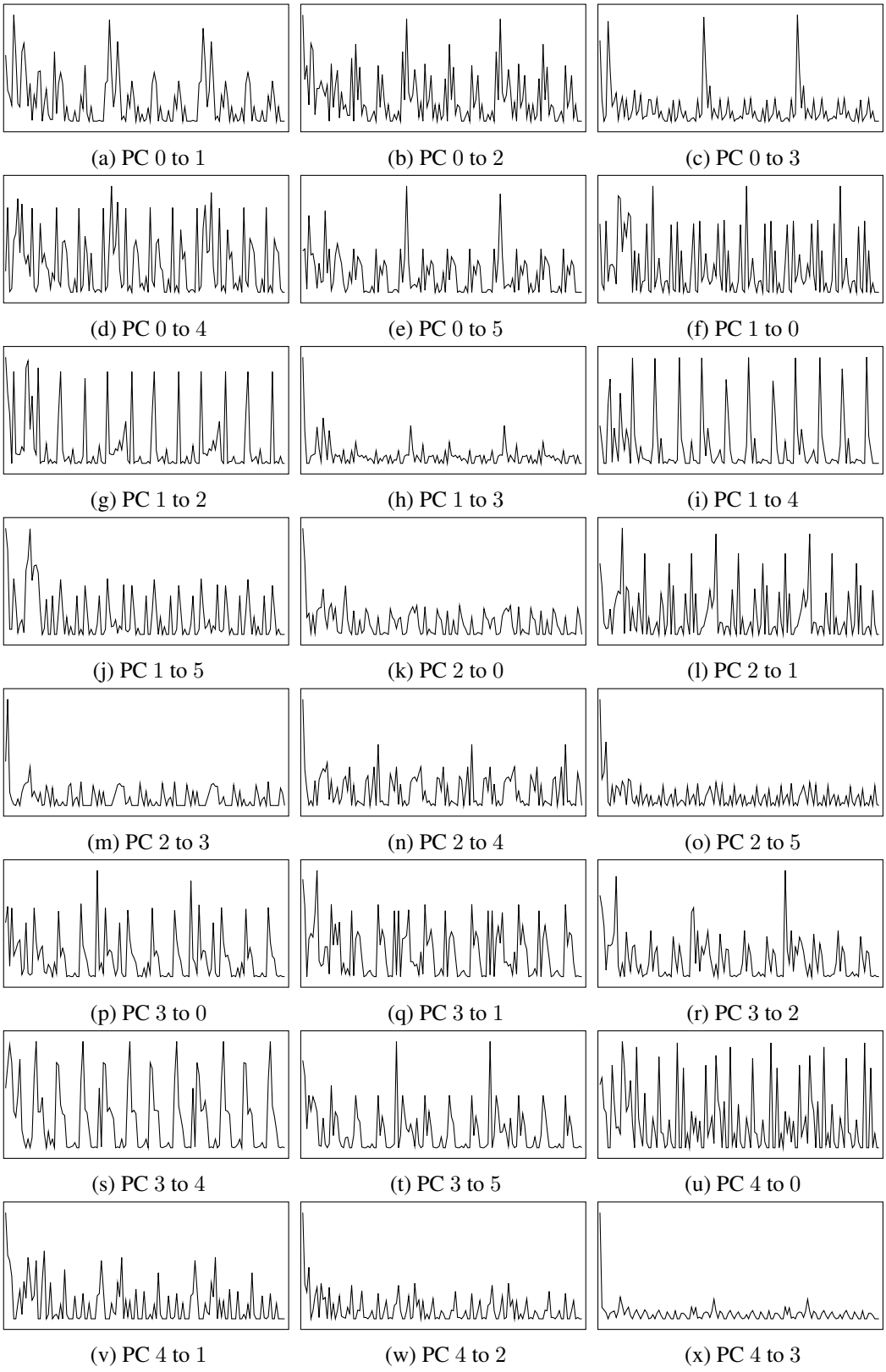
By using symmetric PCs, we aim at simplifying the automatic expression of budgets to perform parametric studies and we argue that prompt PCs should help shorten the delays involved by precedences with forward data. The evaluation of the CSP under different experimental conditions with different budgets featuring asymmetric or slow PCs appears as a good opportunity of future work.

8.3.3 Analysis of results

Figure 8.7 shows the time required by the solver to produce a solution to the CSP in function of various budgets and DMA configurations. The x axis represents the hyperperiod on which the application is scheduled. Short hyperperiods mean high utilization. The y axis represents the time required by the solver to find a solution to the CSP. A timeout is set to 10,800 seconds, thus explaining the saturation of curves on the left sides. Reaching the timeout means that no solution has been found and that the budget, given the application configuration, cannot be considered as *valid*. On right sides of curves, hyperperiods are longer and finding solutions can be done more

easily, thus validating the intuition that smaller utilizations lead to better schedulability. In contrary, it appears clearly in Figure A.23a that the smallest budget enables the solver to find solution up to the shortest hyperperiods, thus reaching the highest utilization ratios. Counter-intuitively, it means that highest utilization is reached with the smallest number of PNs, and thus the smallest amount of computational power. One may also note that the best performances are obtained when the durations of PCs are the smallest (Figures A.23a, A.23d and A.23g), while the number of PNs does not have a clear impact. Based on this, it can be argued that the bottlenecking parameter in this case study is the NoC delay and not the computational power. Moreover, while the WCET of the hypervisor was adjusted for each and every of the four DMA configurations, the impact of the N_{bufs}^{DMA} is not clear. Moreover, since the raw computational power of PNs does not seem to be a major issue in this case, it would be interesting as a future work to see whether relaxing our assumption on the number of PEs allocated to each PN would help improve the performance.

We show in Figures 8.8 and 8.9 the utilization curves of PCs for a given schedule produced by the CSP. Each curve depicts the total number of data that is sent at each activation of each PC slot over a complete hyperperiod. Figures 8.10 and 8.11 show similar curves with the number of non-contiguous memory areas sent instead of the total size of data. We can see that curves of Figures 8.8 and 8.9 appear to be less busy than Figures 8.10 and 8.11 where the limit imposed by the constraint 8.13 is often reached. Based on this, we can conclude that the NoC bottleneck of our case study rather comes from the DMA capability to handle many non-contiguous small data than an actual limitation due to insufficient bandwidth. As we mentioned in Section 8.2.3.4, optimizing the memory mapping of data during while computing the mapping and schedule of the application could help tackle this issue.



(a) PC 0 to 1 (b) PC 0 to 2 (c) PC 0 to 3

(d) PC 0 to 4 (e) PC 0 to 5 (f) PC 1 to 0

(g) PC 1 to 2 (h) PC 1 to 3 (i) PC 1 to 4

(j) PC 1 to 5 (k) PC 2 to 0 (l) PC 2 to 1

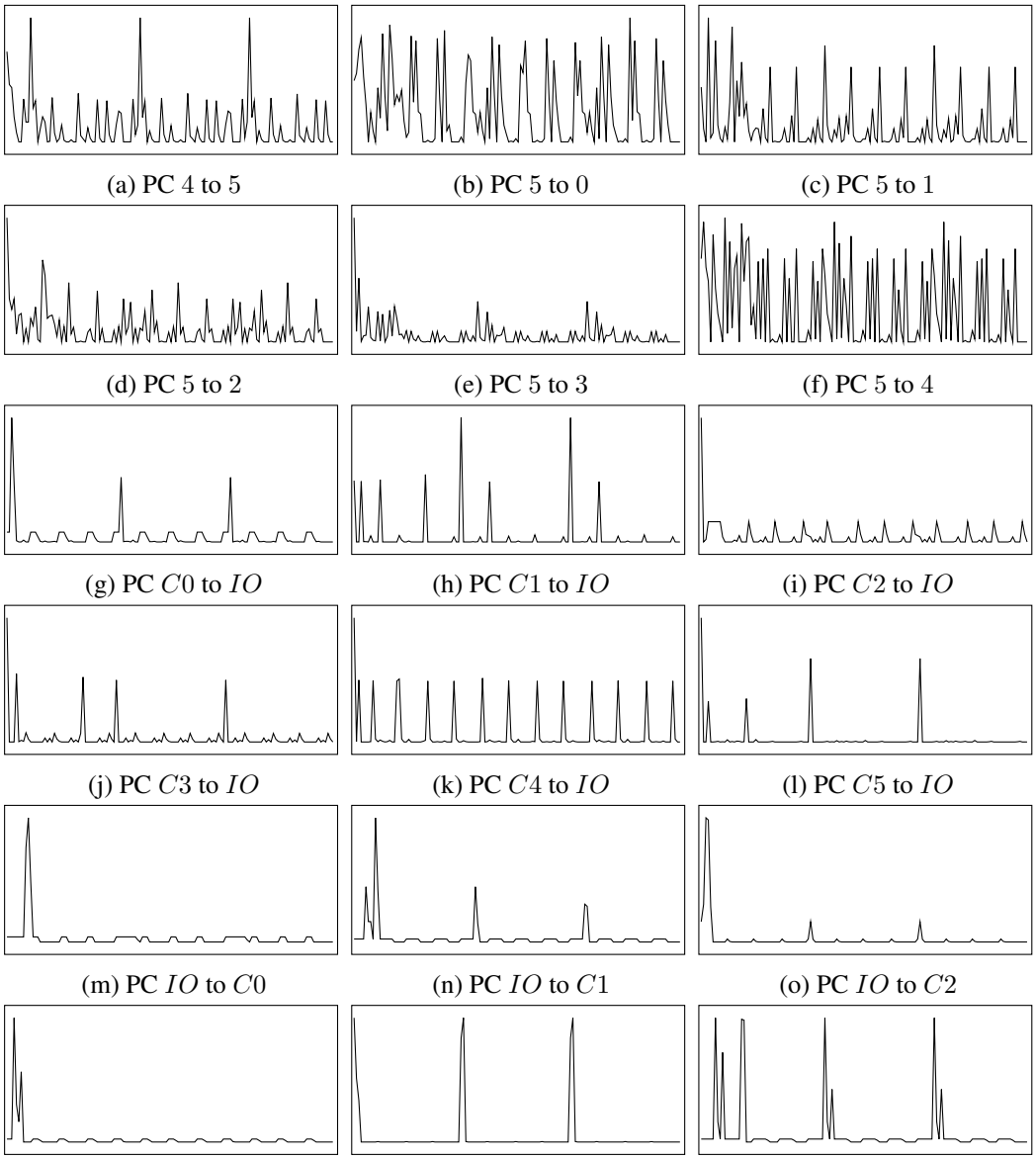
(m) PC 2 to 3 (n) PC 2 to 4 (o) PC 2 to 5

(p) PC 3 to 0 (q) PC 3 to 1 (r) PC 3 to 2

(s) PC 3 to 4 (t) PC 3 to 5 (u) PC 4 to 0

(v) PC 4 to 1 (w) PC 4 to 2 (x) PC 4 to 3

Figure 8.8: Total size of data per PC slot (1/2)



(a) PC 4 to 5 (b) PC 5 to 0 (c) PC 5 to 1

(d) PC 5 to 2 (e) PC 5 to 3 (f) PC 5 to 4

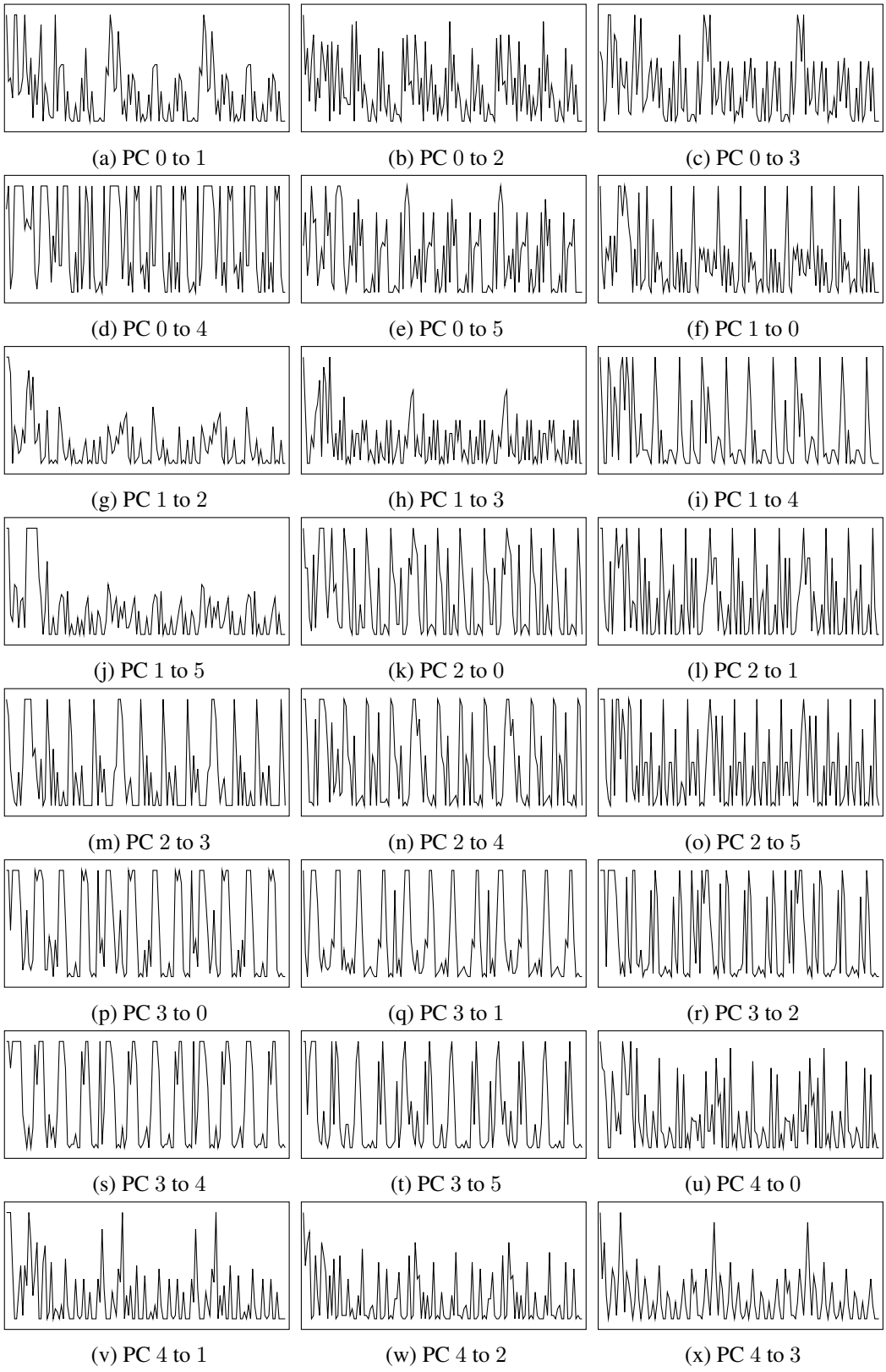
(g) PC $C0$ to IO (h) PC $C1$ to IO (i) PC $C2$ to IO

(j) PC $C3$ to IO (k) PC $C4$ to IO (l) PC $C5$ to IO

(m) PC IO to $C0$ (n) PC IO to $C1$ (o) PC IO to $C2$

(p) PC IO to $C3$ (q) PC IO to $C4$ (r) PC IO to $C5$

Figure 8.9: Total size of data per PC slot (2/2)



(a) PC 0 to 1 (b) PC 0 to 2 (c) PC 0 to 3

(d) PC 0 to 4 (e) PC 0 to 5 (f) PC 1 to 0

(g) PC 1 to 2 (h) PC 1 to 3 (i) PC 1 to 4

(j) PC 1 to 5 (k) PC 2 to 0 (l) PC 2 to 1

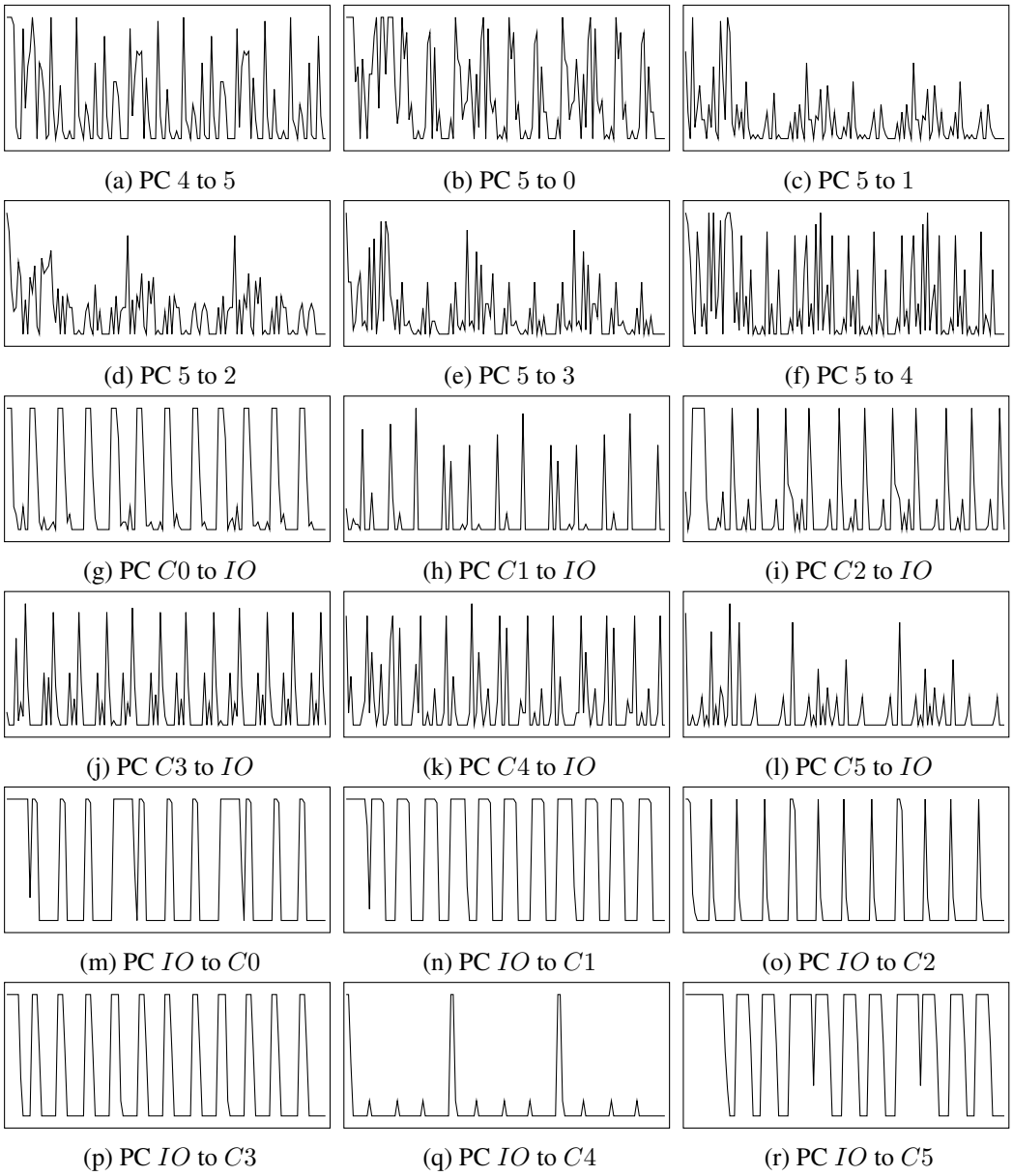
(m) PC 2 to 3 (n) PC 2 to 4 (o) PC 2 to 5

(p) PC 3 to 0 (q) PC 3 to 1 (r) PC 3 to 2

(s) PC 3 to 4 (t) PC 3 to 5 (u) PC 4 to 0

(v) PC 4 to 1 (w) PC 4 to 2 (x) PC 4 to 3

Figure 8.10: Number of data per PC slot (1/2)



(a) PC 4 to 5 (b) PC 5 to 0 (c) PC 5 to 1

(d) PC 5 to 2 (e) PC 5 to 3 (f) PC 5 to 4

(g) PC $C0$ to IO (h) PC $C1$ to IO (i) PC $C2$ to IO

(j) PC $C3$ to IO (k) PC $C4$ to IO (l) PC $C5$ to IO

(m) PC IO to $C0$ (n) PC IO to $C1$ (o) PC IO to $C2$

(p) PC IO to $C3$ (q) PC IO to $C4$ (r) PC IO to $C5$

Figure 8.11: Number of data per PC slot (2/2)

8.4 Summary

In this chapter, we detailed how the *Validate* phase of our work-flow can be implemented using constraint programming. We firstly clarified what our assumptions on budgets are and provided necessary conditions for budgets to be valid. Then, we explained how the notion of *Conditional Time-Intervals* can be used to model our problem and we detailed the actual CSP formulation. Based on this, we evaluated the capability of our approach for several different budgets, DMA configurations and application work-loads. By doing so, we demonstrated that such a formal approach can scale up to industrial-sized instances of the problem and that the computational power of the platform can handle heavy work-loads. Yet, we identified the problem of optimizing the memory mapping of as an interesting challenge to further improve DMA utilization and our approach overall, thus opening the way to future contributions.

Part IV

Conclusion

Chapter 9

General conclusion

Contents

9.1 Summary of contributions 145
9.1.1 Main contributions 145
9.1.2 Limitations of the approach 147
9.2 Future perspectives 148
9.2.1 Execution model and implementation 148
9.2.2 Data management during validation 149
9.2.3 Automated allocation of concrete resources to budgets 150
9.2.4 Long term perspectives 150

9.1 Summary of contributions

This thesis addressed the problem of using many-core processors in a predictable manner. We aimed at leveraging their parallel computational power to support the execution of multiple isolated partitions and to process massive and constrained workloads. Overall, the avionics and industrial context of this work involved specific constraints. For cost and performance reasons, we focused on the management of COTS processors only. In particular, we applied our approach on the KALRAY MPPA®-256. Secondly, we considered that WCETs of programs must be computed using static analysis techniques, thus implying the need for either time-compositional or time-composable systems. In addition, we assumed all configuration, mapping and scheduling activities to be achieved off-line in order to match industry practices. And finally, industrial context required any mapping or scheduling technique to scale in a reasonable amount of time to large applications.

In the remainder of this section, we provide an overview of the main contributions of the thesis and a discussion regarding the limits of the approach.

9.1.1 Main contributions

9.1.1.1 Integration framework

In order to run concurrent applications in isolation on the KALRAY MPPA®-256, we proposed a complete integration framework. Overall, the whole framework relies on the property of temporal isolation provided by an execution model. The framework takes as inputs a set of partitions that

are expected to be executed in a time-composable manner. Each partition contains an application and a resource budget allocated to it. Each budget is validated by verifying that the amount of resource it contains is sufficient to properly execute the application. Then the budgets of all partitions are allocated to concrete hardware resources of the processor. Eventually, temporal isolation between partitions is enforced at run-time by a hypervisor that implements the rules of our execution model. The purpose of this framework is to allow applications designer to design, develop, verify and test their applications without any knowledge of co-running applications. In particular, it enables applications to be certified independently and thus fulfils the industrial need for incremental certification.

9.1.1.2 Execution model

To enable several applications to run concurrently on the KALRAY MPPA®-256 without interfering with each other, we propose to use an execution model. An execution model is a set of rules constraining the acceptable behaviours of applications in order to avoid unpredictable accesses to shared resources. Our execution model is based on an analysis of the KALRAY MPPA®-256 and focuses on the three main sources of interferences on this platform, namely the local memories of compute clusters, the NoC and the external DDR-SDRAM. Inside compute cluster, we propose to use pure spatial partitioning by allocating no more than one partition to each local memory bank and each core. At the NoC level, we avoid competition using a global TDM schedule coupled with an appropriate route allocation of communication channels. At the DDR-SDRAM level, we also enforce a TDM schedule at the bank level but not between banks in order to eliminate row switching costs for co-running partitions. Finally, we require applications designers to define off-line the memory areas that will be sent at run-time to enable off-line verification of the NoC budget. Overall, this execution model drastically mitigates inter-partition interferences and provides a time-composable execution environment. The design of this execution model is the core of our first publication [71].

9.1.1.3 Implementation of the execution model

In order to enforce the rules of the execution model at run-time, we developed a hypervisor which runs underneath applications and limits their accesses to shared resources. More precisely, the hypervisor manages boot of all cores and configures the hardware globally. The spatial partitioning inside compute clusters is enforced using frozen MMU configurations and fixed exception handlers. An instance of the hypervisor is executed by the RM of each compute cluster and manages the DMA to apply the NoC schedule. Since accesses to DDR-SDRAM require DMA transaction through the NoC, computing an appropriate NoC schedule off-line is enough to meet the constraint without further on-line support. Finally, we validated the correctness of our application using several experimental benchmarks based on the ROSACE case study. As expected, benchmarks exhibited a clear temporal isolation between co-running partitions despite sharing resources such as the NoC or the DDR-SDRAM. This work on the hypervisor and the experimental benchmarks was published in [162].

9.1.1.4 Automated budget validation

Since our work is required to apply on industrial-sized applications, the handmade validation of resource budgets can quickly become prohibitive. To tackle this issue, we propose to automatically validate budgets by computing a schedule of the applications using constraint programming. We

leverage the features of modern CP solvers such as the notion of Conditional-Time Intervals introduced in IBM ILOG CP Optimizer in order to deal with large problem instances. In particular, we provide an extensive parametric evaluation of our CSP formulation using an industrial case study from Airbus. We show that the approach not only provides correct schedules of the applications in a few minutes, but also that it can handle more demanding workloads and a variety of different budgets. This contribution has been originally published in [163].

9.1.2 Limitations of the approach

The approach proposed in this thesis correctly solves most of the problems related to the predictable execution of parallel applications on many-core processors. Moreover, it meets all the constraints involved by the industrial and avionics context. Yet, it obviously has some limitations that we detail below.

9.1.2.1 Limits of the execution model

The purpose of an execution model is to constrain the behaviour of applications to avoid undesired situations. The rules imposed to applications designers have an impact on their designs and are usually focused on specific application topologies. In our case, the choices behind our execution model are tailored to avionics applications. Consequently, any application that is out of this scope may face difficulties to be integrated within our framework. In particular, knowing prior to execution exactly which data should be sent at each TDM NoC slot is not a problem in avionics since off-line computation is a standard industry practice. However, other applications may require more flexibility regarding communication schedules. The passive communication paradigm of our approach can be a blocking point for applications needing on-line communication support.

9.1.2.2 Limits of the hypervisor

Our implementation of the execution model through a hypervisor also introduces limitations on what applications can do with the hardware. The design of the hypervisor imposes that lengths and periods of TDM slots must be multiple of the hypervisor's period and thus lower bounds NoC latencies for short messages. In addition, the hypervisor's period is lower bounded by its WCET which depends on the number of data that DMA can handle autonomously. Making a choice regarding this parameter will favour a certain type of applications and thus handicap others.

Moreover, the current hypervisor's implementation relies on frozen MMU configuration and fixed exception handlers. Clearly, making this assumption raises a major issue to run applications on top of a guest operating system. This is not an issue for safety-critical avionics applications that typically run in bare-metal. However, for more dynamic applications, not being able to issue system calls may become a concern and require deep modifications of applications to replace OS dependencies by embedding libraries.

9.1.2.3 Limits of the CSP-based validation

Our CSP formulation of the budget validation problem relies on two assumptions. Firstly, assuming that no code is ever fetched from the external DDR-SDRAM is a limitation. With 32MiB of SRAM in the compute clusters of the KALRAY MPPA®-256, our approach will fail on any application with larger code. Again, making such an assumption is usually not a problem for safety-critical control applications where the size of the code usually stays in reasonable boundaries. However, other types of applications may face difficulties to fit completely in 32MiB.

Secondly, it is assumed in the current CSP formulation that data locations in memory are unknown. Consequently, we make the conservative assumption that no data are contiguous and that DMA can never send concatenated data. Since DMA jumps from one memory location to another take time, making this assumption can limit the overall performance. In particular, we identified in our case study that DMA performance was the bottleneck for performing higher speedups. This assumption leads to potentially sub-optimal schedules and may not be able to validate budgets for which a solution exists when data are concatenated.

Overall, most of the limitations listed above can be eliminated or at least mitigated by optimizing or modifying some parts of the approach. We discuss these opportunities for ameliorations in the next section.

9.2 Future perspectives

We see several direct opportunities to further improve our work. This involves changes at all levels of the framework, including the execution model and its implementation, the automated validation procedure or the introduction of an automated allocation of concrete resources to budgets. In addition, we also see several opportunities to leverage many-core processors in even more disruptive applications that may become of major interest in future aircrafts. Both are detailed in the remainder of this section.

9.2.1 Execution model and implementation

Our execution model is tailored to avionics applications. In particular, requiring the knowledge of all data to be sent over the NoC prior to execution can be argued as particularly constraining. Relaxing this constraint could add significant flexibility to the approach and enable more dynamic applications to be handled efficiently. However, doing so involves deep changes in the design of the hypervisor. To gain in flexibility, there are two main options: changing the scheduling scheme of the NoC or extend the current hypervisor to manage on-line communication requests.

9.2.1.1 Asynchronous NoC Schedule

One may decide to drop the TDM schedule of the NoC and to use the hardware limiters of the KALRAY MPPA®-256's NoC interfaces instead. Using network calculus, it may be possible to compute WCTTs of NoC packets and thus to temporally validate applications. In addition, the software support for NoC communications could be significantly reduced. Moreover, periods of communications would not need to be multiples of the hypervisor's period anymore. However, using that sort of asynchronous scheme also has some drawbacks. Although network calculus is perfectly suitable to guarantee bandwidths to applications, the guarantees regarding latencies are likely to be large. This is not due to pessimism in calculation, it is the cost of asynchronism. In addition, dropping the TDM scheme at the NoC level involves to drop it at the DDR-SDRAM level as well. Consequently, the row conflicts in banks could not be avoided by construction anymore. This leaves two choices to designers:

1. they can enforce pure spatial partitioning at the DDR-SDRAM bank level. This means having no more than one partition allocated to each bank. As a consequence, no more communications between partitions using shared memory can be achieved and the number of partitions using DDR-SDRAM is upper-bounded by the total number of DDR-SDRAM banks.

2. they can allow several partitions to share a bank and thus avoid the problems mentioned above. However, since row conflicting requests in banks are now possible, they must be accounted when computing memory access time. Consequently, the guaranteed DDR-SDRAM throughput is likely to decrease while guaranteed memory access latencies are likely to grow.

In either cases, managing communications on-line using the KALRAY MPPA®-256's packet shapers involves significant modifications of the approach and may, in some case, come at the cost of higher guaranteed latencies. We argue that investigating the trade-offs of such designs is an interesting opportunity for future work.

9.2.1.2 On-line communication request over TDM schedule

Relaxing the NoC communication scheme with on-line requests rather than a passive approach seems possible without dropping the TDM schedule. Assuming that NoC communications are still performed in a TDM fashion, one may imagine the hypervisor to send data in the TDM slot in response to requests from applications on-line. In this case, applications could asynchronously submit communication requests in a queue. At the beginning of a TDM slot, the hypervisor would dequeue a request and decide to perform it or not. Depending on the amount of data requested, the hypervisor would have to check whether the TDM slot is long enough to send all the data and to avoid running over the TDM schedule. In a safety critical context, this decision would be critical to avoid TDM violations. Several solutions are possible. If a request is identified as too large for a TDM slot, the hypervisor may decide to either refuse to process it and send an error to the initiating application, or to split it in sub-requests to be processed over several TDM slots. Overall, enabling on-line requests will lead to an increased complexity of the hypervisor anyway. The decision making algorithm would have to be safe and fast to avoid system failures or prohibitive overheads. Finally, TDM slots would have to be reserved for all communication channels, no matter how they are used. For an application with uneven communication needs, the reservation to meet peak load will leave many slots empty during idle periods and lead to low NoC utilization.

Overall, introducing on-line request processing into the hypervisor raises several challenges regarding its implementation and the allocation of TDM slots to partitions. In particular, there are critical concerns regarding robustness and overheads. Yet, exploring this path appears as a good opportunity for future work.

9.2.2 Data management during validation

The management of data in our CSP formulation of the validation problem can be optimized. We saw during experiments that DMA jumps between non-contiguous memory areas are time consuming. By assuming that data locations are not known, we greatly simplify the CSP and enable it to scale to large applications using simple cumulative functions. Yet, this is a conservative assumption leading to potentially sub-optimal results. In order to improve the current validation procedure, data locations may be optimized using one of the three following methods:

1. data locations may be fixed in pre-processing. Before solving the CSP, one may compute a mapping of the application's data. Once fixed, the CSP formulation would have to be extended to account for this mapping in order to avoid the N_{flit}^{gap} when two contiguous data are sent in the same TDM slot. In this case, computing such a mapping prior to solving is not obvious since the location of sub-tasks which produce the data are not known. Thus, a *good* pre-processing data optimizer would place contiguously data produced by sub-tasks that are

likely to be assigned the same cluster. Although this may improve results overall, it is clear that this approach remains sub-optimal.

2. data locations may be computed together with the mapping of the other application's elements. In this case, several decision variables should be added to the CSP in order to encode the resulting location of each data on each cluster. In this case, the CSP formulation needs to be extended to compute the location of the data in memory and to use this information in the mapping of sub-tasks to clusters. While this approach will provide the best results, it will also greatly harden the problem. When it comes to mapping large applications, it is not clear whether such complexity increase can be handled or not.

3. data locations may be optimized in post-processing. After solving the CSP without modifications, the allocation of sub-tasks to clusters and data to TDM slots will be known. Based on this, one may now optimize locations of memory mapping to concatenate as many data as possible and reduce the number of DMA jumps during TDM slots. By doing so, DMA will process data more efficiently and will probably finish their work earlier than expected in many cases. Such an optimization does not really help to find more solutions, it will rather improve an existing solution that was computed using the conservative approach and increase its margins. However, it is possible that some infeasible mappings computed with fake parameters in the conservative approach may actually become feasible once optimized in post-processing. Although the conservative approach will probably be preferred in the industry, it may happen that only this optimistic approach is able to find schedules.

Overall, it is clear that optimizing the locations of data in local memories is important. Pre- and post-processing optimization are likely to be easier but remain sub-optimal. Introducing the data locations as decision variables in the CSP is clearly going to provide the best solution but also adds a lot of complexity. The evaluation of the benefits and drawbacks of these three approaches is an interesting perspective of amelioration.

9.2.3 Automated allocation of concrete resources to budgets

As we mentioned earlier in this thesis, the allocation of concrete resources to partitions automatically may be implemented using constraint programming. One of the main problems regarding this task is to allocate routes and offsets to PCs while simultaneously allocating clusters to PNs. It appears that allocating the offsets to PCs is close to scheduling problems in *Offset free systems* [164]. In essence, the offset-assignment problem in strictly periodic systems has been studied for a long time [165] and previous results are now used to build more complex scheduling algorithms using ILP [166, 167]. Overall, it seems possible to extend those results with additional constraints on route assignments to PCs and cluster assignments for PNs. We argue that providing such extensions could be a valuable addition to our work.

9.2.4 Long term perspectives

Using distributed systems to process heavy workloads is not a new idea. For example, Google introduced MapReduce [168] in 2004 to ease the execution of parallel programs on large clusters of machines. In this thesis, we focused on the KALRAY MPPA®-256 which actually shares many of the properties of such distributed system, but at a different scale. Porting the techniques applied at macroscopic size within distributed but embedded systems thus appears as a particularly interesting research opportunity. Moreover, new types of software are likely to find their path to aerospace

applications in the future. Machine learning algorithms are good examples of such new workloads that are already processed by embedded systems in other industries. Since most of these algorithms heavily rely on matrix arithmetics, they usually benefit from parallelization techniques. Again, many-core processors appear here as good candidates to support these workloads and thus to design future avionics computers.

Part V

Résumé étendu en français

Chapitre A

Exécution prédictible sur processeurs pluri-cœurs

Ce travail a été financé par Airbus et l'Association Nationale de la Recherche et de la Technologie (ANRT) dans le cadre du contrat CIFRE n°2013/1394.

A.1 Introduction

Cette thèse s'inscrit dans le cadre général de la conception de systèmes avioniques temps réels critiques. Plus précisément, l'objectif est d'étudier les possibilités d'utilisation des processeurs pluri-cœurs pour concevoir ces systèmes. Globalement, le contexte de cette thèse est triple. D'abord, le contexte avionique impose des contraintes fortes de sûreté sur la conception de tels systèmes, et en particulier des contraintes de certification, notamment lors des développements logiciels. Deuxièmement, la dimension industrielle impose de concevoir des systèmes efficaces qui seront capables de supporter l'augmentation des besoins en puissance de calcul des futures applications embarquées. Enfin, le contexte technologique offre des perspectives intéressantes pour répondre à ces deux besoins en utilisant une technologie émergente prometteuse: les processeurs pluri-cœurs.

A.1.1 Contexte de la thèse

Les avions doivent être sûrs. Ainsi, les systèmes qui peuvent avoir un impact sur la sûreté du vol doivent être conçus avec une attention toute particulière. Le monde de l'aéronautique civile est régi par des autorités de régulations indépendantes qui vérifient la sûreté des avions avant de permettre leur utilisation opérationnelle. Plus précisément, les avionneurs doivent démontrer aux autorités le respect des exigences qu'elles fixent afin d'obtenir la *certification* de leur appareil, synonyme d'autorisation pour le vol de civils. Aujourd'hui, les exigences des autorités de certification sont essentiellement communiquées au travers de standards tels que la DO-178C [4] pour les développements logiciels. De manière générale, la DO-178C contient majoritairement des contraintes sur le processus de développement des logiciels. Cependant, certaines exigences contraignent également les choix de conception qui peuvent être faits. En particulier, la DO-178C impose que, pour les logiciels les plus critiques, les temps d'exécution pire-cas (ou *WCETs* en anglais) des programmes soient calculés. Malheureusement, le calcul de WCETs à la fois sûrs et peu pessimistes est de plus en plus difficile à mesure que les architectures des processeurs se complexifient.

Dans un milieu industriel, la conception des systèmes est non seulement contrainte par une exigence de sûreté, mais également par un besoin de performance pour soutenir la croissance des

applications embarquées et par la nécessité de maintenir un coût bas pour des raisons commerciales. L'augmentation du besoin en puissance de calcul à bord est liée à deux tendances. D'abord, l'introduction de nouvelles fonctionnalités qui augmentent les performances de l'avion implique souvent une charge de travail additionnelle pour les calculateurs embarqués. Les innovations au niveau des systèmes de contrôle-commande tels que le *Gust Load Alleviation* dans les commandes de vol de l'A350 permettent notamment des gains de masse importants en réduisant la fatigue structurelle des avions, mais au prix d'une charge de travail accrue pour le processeur. Les systèmes du cockpit du futur sont en ce moment au centre de plusieurs projets d'amélioration qui utilisent globalement toujours plus d'électronique embarquée. A l'avenir, de nouveaux types de programmes utilisant par exemple des méthodes d'apprentissage automatique pourraient trouver une application dans un contexte avionique et par conséquent amener de nouveaux besoins en puissance de calcul. Orthogonalement, la tendance de conception initiée depuis l'A380 avec l'*Avionique Modulaire Intégrée* (ou l'*IMA* en anglais) se confirme. Les avionneurs visent un futur où les calculateurs comportant plusieurs fonctions en hébergeront de plus en plus. De ce fait, le nombre de calculateurs pourrait être diminué, ainsi que la masse totale, la consommation énergétique et les efforts de maintenance associés au support des équipements avioniques. Le partage d'une cible d'exécution par plusieurs systèmes pose plusieurs problèmes techniques intéressants afin de pouvoir garantir que la faute d'un système ne se propage pas à d'autres systèmes et pour permettre de concevoir, développer, vérifier, tester et certifier les systèmes indépendamment les uns des autres pour des raisons de coût. De manière générale, un contexte industriel implique deux défis principaux pour la conception de systèmes avioniques du futur. L'augmentation du besoin en puissance de calcul liée à la croissance des applications doit être soutenue pour augmenter encore davantage les performances des avions alors que les calculateurs devront héberger de plus en plus d'applications concurrentes pour réduire les coûts. Ainsi, la résolution de ces deux problèmes simultanément requiert l'utilisation de plateformes d'exécution qui soient non seulement calculatoirement performantes mais également calculatoirement partageables.

Dans le même temps, le paradigme architectural des microprocesseurs modernes évolue. En 2004, l'introduction des architectures multi-cœurs a permis de continuer l'augmentation des performances des processeurs au moment où les optimisations sur un cœur telles que les pipelines profonds, les branchements spéculatifs ou l'exécution dans le désordre ne devenaient plus énergétiquement rentables. Aujourd'hui, ces architectures multi-cœurs, qu'elles soient basées sur une interconnexion par bus ou par crossbar, font également face à des difficultés pour exploiter efficacement les transistors fournis par la loi de Moore. Une solution à ce problème a été proposée avec l'introduction de processeurs massivement parallélisés, appelés processeurs pluri-cœurs. Ceux-ci sont habituellement conçus avec une architecture composée de *tuiles* interconnectées par un ou plusieurs *Réseaux-sur-puce* (ou *NoC* en anglais). De ce fait, ce type d'architecture offre de meilleures opportunités pour le passage à l'échelle vers des dizaines, des centaines, voire des milliers de cœurs à intégrer sur une seule puce. Aujourd'hui, plusieurs processeurs conçus selon ce paradigme sont disponibles sur le marché. Si la majorité d'entre eux, incluant les Intel Xeon Phi [10] et SCC [14], l'Adapteva Epiphany [11] ou les Mellanox Tile Gx* [15] et Mx* [16], comporte des caractéristiques non satisfaisantes pour être embarquables dans un avion (une dissipation thermique trop importante ou une puissance de calcul trop faible par exemple), le cas du KALRAY MPPA®-256 [17] apparaît quant à lui comme un compromis entre puissance de calcul et puissance dissipée plus adapté compte tenu des besoins des applications Airbus.

A.1.2 Problématiques

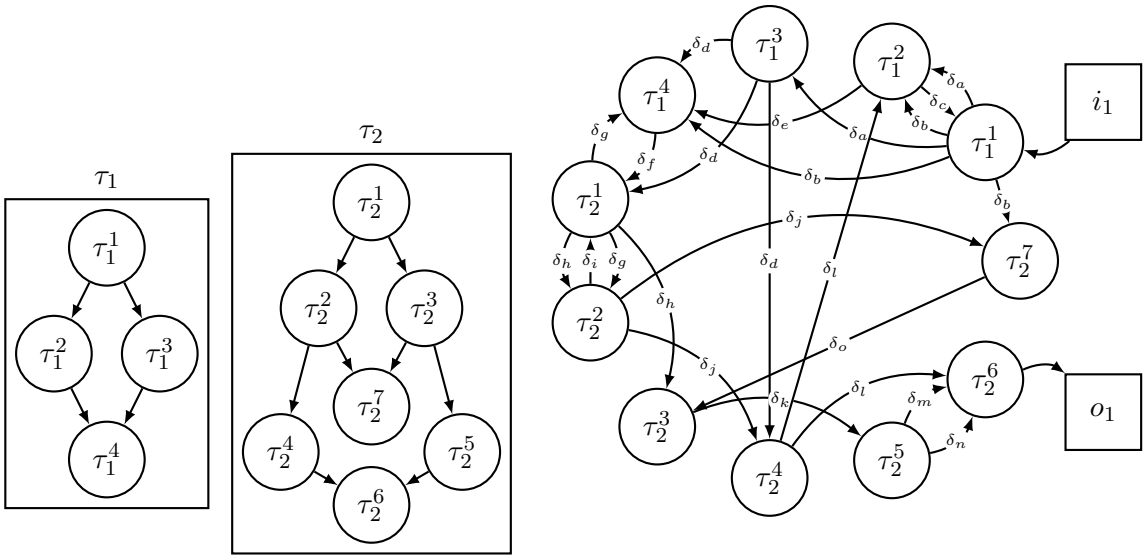
Dans un contexte avionique où les logiciels doivent être certifiés et un contexte industriel où des applications de plus en plus grosses doivent partager leur plateforme d'exécution, le KALRAY MPPA®-256 offre des perspectives de conception intéressantes. Cependant, plusieurs défis techniques doivent être relevés avant son utilisation. Premièrement, pour des raisons de certification, les WCETs des programmes doivent être calculés. Malheureusement, cette tâche est de plus en plus difficile sur les processeurs modernes. En particulier, les travaux effectués sur les processeurs multi-cœurs ont montré que lorsque l'on considère du logiciel parallèle, la difficulté majeure pour le calcul des WCETs est liée à la gestion des interférences sur les ressources partagées. Certaines ressources des processeurs pluri-cœurs étant massivement partagées, le premier défi à relever consistera en la maîtrise des interférences sur ces ressources.

Deuxièmement, l'architecture distribuée des processeurs pluri-cœurs rend leur programmation complexe. Le KALRAY MPPA®-256 possède une architecture hiérarchique avec plusieurs types de périphériques, de mémoires, de réseaux-sur-puces et de cœurs de calcul. Pour garantir la sûreté, les concepteurs de logiciels critiques doivent avoir un contrôle total du matériel et par conséquent être capables de programmer et de configurer toutes les parties du processeur à un très bas niveau. Dans le cas d'une architecture complexe comme celle du KALRAY MPPA®-256, relever ce défi s'avère particulièrement difficile.

Dernièrement, compte tenu de la complexité des architectures considérées, le placement et l'ordonnancement d'applications sur un processeur pluri-cœurs semblent difficilement réalisables à la main, et en particulier lorsque les applications ciblées sont de grande tailles. Fournir des outils permettant de réaliser ce travail automatiquement constituerait un atout considérable pour permettre l'utilisation de processeurs pluri-cœurs dans un contexte industriel. Cependant, identifier les méthodes qui permettront à la fois la modélisation d'une architecture de processeur complexe et le passage à l'échelle sur des applications de taille industrielle sera un défi majeur pour la réalisation de tels outils.

A.1.3 Approche

Afin de répondre aux problématiques soulevées dans la section précédente, nous proposons tout d'abord une analyse détaillée du KALRAY MPPA®-256 et notamment des mécanismes mis en jeu lors des accès aux ressources partagées. Sur la base de cette analyse, nous proposons un *modèle d'exécution* contraignant l'accès à ces ressources afin d'isoler temporellement les applications s'exécutant simultanément sur le processeur. Le détail de ce modèle d'exécution est décrit dans la Section A.4. Nous proposons dans la Section A.5 un atelier d'intégration permettant la conception, le développement, la vérification, le test et la certification des applications indépendamment les unes des autres, et ce malgré le partage de la plateforme d'exécution. Pour ce faire, les contraintes imposées par le modèle d'exécution sont garanties d'être respectées en-ligne grâce à un hyperviseur dont nous détaillons le fonctionnement en Section A.6. En outre, nous proposons en Section A.7 une approche basée sur la programmation par contraintes pour résoudre le problème du placement et de l'ordonnancement hors-ligne d'applications de grandes tailles sur le KALRAY MPPA®-256. Enfin, nous identifions en Section A.8 les limites de notre approche et les perspectives d'amélioration et de travaux futurs pour l'utilisation de processeurs pluri-cœurs dans un contexte embarqué critique.



(a) Relations de précédence

(b) Échanges de données et buffers I/O

Figure A.1: Exemple de modèle d'application

A.2 Entrées de la thèse

Dans cette section, nous détaillons les entrées de notre travail imposées par le contexte industriel. Nous présentons le modèle d'application puis le processeur sur étagère retenu par Airbus en expliquant ce choix.

A.2.1 Modèle d'application

Le modèle d'application considéré dans cette thèse est représentatif des applications de contrôle existantes à Airbus. Une application est une paire $< \tau, \delta >$ où τ est un ensemble fini de tâches et δ un ensemble fini de données. Chacune des tâches comprises dans $\tau = \{\tau_1, \ldots, \tau_n\}$ est définie par $\tau_i =< S_i, P_i, T_i >$ avec:

- $S_i = \{\tau_i^1, \ldots, \tau_i^{n_i}\}$ un ensemble de n_i sous-tâches. Toutes les sous-tâches sont activées simultanément à l'activation de τ_i et elles partagent toutes la même échéance implicite égale à la période de τ_i. Chaque sous-tâche est modélisée par un quadruplet $\tau_i^j =< C_i^j, M_i^j, I_i^j, O_i^j >$ avec:
  - C_i^j le temps d'exécution pire-cas de τ_i^j;
  - M_i^j l'empreinte mémoire de τ_i^j, c'est-à-dire la somme de la taille de son code et de ses données statiques et rémanentes;
  - I_i^j et O_i^j respectivement les buffers d'entrée et de sortie dans lesquels τ_i^j va lire et/ou écrire pour dialoguer avec les composants externes tels que la mémoire DDR-SDRAM.
- $P_i \in S_i \times S_i$ un ensemble de paires ordonnées de sous-tâches représentant des relations de précédence contraignant l'ordre dans lequel elles peuvent être exécutées. Plus précisément, $(\tau_i^x, \tau_i^y) \in P_i$ signifie que τ_i^x doit terminer son exécution avant que τ_i^y puisse démarrer. P_i impose une relation d'ordre partiel (par conséquent acyclique) entre les sous-tâches qui peut être représentée par un Graphe Orienté Acyclique (ou *DAG* en anglais).

Sous-tâches	C_i^j (en ck)	M_i^j (en KiB)	I_i^j	O_i^j
τ_1^1	500	519	i_1	$\varnothing$
τ_1^2	500	397	$\varnothing$	$\varnothing$
τ_1^3	700	642	$\varnothing$	$\varnothing$
τ_1^4	400	262	$\varnothing$	$\varnothing$
τ_2^1	1,000	287	$\varnothing$	$\varnothing$
τ_2^2	400	799	$\varnothing$	$\varnothing$
τ_2^3	500	542	$\varnothing$	$\varnothing$
τ_2^4	800	764	$\varnothing$	$\varnothing$
τ_2^5	900	490	$\varnothing$	$\varnothing$
τ_2^6	500	399	$\varnothing$	o_1
τ_2^7	600	12	$\varnothing$	$\varnothing$

Tableau A.1: Paramètres des sous-tâches

- T_i la période de τ_i. T_i sert également de contrainte d'échéance implicite pour l'ensemble des sous-tâches de τ_i.

Nous définissons *l'hyper-période* de ce système de tâches comme $H = \underset{i\in[1,n]}{ppcm}(T_i)$ où $ppcm$ est la fonction renvoyant le plus petit commun multiple.

Les données $\delta = \{\delta_1, \ldots, \delta_m\}$ sont produites et consommées par les sous-tâches composant l'application. En définissant $S = \bigcup_{\tau_i\in\tau} S_i$ comme l'ensemble de toutes les sous-tâches, chaque donnée δ_k peut être définie par un triplet $\delta_k =< m_k, prod, cons >$ où:

- m_k représente la taille de la donnée;
- $prod : \delta \mapsto S$ associe chaque donnée avec la sous-tâche qui la produit;
- $cons : \delta \mapsto 2^S$ associe chaque donnée avec un ensemble de sous-tâches la consommant.

Toute paire de sous-tâches, même si non-contraintes par une relation de précédence, peut échanger des données. En revanche, dans notre modèle, toute paire de sous-tâches soumise à une relation de précédence implique systématiquement que les sous-tâches échangent au moins une donnée.

Notre modèle ne permet pas d'imposer de contrainte de précédence entre des sous-tâches n'appartenant pas à la même tâche. Ainsi, l'ordre de production et de consommation des données partagées par plusieurs tâches ne peut pas être contraint. Ici, l'hypothèse sous-jacente est que l'utilisation qui est faite de ces données inter-tâches est fonctionnellement robuste au plus grand délai possible entre leur production et leur consommation. Notre modèle impose que la consommation d'une donnée soit toujours celle de la valeur la plus fraîche. Ainsi, le délai production-consommation maximum d'une donnée est égal à la période de la sous-tâche productrice.

Exemple 1 (Modèle d'application)**.** *Nous considérons une application composée de 2 tâches τ_1 et τ_2 qui sont respectivement composées de 4 et 7 sous-tâches avec des périodes $T_1 = 24$ et $T_2 = 48$. Les 11 sous-tâches échangent 15 données et lisent et écrivent dans 1 buffer d'entrée i_1 et 1 buffer de sortie o_1. Les relations de précédence ainsi que les échanges de données sont représentés sur la Figure A.1. Les paramètres des sous-tâches sont listés dans le tableau A.1 (ou* ck *représente un nombre de cycle d'horloge).*

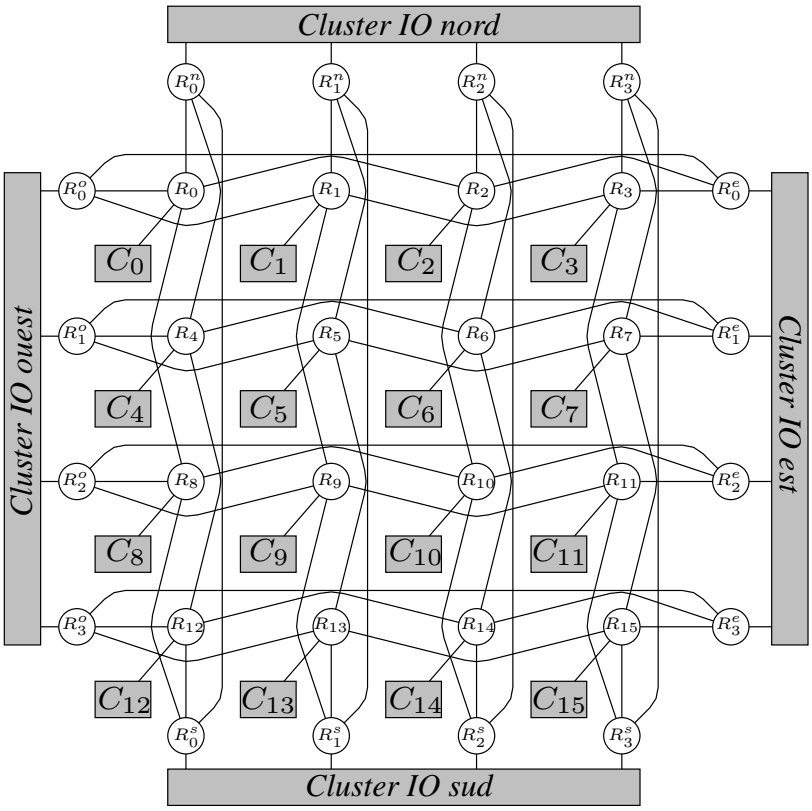


Figure A.2: Architecture du KALRAY MPPA®-256

A.2.2 Architecture du KALRAY MPPA®-256

Le KALRAY MPPA®-256 [17] est un processeur pluri-cœurs comprenant 288 cœurs sur une seule puce. Il est particulièrement adapté aux applications temps réel nécessitant une puissance de calcul importante de part sa faible dissipation thermique ainsi que ses bonnes propriétés temporelles. Dans le cadre de cette thèse, nous utilisons la deuxième version du KALRAY MPPA®-256 dénommée *Bostan*.

Comme montré sur la Figure A.2, le KALRAY MPPA®-256 comporte 16 *clusters* dédiés au calcul (les C_i sur la figure). De plus, à la périphérie du processeur se trouvent 4 clusters *IO*, c'est-à-dire dédiés à la gestion des interactions avec les composants externes tels que la DDR-SDRAM ou le réseau Ethernet. Les clusters sont interconnectés par 2 Réseaux-sur-Puces (ou *NoC* en anglais) qui permettent des communications point à point par l'envoi de message.

A.2.2.1 Les clusters de calcul

Comme cela est montré sur la Figure A.3, chaque cluster de calcul comporte:

- 16 cœurs de calcul, appelés les *Processing Elements* (ou *PE*). Les PEs sont dédiés à l'exécution de code utilisateur.

- un cœur supplémentaire, appelé le *Resource Manager* (ou *RM*) qui gère l'ensemble des ressources locales du cluster.

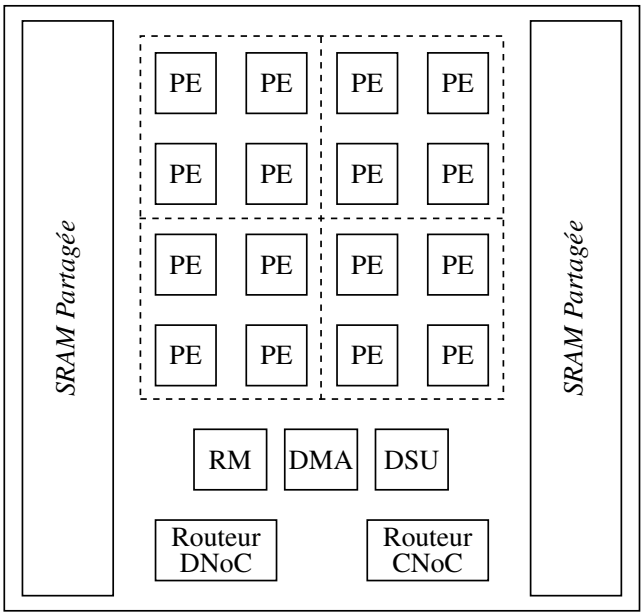


Figure A.3: Architecture d'un cluster de calcul du KALRAY MPPA®-256

- une unité de débogage, la *Debug and Support Unit* (ou *DSU*) qui facilite le développement et permet notamment d'utiliser le JTAG.
- un contrôleur *DMA* (pour *Direct Memory Access*) qui permet l'envoi de données au travers du NoC.
- 2MiB de SRAM partagée qui est organisée en 16 bancs indépendants.
- 1 interface d'accès à chacun des deux NoCs.

Tous les cœurs du KALRAY MPPA®-256 sont conçus selon la même architecture (*k1b* sur Bostan) laquelle repose sur un pipeline à 7 étages exécutant des bundles de 5 instructions en *VLIW* (pour *Very Long Instruction Word*). La fréquence d'horloge des cœurs peut être élevée jusqu'à 600MHz. Chaque cœur peut effectuer des opérations arithmétiques sur des entiers en 32 et 64 bits ainsi que des opérations de multiplication-sommation sur des flottants en double précision. Par ailleurs, deux modes d'exécution (*utilisateur* ou *privilégié*) sont disponibles pour supporter l'exécution d'un système d'exploitation. Chaque cœur dispose également d'une unité de gestion de mémoire (ou *MMU* en anglais) permettant une protection et une virtualisation de la mémoire paginée. Dans les clusters de calculs, tous les cœurs disposent de caches de données et d'instructions privés (8KiB chacun) pour rendre les accès à la mémoire SRAM locale plus efficaces.

La mémoire SRAM locale d'un cluster de calcul n'est accessible que par les éléments internes du cluster. La mémoire est découpée en 16 bancs de 128KiB chacun. Dans le cas où deux bancs sont adressés simultanément par deux masters, les chemins d'accès à chaque banc étant indépendants, aucune interférence ne se fera ressentir. En revanche, des accès concurrents au même banc subiront un arbitrage en Round-Robin hiérarchique comme montré sur la Figure A.4. En règle générale, la SRAM locale peut être utilisée pour permettre des communications internes au cluster par mémoire partagée. Cependant, il est important de noter qu'aucune forme de cohérence de cache n'est assurée par le matériel et qu'il revient donc au logiciel de gérer ce problème. Pour ce faire, il est possible soit d'utiliser des instructions spécifiques permettant explicitement de purger

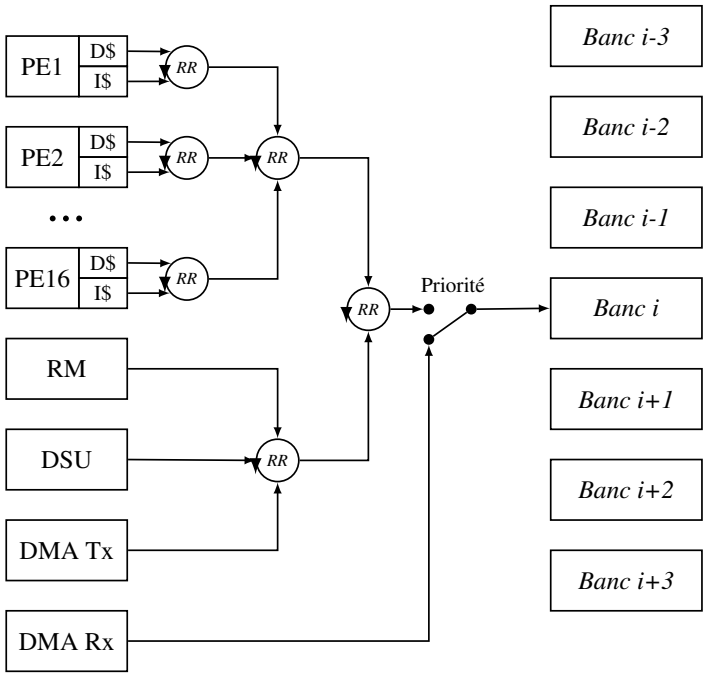


Figure A.4: Politique d'arbitrage vers la SRAM locale sur le KALRAY MPPA®-256

tout ou partie du cache d'un cœur, soit d'utiliser les MMUs des cœurs pour forcer des politiques de cachage spécifiques (*Write-Through* ou *Bypass* notamment) sur certaines pages mémoire. Cependant, il existe également un second moyen de communication entre les cœurs d'un même cluster. En effet, comme indiqué sur la Figure A.3, les PEs sont groupés par paquets de quatre au sein de chaque cluster de calcul. À l'intérieur de chaque groupe, les PEs peuvent émettre et recevoir des événements vers et depuis leurs voisins. Cela permet notamment l'implémentation de barrières de synchronisation efficaces en évitant le polling sur la mémoire SRAM locale. Par ailleurs, les PEs au sein d'un groupe disposent de facilités d'écriture dans certains registres des PEs voisins, permettant ainsi des échanges de données sans utiliser de zone de mémoire partagée.

A.2.2.2 Entre les clusters

Les communications entre les clusters se font par le passage explicite de messages ou par tirs DMA distants au travers d'un NoC. Les contrôleurs DMA se trouvant dans chaque cluster permettent de décharger les cœurs du travail d'émission et de réception des paquets NoCs. Chaque DMA dispose ainsi de 8 canaux d'émission qui peuvent être configurés indépendamment. En particulier, le choix de la route à suivre pour les paquets ainsi que les paramètres de régulation à l'injection sont spécifiques à chaque canal d'émission. Par défaut, seul le RM a la capacité d'écrire dans les registres de configuration du DMA. Les PEs peuvent être autorisés à faire de même par configuration. Le contrôleur DMA comporte un micro-moteur capable d'exécuter simultanément jusqu'à 8 threads. Chaque thread exécute des instructions issues d'un binaire résultant de la compilation d'un micro-code écrit dans un langage d'assemblage léger et spécifique à Kalray. En réception, chaque DMA dispose de 256 canaux qui peuvent, eux aussi, être configurés indépendamment. Ainsi, chaque canal de réception peut être associé à une zone de mémoire locale de taille finie dans laquelle seront écrits les messages entrant sur ce canal. Par ailleurs, les canaux de réception peuvent être configurés pour envoyer un événement au RM dans certaines situations telles que la réception

d'un paquet notifiant la fin d'une transmission ou lorsque la quantité de données reçues dépasse un certain seuil.

Les deux NoCs disponibles sur le KALRAY MPPA®-256 comportent la même topologie en tore 2D comme représenté sur la Figure A.2. Le *DNoC* (ou *Data-NoC*) est dédié à l'envoi des données volumineuses par l'intermédiaire du DMA. Le *CNoC* (ou *Control-NoC*) est quant à lui dédié à l'envoi de messages de contrôle courts qui peuvent notamment être utilisés pour effectuer des synchronisations inter-clusters. Les 2 NoCs suivent une politique de routage en wormhole avec un choix des routes à la source.

A.2.2.3 Adéquation aux contraintes avioniques

Le choix du KALRAY MPPA®-256 comme cible privilégiée pour l'application de ce travail de thèse est motivé par plusieurs raisons aussi bien techniques que non-techniques. Tout d'abord, la faible dissipation thermique du KALRAY MPPA®-256 est un premier atout majeur. Une faible consommation énergétique permet non seulement de réduire la consommation de carburant de l'avion mais surtout de faciliter la conception des cartes électroniques embarquées et de simplifier leur certification. En effet, il semble que, sous certaines conditions d'utilisation, le KALRAY MPPA®-256 dissipe une puissance suffisamment faible pour permettre de le refroidir passivement et par conséquent d'éviter le coût additionnel lié à la certification d'une solution de refroidissement active.

Deuxièmement, les propriétés temporelles du KALRAY MPPA®-256 semblent être particulièrement adaptées à son utilisation dans un contexte temps réel dur. Il a notamment été démontré dans le cadre du projet CERTAINTY [169] que les cœurs de calcul du KALRAY MPPA®-256 exhibent une propriété de *full timing compositionality*. En conséquence, le calcul des temps d'exécution pire-cas des programmes est simplifié et peut être effectué par la composition de plusieurs sous-analyses évitant ainsi de devoir effectuer une seule analyse coûteuse du système complet.

Troisièmement, il apparaît que très peu d'opérations sont effectuées de manière implicite sur le KALRAY MPPA®-256. L'absence de cohérence de cache matérielle et la capacité à envoyer des messages au travers du NoC par la programmation du DMA qui lui-même exécute un code modifiable en sont de bons exemples. De manière générale, la programmabilité de la plateforme est excellente et la capacité de contrôle par le logiciel est importante. Même si cette spécificité ne simplifie clairement pas la programmation haut niveau, cela apparaît comme un véritable atout dans un contexte avionique où la capacité à maîtriser de la plateforme est un enjeu particulièrement critique.

Enfin, KALRAY en tant qu'entreprise s'est montré particulièrement enclin à communiquer à ses clients des informations précises et détaillées sur leur plateforme. Dans le cas où le calcul des temps d'exécution pire-cas serait effectué par analyse statique, des modélisations fines du matériel seront nécessaires. Par conséquent, être capable d'obtenir les informations adéquates pour la construction de ces modèles s'avère être particulièrement important d'un point de vue industriel.

A.2.3 Synthèse

Les entrées de notre travail sont de deux natures. Nous avons d'une part un modèle d'application sous la forme d'un ensemble de tâches parallélisables et d'autre part une plateforme d'exécution puissante bien que complexe avec le KALRAY MPPA®-256. Notre objectif sera ici de définir le moyen d'exploiter la puissance de calcul parallèle du KALRAY MPPA®-256 pour permettre non seulement l'exécution prédictible de plusieurs applications temporellement isolées les unes des autres mais également pour supporter l'augmentation des besoins en puissance de calcul des systèmes avioniques par la distribution d'applications sur plusieurs clusters du processeur. En outre,

nous prendrons en compte les contraintes inhérentes au contexte industriel telles que la nécessité d'avoir des outils fonctionnant en temps raisonnable ou le besoin de solution reposant majoritairement sur le pré-calcul pour limiter le coût de certification d'un code volumineux qui prendrait les décisions en-ligne.

A.3 État de l'art

Deux problématiques majeures se posent pour ce travail. Tout d'abord, considérant un composant sur étagère non-conçu spécifiquement pour le temps réel, il sera nécessaire d'adopter une stratégie logicielle assurant la maîtrise de plateforme matérielle afin de garantir des exécutions prédictibles. Et deuxièmement, au vue de la complexité des applications industrielles visées, il semble nécessaire de permettre le placement et l'ordonnancement d'applications parallèles de façon automatisée et efficace. Nous détaillons les travaux existant sur ces deux sujets dans cette section.

A.3.1 Maîtrise du matériel

La problématique de la maîtrise de processeurs sur étagère à des fins d'exécution temps réel n'est pas nouvelle. Essentiellement deux types de stratégies logicielles ont été proposées dans ce cadre. D'abord, il y a les approches uniquement basées sur l'utilisation d'un logiciel de contrôle. Dans ce cas, les applications sont exécutées sans les modifier aucunement et seule une couche basse de logiciel garantit un accès ordonné à des ressources sensibles. Dans cette idée, Fisher [99] a proposé le premier logiciel de contrôle permettant la certification d'applications au niveau SIL 4 des régulations ferroviaires sur un processeur multi-cœurs. Dans son approche, les tâches doivent être classifiées comme critiques ou non-critiques. Ensuite, le temps est découpé en slots successifs. Certains slots sont marqués comme critiques et les autres comme non-critiques. L'idée est que, lors des slots non-critiques, tous les cœurs peuvent être utilisés simultanément pour l'exécution de tâches non-critiques pour lesquelles les contentions d'accès aux ressources partagées ne sont pas problématiques. Lors de slots critiques en revanche, seul un cœur est habilité à exécuter une tâche critique pendant que les autres cœurs sont maintenus inactifs. Ce faisant, la tâche critique s'exécutant seule ne subira aucunte interférence d'accès aux différentes ressources du processeur. Cela permet d'obtenir des garanties temporelles fermes sur l'exécution des tâches critiques mais peut aussi amener à des faibles utilisations des cœurs, et en particulier lorsque beaucoup de tâches critiques sont présentes.

Dans [101], Jean *et al.* ont proposé MARTHY, un hyperviseur permettant d'assurer un partitionnement robuste entre plusieurs applications concurrentes s'exécutant sur un processeur multi-cœurs. MARTHY est verrouillé dans les caches des cœurs et détecte l'utilisation d'instructions *sensibles*, c'est-à-dire qui impactent le trafic traversant l'interconnect ou bien l'état des autres cœurs. Lorsque une telle instruction est utilisée par une application, MARTHY reprend le contrôle du cœur et attend le début d'un créneau d'accès préalablement alloué à ce cœur pour exécuter l'instruction. Les créneaux d'accès alloués aux cœurs n'ayant pas de recouvrement temporel, les conflits d'accès aux ressources partagées sont éliminés. Ainsi, MARTHY permet d'apporter des garanties temps réel dur à des applications tout en permettant l'exécution simultanée de plusieurs tâches critiques sur des cœurs différents (tant qu'elles n'accèdent pas aux ressources partagées). Cette approche résout (au moins partiellement) les problèmes identifiés pour l'approche précédente mais il apparaît que certaines applications souffrent d'importantes pertes de performance avec MARTHY [102]. En effet, ces applications peuvent être utilisées sans modification mais peuvent subir des latences d'accès importantes aux ressources partagées, et en particulier lorsqu'elles sont mal alignées tem-

porellement avec le slot d'accès de leur cœur.

Dans [103], Yun *et al.* ont proposé Memguard, un système de régulation fournissant une bande passante mémoire garantie aux applications s'exécutant sur une cible multi-cœurs. Pour cela, chaque cœur est associé à un budget qui contient un nombre maximal d'accès au bus pour une fenêtre temporelle donnée. Les accès de chaque cœur sont comptés par des compteurs matériels évaluant le nombre de miss dans le cache de dernier niveau. Lorsque le budget est atteint, une interruption est générée et le cœur est stoppé jusqu'à la fin de la fenêtre en cours. Ce faisant, en allouant des budgets aux cœurs dont la somme est inférieure à la bande passante de la mémoire, il est possible de fournir des garanties qui faciliteront par la suite l'évaluation des WCETs. Toutefois, Memguard intègre un mécanisme de redistribution de la bande passante non-utilisée aux applications *best-effort* qui effectue des spéculations sur l'utilisation de la mémoire par les cœurs afin de réallouer la bande passante. Malheureusement, cette spéculation peut parfois être optimiste et ainsi fragiliser les garanties apportées aux applications temps réel.

D'une manière générale, les trois travaux mentionnés précédemment reposent uniquement sur l'utilisation d'un logiciel de contrôle et ne requièrent pas de modifications des applications. D'autres travaux sont basés sur une idée différente où la modification des applications ferait partie de la stratégie de contrôle. L'objectif est généralement d'exploiter plus efficacement le matériel afin d'obtenir des performances accrues. Pellizzoni *et al.* ont présenté *PREM* (pour *Predictable Execution Model*) [105] en suivant cette philosophie. Avec PREM, il est demandé aux développeurs d'applications d'annoter leur code pour mentionner les sections *predictibles*. Ainsi, les sections prédictibles sont découpées par le compilateur en phases mémoire et en phases d'exécution. L'idée est qu'une phase mémoire correspond à une étape de chargement de code et de données en cache et que les phases d'exécution utilisent le code et les données préalablement chargées sans jamais faire de miss en cache. Les phases mémoire des tâches sont ordonnancées de façon à éviter les conflits d'accès aux ressources et les phases d'exécution sont ordonnancées sur les cœurs avec la garantie qu'elles se feront sans interruptions liées à des ressources externes.

Les concepts introduits dans PREM on été étendus par Durrieu *et al.* dans [107]. Leur objectif est de permettre le portage d'une application industrielle de Thales sur un multi-cœurs COTS [108]. Les auteurs supposent un modèle de tâche *Acquisition, Exécution, Restitution* (ou *AER*) ou les exécutions et communications sont découplées. Les caches L2 du processeur sont configurés en SRAM afin de permettre lors des phases d'acquisition le chargement du code et des données requises pour la phase d'exécution suivante. Celle-ci se poursuit alors en isolation sur un cœur sans accéder aux ressources partagées du processeur. Enfin, les données produites sont propagées en mémoire DDR-SDRAM externe lors des phases de restitution. Avec AER, les phases d'acquisition et de restitution sont ordonnancées sans recouvrement temporel pour améliorer la prédictibilité d'accès aux ressources partagées et les exécutions concurrentes sont acceptées.

D'autres travaux[109, 111] sont basés sur ces concepts similaires afin de garantir des exécutions prédictibles sur des processeurs mono- ou multi-cœurs. Cependant, que ce soit avec ou sans modification d'application, la maîtrise d'un processeur *pluri*-cœurs COTS avec une stratégie de contrôle purement logicielle est un domaine qui reste aujourd'hui relativement peu exploré. La résolution de ce problème est le premier objectif de notre travail.

A.3.2 Placement et ordonnancement d'applications

Le placement et l'ordonnancement d'applications sur des plateformes pluri-cœurs est un sujet de recherche en plein essor. Parmi les travaux visant spécifiquement le KALRAY MPPA®-256, Giannopoulou *et al.* [47] ont proposé l'application d'une politique d'ordonnancement appelée *Flex-*

ible Time-Triggered Scheduling (*FTTS*) initialement présentée dans [145]. Un ordonnancement FTTS est composé d'une succession de *frames* d'une durée préalablement figée. Cette séquence de frames est répétée indéfiniment sur un cycle d'ordonnancement dont la durée est appelée hyper-période. Chaque frame est découpée en sous-frames de durées variables. Les tâches sont exécutées dans les sous-frames. La stratégie proposée repose sur l'allocation des tâches de haute criticité aux sous-frames qui débutent les frames. Inversement, les tâches peu critiques sont exécutées en fin de frames. Ainsi, les tâches hautement critiques sont exécutées sans encombre et les peu critiques le sont (en mode nominal, en mode dégradé ou ne le sont pas) si le temps restant dans la frame le permet. Avec FTTS, une exécution sûre des tâches les plus critiques est garantie et une exécution opportuniste des tâches peu critique est permise. Toutefois, il est nécessaire que les WCETs des tâches restent tous dans le même ordre de grandeur pour minimiser les temps d'attente dans les sous-frames. En plus du travail d'ordonnancement, les auteurs ont effectué une analyse conjointe des interférences d'accès à la SRAM locale des clusters de calcul et des pires temps de traversée des paquets NoC en utilisant le Calcul Réseau [58, 59]. Malheureusement, ces travaux n'ont, à notre connaissance, pas été implémentés sur cible réelle et n'ont pas été évalués expérimentalement. En particulier, la complexité du logiciel de contrôle supportant l'exécution des frames et sous-frames n'est pas décrite.

Dans [146], Becker *et al.* ont proposé une méthode d'ordonnancement au sein d'un seul cluster du KALRAY MPPA®-256. Ils supposent un modèle de tâche suivant la philosophie d'AER. Chaque tâche est composée de 3 phases. Ainsi, le WCET d'une tâche est décomposé en $C_i = C_i^{rd} + C_i^{ex} + C_i^{wr}$ avec:

- une phase de lecture de durée C_i^{rd} pendant laquelle le code et les données de la tâche considérée sont lus depuis la DDR-SDRAM externe, envoyés au travers du NoC et écrits en SRAM locale du cluster;
- une phase d'exécution de durée C_i^{ex} pendant laquelle le code préalablement chargé est exécuté par un PE;
- une phase d'écriture de durée C_i^{wr} pendant laquelle les données produites lors de la phase d'exécution sont propagées en DDR-SDRAM par un tir DMA distant.

Sur cette base, les auteurs proposent deux méthodes d'ordonnancement, la première optimale basée sur la programmation linéaire en nombre entiers (ou *ILP* en anglais), et la deuxième basée sur une heuristique permettant le passage à l'échelle sur de grandes applications. Ces deux méthodes assignent les phases d'exécution aux PEs et ordonnent les communications NoC pour éviter les conflits lors des phases de lecture et d'écriture. En outre, les auteurs proposent de réserver un banc de SRAM locale pour le stockage des données échangées par plusieurs tâches. De ce fait, le schéma de communication utilisé est implicite et restreint l'approche au cas des applications mono-cluster uniquement. De plus, les durées C_i^{rd} et C_i^{wr} semblent être calculées sans considérer de conflit d'accès à la DDR-SDRAM, ce qui implique en l'état qu'au plus 2 applications peuvent s'exécuter en parallèle (avec une application par contrôleur DDR-SDRAM).

Des méthodes comparables ont également été appliquées au placement d'applications sur des cibles pluri-cœurs différentes du KALRAY MPPA®-256. Carle *et al.* [147] ont présenté *LoPhT*, un outil de placement statique d'application dataflow sur des architectures pluri-cœurs. Les auteurs présentent plusieurs heuristiques calculant un ordonnancement global complètement dirigé par le temps des cœurs et des liens de communication. Bien que fournissant des résultats prometteurs, cette approche n'est pas complètement satisfaisante avec les contraintes que nous considérons.

Notamment, le cas d'applications multi-périodique n'est pas traité, la gestion de la DDR-SDRAM n'est pas abordée et la faisabilité de l'exécution d'un tel ordonnancement doit encore être vérifiée expérimentalement.

Puffitsch *et al.* ont présenté dans [148] une méthode de placement de tâches dépendantes sur des cibles pluri-cœurs. Ils énoncent un *modèle d'exécution* de haut niveau permettant l'exécution prédictible d'applications. Celui-ci est instancié sur 3 cibles matérielles COTS: l'Intel SCC [14], le Texas Instrument TMS320C6678 [108] et le Mellanox TILEmpower-Gx36 [149]. De plus, les auteurs présentent une technique de placement fondée sur la programmation par contraintes et qui passe à l'échelle d'application "*raisonnablement grandes*". Les communications au travers du NoC reposent sur la notion de *Message Passing Areas* (ou MPA) inspirées des *Message Passing Buffers* du SCC. Les lectures et écritures distantes dans les MPAs sont supposées être faites en temps borné (qui est obtenu dans l'article par mesure avec des benchmarks). Le passage à l'échelle est démontré par l'utilisation d'un cas d'étude comprenant plusieurs centaines de tâches. Malheureusement, cette approche répond à la majorité des contraintes imposées dans notre situation, mais semble peu adaptée à un portage sur le KALRAY MPPA®-256. En effet, les lectures distantes proposées dans le modèle d'exécution requerraient un important support logiciel et impliqueraient probablement de fortes latences qui devraient être prises en compte lors du calcul de placement. Par ailleurs, la prise en compte du conflit NoC maximum semble inadaptée au KALRAY MPPA®-256 où le routage wormhole implique qu'une utilisation chaotique pourrait potentiellement entraîner un interblocage (sauf s'il était évité avec la technique de [57] mais cela correspondrait alors à une utilisation de la puce non-conforme au modèle d'exécution initial).

Bien que les travaux présentés dans cette section apportent des résultats encourageants pour l'utilisation de processeurs pluri-cœurs dans un contexte temps réel dur, il semble qu'aucun d'entre eux ne considère simultanément un modèle d'application proche du notre et les contraintes avioniques et industrielles auxquelles nous nous soumettons. La résolution de ce problème est la deuxième motivation principale du travail présenté dans ce document.

A.4 Modèle d'exécution sur le KALRAY MPPA®-256

Dans cette section, nous identifions certaines des ressources du KALRAY MPPA®-256 comme étant les points de contention majeurs créant de la variabilité dans les temps exécution des programmes en cas d'accès concurrents. Sur cette base, nous proposons ensuite un *modèle d'exécution* permettant de réduire ou d'éliminer les interférences d'accès à ces ressources.

A.4.1 Identification des points interférences

La Figure A.5 décrit la procédure d'accès à la mémoire DDR-SDRAM par un PE sur le KALRAY MPPA®-256. Au cours de cette procédure, nous pouvons identifier des sources d'interférences potentielles à plusieurs niveaux avec des applications s'exécutant sur les autres PEs. Dans le cas du processus d'écriture en DDR-SDRAM, nous prenons l'exemple d'une application s'exécutant sur le cœur 3 du cluster de calcul B et produisant une donnée qui doit être écrite dans le banc 4 de la DDR-SDRAM externe. La procédure se déroule comme suit:

1. le cœur 3 écrit la donnée produite dans un banc de mémoire locale SRAM. Lors de cette étape, d'autres éléments du cluster (autres PEs, DMA, ...) peuvent potentiellement accéder au même banc et par conséquent interférer avec l'exécution de l'application du cœur 3;

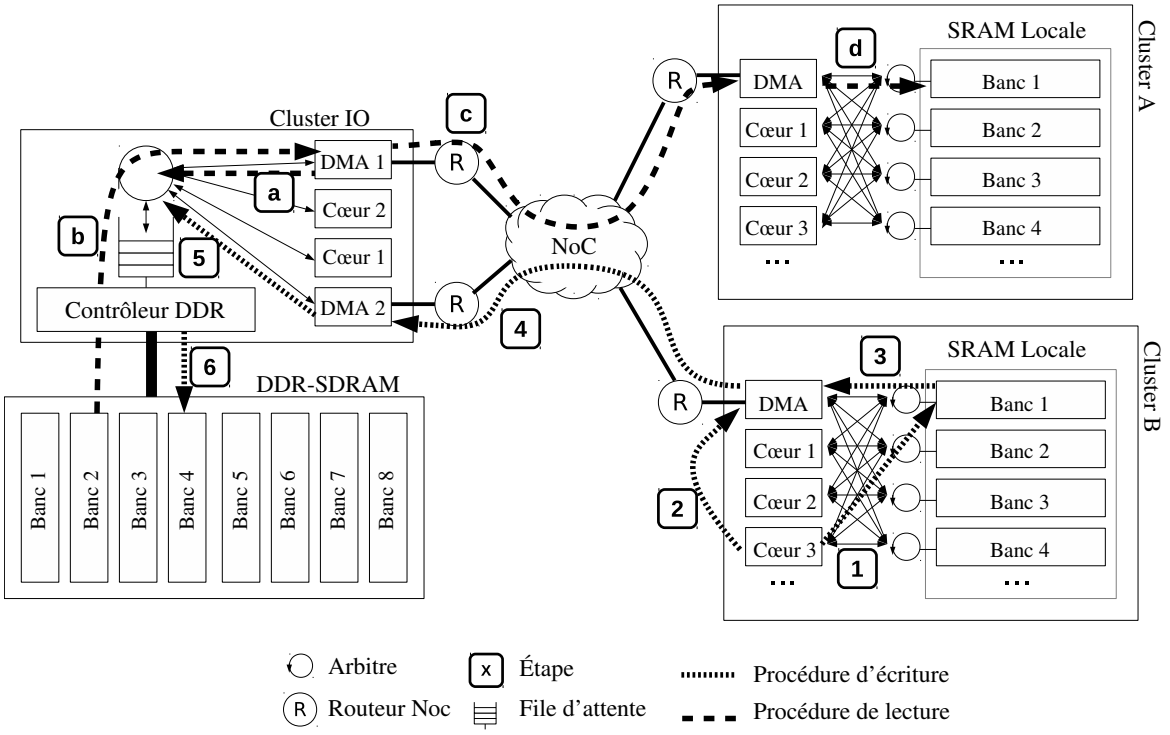


Figure A.5: Exemple d'accès mémoires distants sur le KALRAY MPPA®-256

2. le cœur notifie le DMA afin d'envoyer la donnée au travers du réseau;
3. le DMA lit la donnée à envoyer depuis la mémoire locale et subit potentiellement des interférences à ce niveau (avec une priorité différente de l'étape 1 en raison de l'arbitrage hiérarchique appliqué aux bancs de SRAM locale comme décrit sur la Figure A.4);
4. au cours de la traversée du NoC à destination du cluster IO, le paquet contenant la donnée peut se trouver en situation de conflit avec d'autres paquets envoyés par d'autres applications et ainsi être retardé. De ce fait, le temps de traversée du NoC pour une application est corrélé à l'utilisation qui en faite par d'autres applications;
5. à la réception du paquet NoC, un DMA du cluster IO envoie une requête d'écriture vers le contrôleur DDR-SRAM. Celui-ci effectue alors un arbitrage entre les différentes requêtes qui lui ont été soumises, incluant donc les requêtes émises par d'autres applications;
6. enfin, une fois notre requête élue par l'arbitre (et qui peut avoir été retardée d'un temps très variable en fonction de l'utilisation qui est faite de la DDR-SDRAM par les concurrents), l'écriture dans la zone mémoire adéquate est effectuée.

La procédure de lecture depuis la DDR-SDRAM s'effectue de manière relativement symétrique à celle d'écriture, à la différence qu'un RM sur un cluster IO doit être à l'initiative. Celui-ci peut lancer la procédure soit en réponse à une requête préalablement envoyée depuis un cluster de calcul soit en suivant les directives qui lui ont été imposées au travers d'une table d'ordonnancement.

En prenant l'exemple des procédures d'accès à la DDR-SDRAM, nous avons pu identifier les trois niveaux de ressources impliquant une corrélation entre les temps d'exécution d'applications

concurrentes. Pour éliminer cette corrélation et permettre l'exécution des applications indépendamment les unes des autres, nous proposons de contraindre l'accès à ces trois ressources (la SRAM locale, le NoC et la DDR-SDRAM) par le biais d'un modèle d'exécution que nous détaillons et motivons dans la section suivante.

A.4.2 Réduction des interférences

Afin d'éliminer les interférences non-désirées entre applications concurrentes, nous proposons d'utiliser un modèle d'exécution contraignant l'accès aux bancs de mémoire SRAM, au NoC et à la DDR-SDRAM. Ce modèle d'exécution est composé de 4 règles qui, si elles sont respectées (et nous verrons par la suite comment s'en assurer), permettent de partitionner le KALRAY MPPA®-256 en plusieurs environnements d'exécution indépendants les uns des autres. Nous définissons ainsi la notion de *partition* comme suit:

Définition 1 (Partition)**.** *Une partition est un environnement d'exécution au sein duquel le comportement temporel d'une application ne dépend pas du comportement des applications appartenant à d'autres partitions.*

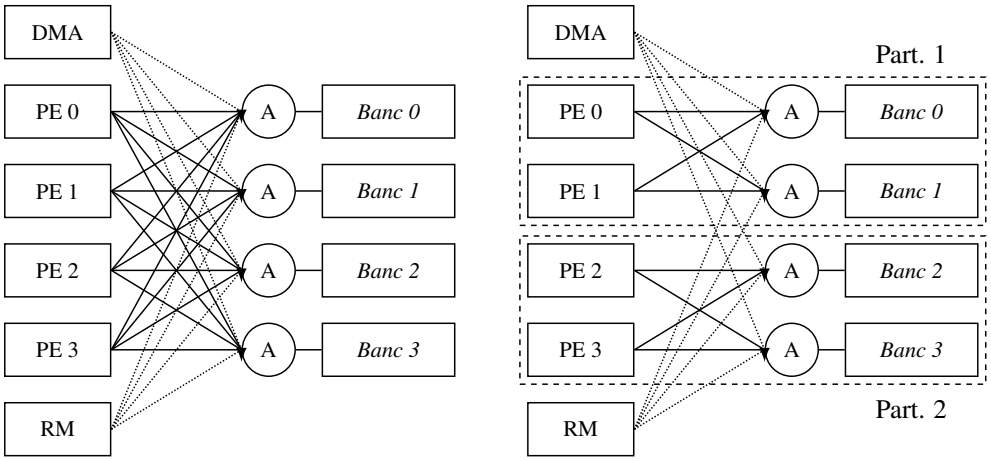
Notre modèle d'exécution impose aux applications d'accéder aux ressources partagées tout en respectant 4 règles.

Règle 1. *Tout PE (et respectivement tout banc de SRAM locale) ne peut pas être réservé par plus d'une partition.*

Grâce à la règle 1, nous assurons qu'un PE accèdant à un banc de SRAM locale ne subira pas d'interférences causées par des PEs d'autres partitions. Ainsi, les seules interférences à prendre en compte lors de l'accès à la SRAM sont celles provenant du trafic *ami*, c'est-à-dire celui des autres PEs alloués à la partition. De ce fait, le calcul des WCETs peut non seulement être effectué de manière plus efficace en évitant de prendre en compte les pénalités d'interférence maximale (c'est-à-dire lorsque tous les PEs accèdent simultanément au même banc) mais cela permet également d'apporter une isolation temporelle entre les partitions. Le trafic initié par le RM et/ou le DMA peut toujours entrer en conflit avec celui des PEs d'une partition. Dans notre approche, nous prendrons l'hypothèse (que nous vérifierons par la suite) que ce trafic du RM ou du DMA dans un banc alloué à une partition n'est lié qu'au travail effectué dans cette partition et ne viole donc pas l'isolation temporelle. La Figure A.6 représente un exemple d'application de cette règle sur un cluster de calcul simplifié et partagé par deux partitions. Ici, le choix de découpage a été fait équitablement (2 cœurs et 2 bancs par partition) mais n'importe quel autre découpage aurait été possible (3 cœurs et 1 banc pour l'une, 1 cœur et 3 bancs pour l'autre, ...).

Règle 2. *Les communications sur le NoC doivent respecter un ordonnancement global dirigé par le temps et éviter les conflits en utilisant soit des routes différentes soit en accédant aux ressources à des instants différents. L'ordonnancement du NoC doit garantir que, lorsque une route est utilisée par une communication, elle lui est complètement réservée pour sa durée d'utilisation.*

Un ordonnancement dirigé par le temps permet de largement simplifier le calcul des temps de traversée maximaux des paquets envoyés. En effet, en ne considérant aucun conflit d'accès aux ressources NoC, les pires temps de traversée peuvent être déduits efficacement de modèles matériels simples. En revanche, ce bénéfice n'est possible qu'au détriment de la flexibilité de l'approche et au coût d'efforts supplémentaires lors du pré-calcul de l'ordonnancement pour éviter les conflits par construction. Les travaux précédents visant à rendre le NoC du KALRAY MPPA®-256



(a) Sans la règle 1 (b) Avec la règle 1

Figure A.6: Exemple d'application de la règle 1 du modèle d'exécution sur une architecture simplifiée de cluster de calcul avec 2 partitions

prédictible utilisent généralement une approche asynchrone en utilisant les régulateurs d'injection matériels des interfaces NoC. Dans [46] ou [47], les auteurs configurent par exemple ces régulateurs d'injection en appliquant la théorie du Calcul Réseau [58, 59]. Si ces approches sont généralement reconnues comme apportant une certaine flexibilité et des garanties sûres et peu pessimistes sur les *bandes passantes* allouées aux applications, notre choix d'un ordonnancement global dirigé par le temps s'explique par deux raisons:

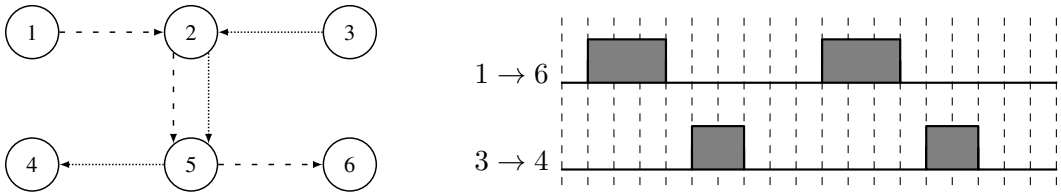
1. Notre objectif principal est d'apporter une garantie d'isolation temporelle stricte entre les partitions. Les approches asynchrones utilisant le calcul réseau permettent de borner de manière sûre les interférences sur le NoC mais n'évitent pas les conflits. Ainsi, le temps de traversée d'un paquet dans le cas asynchrone est, même si cela peut être borné, dépendant de l'utilisation qui est faite du NoC par les partitions concurrentes. Pour obtenir une isolation temporelle stricte, l'évitement complet des conflits par un ordonnancement global apparaît alors plus adapté.

2. Un des objectifs sous-jacents à notre approche est de permettre l'exploitation efficace du KALRAY MPPA®-256 avec des applications distribuées sur plusieurs clusters. Ainsi, notre capacité à garantir des *latences* courtes pour l'échange de messages entre les clusters est un point particulièrement important. Il a été montré dans [152] qu'un ordonnancement global dirigé par le temps permet de satisfaire ce type de contraintes là où des approches asynchrones basées sur le calcul réseau permettent plutôt d'obtenir des garanties efficaces sur les *bandes passantes*.

La Figure A.7 représente un exemple d'application de la règle 2 du modèle d'exécution sur un NoC simplifié pour lequel une ressource est partagée par deux communications. Le conflit d'accès à cette ressource est alors résolu par l'allocation de créneaux d'accès périodiques à chaque communication en évitant leur recouvrement temporel.

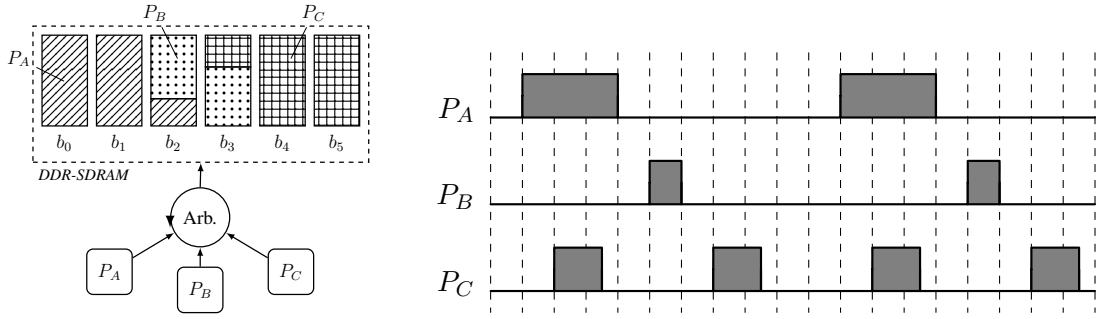
Règle 3. *Toutes les zones mémoires qu'une application souhaite envoyer au travers du NoC doivent être intégralement définies hors-ligne.*

L'application de la règle 3 n'est pas absolument nécessaire pour garantir une isolation temporelle entre partitions. Cependant, elle apporte plusieurs propriétés intéressantes pour l'implémentation.



(a) NoC Simplifié (b) Ordonnancement des communications

Figure A.7: Exemple d'application de la règle 2 du modèle d'exécution avec 2 communications partageant un lien NoC ② → ⑤



(a) Accès à la DDR-SDRAM (b) Ordonnancement des accès à la DDR-SDRAM

Figure A.8: Exemple d'application de la règle 4 du modèle d'exécution sur un arbitre DDR-SDRAM simplifié et 3 masters P_A, P_B et P_C en concurrence pour accéder à la mémoire

En connaissant à l'avance les zones mémoire qui seront envoyées au travers des créneaux alloués aux communications, il devient possible de vérifier hors-ligne que la taille de ces créneaux est suffisante pour permettre l'envoi des données. En particulier, il possible de vérifier en utilisant des modèles du matériel relativement simples (sachant que l'on prend l'hypothèse que les conflits sont évités par construction) que le temps alloué à l'envoi d'une donnée est suffisant pour permettre sa traversée complète du NoC. Par ailleurs, cette règle simplifie considérablement l'implémentation de notre *hyperviseur* et permet notamment de réduire son WCET qui, comme nous le montrerons par la suite, est un critère majeur pour obtenir de bonnes performances. Enfin, connaître à priori les zones mémoires visées par le DMA (notamment en réception) est un atout pour calculer efficacement les pénalités d'interférence subies par les PEs en accédant à leurs bancs de SRAM locale.

Règle 4. *Un banc de DDR-SDRAM externe peut être partagé par plusieurs partitions si et seulement si elles n'y accèdent jamais simultanément.*

L'application de la règle 4 permet d'éviter les conflits d'accès entre plusieurs masters accédant à différentes lignes d'un même banc de DDR-SDRAM. En effet, les lectures et écritures en DDR-SDRAM ne peuvent s'effectuer qu'après l'ouverture préalable par le contrôleur de la ligne adéquate dans le banc visé. Dans le cas où deux masters initieraient des transactions visant des zones différentes d'un même banc simultanément, il serait nécessaire pour borner la durée de ces transactions de considérer les fermetures de lignes intempestives dues aux accès concurrents. En évitant ce problème par construction, on peut tirer parti du parallélisme de la DDR-SDRAM banquée tout en garantissant une bonne prédictibilité. En revanche, cela implique une difficulté supplémentaire lors du pré-calcul de l'ordonnancement des accès à la mémoire.

La Figure A.8 représente un exemple d'application de la règle 4 dans une situation où 3 partitions (P_A, P_B et P_C) accèdent à une DDR-SDRAM qui comporte des bancs partagés. Dans ce cas, les accès issus des partitions en situation de partage sont ordonnancés de façon à éviter le recouvrement temporel, et donc le cas où deux partitions accèderaient à des zones différentes d'un banc en même temps. En outre, les partitions ne partageant pas de bancs (P_A et P_C dans cet exemple) sont quant à elles autorisées à accéder simultanément à la DDR-SDRAM.

A.4.3 Synthèse

Dans cette section, nous avons identifié la SRAM locale, le NoC et la DDR-SDRAM comme les trois ressources impliquant une corrélation entre les temps d'exécution des programmes concurrents non contrôlés sur le KALRAY MPPA®-256. Dans un contexte où nous voudrions exécuter des applications dans des environnements d'exécution temporellement isolés les uns des autres, nous avons défini une notion de partition qui capture la problématique de l'isolation temporelle. Sur cette base, nous avons défini un modèle d'exécution permettant d'éliminer les interférences subies par les applications lorsque elles accèdent aux ressources partagées. Ce modèle d'exécution repose sur un partitionnement purement spatial au sein des clusters de calcul et sur un partitionnement spatialo-temporel pour le NoC et la DDR-SDRAM. De manière générale, notre approche s'appuie sur une pré-réservation des ressources matérielles du processeur. Dans la section suivante, nous formaliserons une notion de *budget de partition* qui représente cette pré-réservation et nous montrerons comment elle peut être utilisée dans le cadre d'un processus d'intégration automatisé.

A.5 Atelier d'intégration

Dans cette section, nous présentons un atelier permettant l'intégration de plusieurs applications temporellement isolées les unes des autres sur le KALRAY MPPA®-256. Cet atelier repose principalement sur la propriété d'isolation temporelle apportée par notre modèle d'exécution. Comme cela est décrit sur la Figure A.9, plusieurs étapes sont nécessaires à l'intégration de partitions à partir de modèles hauts niveaux des applications jusqu'à une exécution concurrente sur la cible de façon prédictible. D'une manière générale, l'atelier repose sur un ensemble de calculs hors-ligne permettant la génération de tables d'ordonnancement qui seront appliquées en-ligne. Nous détaillons d'abord les entrées de cet atelier et formalisons notamment la notion de *budget de ressource* avant de décrire les objectifs des différentes étapes de pré-calcul et d'introduire le support logiciel nécessaire en-ligne garantissant le respect du modèle d'exécution.

A.5.1 Paramètres d'entrées

Notre atelier d'intégration prend tout d'abord en entrée un modèle de l'architecture du KALRAY MPPA®-256. Nous notons les éléments de ce modèle comme un quadruplet $< C_K, I_K, N_K, R_K >$ où:

- $C_K = \{c_K^0, \ldots, c_K^{15}\}$ représentent les 16 clusters de calcul.
- $I_K = \{i_K^0, \ldots, i_K^{15}\}$ représentent les 16 RMs des clusters I/O.
- $N_K = \{n_K^0, \ldots, n_K^{31}\}$ représentent les 32 nœuds du NoC.
- $R_K = \bigcup R_K^{ij}$ est l'ensemble des chemins NoC avec R_K^{ij} les chemins de n_K^i à n_K^j.

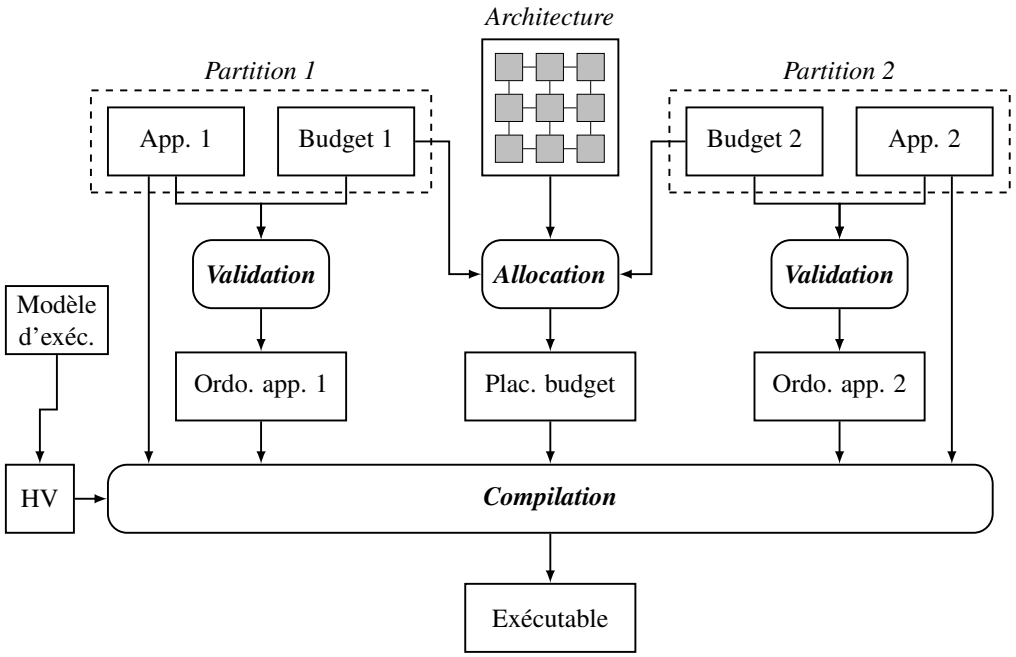


Figure A.9: Atelier d'intégration de partitions multiples sur le KALRAY MPPA®-256

L'autre type d'entrée de notre atelier est les partitions. Chaque application dans une partition est modélisée comme précédemment expliqué (Section A.2.1). En outre, chaque partition est associée à un *budget de ressources* qui représente la quantité de ressources matérielles dont l'application a besoin pour s'exécuter correctement. Plus précisément le budget d'une partition $\mathcal{P}_i$ s'écrit comme un quadruplet $< \mathcal{N}_i, \mathcal{I}_i, \mathcal{C}_i, \mathcal{B}_i >$ où:

1. $\mathcal{N}_i = \{pn_1^i, \ldots, pn_n^i\}$ est un ensemble fini de *Nœuds de Partition* (ou *PNs*). Un PN représente un besoin en puissance de calcul à l'intérieur d'un cluster de calcul pour une partition. Un PN représente la réservation d'un nombre de PEs et d'un nombre de bancs de SRAM locale pour une partition. Plus formellement:
   - $N_c : \mathcal{N}_i \mapsto \mathbb{N}$ associe chaque PN avec le nombre de PEs qui se doivent de lui être réservés dans un cluster de calcul.
   - $N_b : \mathcal{N}_i \mapsto \mathbb{N}$ associe chaque PN avec le nombre de bancs de SRAM locale qui se doivent de lui être réservés dans un cluster de calcul.
2. $\mathcal{I}_i = \{io_1^i, \ldots, io_o^i\}$ est un ensemble de *Nœuds I/O* (ou *ION*s). Un ION est un point d'accès à la DDR-SDRAM externe matérialisé par un RM de cluster I/O. Son objectif est de répondre aux requêtes d'écritures et de lectures en DDR-SDRAM distantes initiées par les PNs.
3. $\mathcal{C}_i = \{pc_1^i, \ldots, pc_p^i\}$ est un ensemble de *Communications de Partition* (ou *PC*). Une PC est un canal de communication directionnel permettant l'échange de données entre PNs ou entre PNs et IONs. Une PC représente un créneau d'accès périodique au NoC sur un chemin donné. Plus formellement:
   - $src : \mathcal{C}_i \mapsto \mathcal{N}_i \cup \mathcal{I}_i$ associe chaque PC avec son PN ou ION source.
   - $dst : \mathcal{C}_i \mapsto \mathcal{N}_i \cup \mathcal{I}_i$ associe chaque PC avec son PN ou ION destination.

- $T : \mathcal{C}_i \mapsto \mathbb{N}$ fournit la *période* d'activation du PC.
- $C : \mathcal{C}_i \mapsto \mathbb{N}$ fournit la *durée* du créneau d'accès au NoC d'un PC.
- $O : \mathcal{C}_i \mapsto \mathbb{N}$ fournit l'*offset* du PC, aussi appelé la première date d'activation.
- $R : \mathcal{C}_i \mapsto \mathbb{N}$ fournit le plus grand nombre de nœuds du NoC qui peuvent être traversés par le chemin alloué au PC. Cela peut être utilisé afin d'imposer une contrainte sur la latence NoC maximale acceptable pour le PC en considération.

4. $\mathcal{B}$ représente un nombre de bancs de DDR-SDRAM externe.

De façon générale, une partition inclut une application ainsi qu'un budget de ressources *abstraites* formalisées sous la notion de *budget*. De ce fait, les développeurs d'applications peuvent exprimer leurs besoins en ressources matérielles tôt dans le processus de développement. Avec une telle approche fondée sur la pré-réservation de ressources, il est possible de concevoir, de développer, de tester et de qualifier une application sans nécessiter d'avoir connaissance des partitions concurrentes. En revanche, cela implique d'apporter un support hors-ligne et en-ligne du modèle d'exécution. Ces deux aspects sont développés dans les sections suivantes.

A.5.2 Calculs hors-ligne

Comme cela est montré sur la Figure A.9, notre atelier de conception suppose deux étapes de calculs hors-ligne pour la *Validation* et l'*Allocation* des budgets. L'étape de Validation consiste à confirmer que le budget alloué à une application est effectivement suffisant pour qu'elle s'exécute de manière satisfaisante, c'est-à-dire en respectant l'ensemble de ses contraintes fonctionnelles et non-fonctionnelles. Plus précisément, nous recherchons lors de la Validation à déterminer un placement et un ordonnancement de l'application sur les ressources qui lui sont associées tout en respectant les contraintes applicatives telles que les précédences ou échéances des tâches et sous-tâches. L'automatisation de cette étape en utilisant la programmation par contraintes est l'objet de la Section A.7.

Lors de l'étape d'Allocation, les ressources *abstraites* représentées par les PNs, IONs et PCs sont allouées à des ressources *concrètes* de la cible matérielle. Par exemple, c'est lors de cette étape d'allocation que la décision de placer (ou non) deux PNs de deux partitions différentes sur le même cluster sera prise, compte tenu des ressources matérielles disponibles (16 PEs et 15 bancs de SRAM locale [1] dans chaque cluster) et des ressources matérielles requises pour les PNs. En l'état d'avancement actuel, nous disposons d'une version préliminaire de l'implémentation automatique de l'étape d'Allocation qui, en utilisant la programmation par contraintes, semble fournir des résultats intéressants. Il apparaît toutefois qu'une série d'analyses et d'évaluations plus poussées sont nécessaires pour que celle-ci puisse être considérée comme mature. Ainsi, les travaux développés dans ce document se focaliseront sur l'étape de Validation, laissant l'automatisation de l'étape d'Allocation comme une perspective intéressante de travaux futurs.

A.5.3 Support d'exécution en-ligne

Une fois les tâches, sous-tâches et données des applications placées et ordonnancées à l'intérieur de leurs budgets, et une fois les budgets des différentes partitions alloués à des ressources matérielles concrètes, il reste à garantir à l'exécution que les frontières entre partitions ne sont pas violées par les applications pour garantir leur isolation temporelle. Pour ce faire, nous proposons d'instancier

[1] Le dernier banc de SRAM locale sera réservé pour le système dans notre approche. Cela est détaillé en Section A.6.

notre modèle d'exécution au sein d'une couche basse de régulation logicielle appelée un *hyperviseur*. Celui-ci sera en charge de forcer le respect des règles du modèle d'exécution et l'isolation temporelle entre partitions en-ligne, y compris en cas de logiciel applicatif bogué ou malveillant. L'hyperviseur sera en charge de:

- forcer le partitionnement spatial entre les PNs partageant un cluster de calcul;
- gérer le NoC pour mettre en œuvre son ordonnancement global dirigé par le temps et ainsi garantir l'évitement des conflits;
- configurer les contrôleurs DMAs pour transférer les zones mémoires adéquates lors des activations des PCs;
- effectuer un certain nombre de configurations matérielles aussi bien au niveau des clusters de calcul que des clusters IO;
- et maintenir les applications dans les limites de leur budget, y compris lorsque leur comportement n'est pas nominal.

Nous détaillons dans la Section A.6 comment un hyperviseur remplissant l'ensemble de ces fonctions peut être implémenté sur le KALRAY MPPA®-256 et nous montrons notamment que la propriété attendue d'isolation temporelle entre partitions est effective en pratique.

A.6 Implémentation du modèle d'exécution

Dans cette section, nous présentons l'instanciation de notre modèle d'exécution au travers d'un hyperviseur s'exécutant sur le KALRAY MPPA®-256. Nous détaillons l'architecture de cet hyperviseur et les moyens mis en œuvre pour forcer le respect des 4 règles du modèle d'exécution. De plus, nous montrons par un cas d'étude académique que les propriétés temporelles attendues sont vérifiées expérimentalement.

A.6.1 Description de l'hyperviseur

Notre hyperviseur s'exécute en *bare-metal*[2] sur le KALRAY MPPA®-256 pour gérer l'ensemble des ressources matérielles à bas niveau. Le RM de chaque cluster exécute une instance de l'hyperviseur sous la forme d'une tâche logicielle périodique. Dans notre implantation actuelle, toutes les instances de l'hyperviseur s'exécutant sur tous les clusters sont simultanément activées sur tous les RMs. Chaque instance est configurée en fonction des PNs hébergés sur son cluster et est notamment en charge des procédures de démarrage des PEs, de la gestion de leurs routines d'interruption (ou *ISR* en anglais) et de la configuration des DMAs.

A.6.1.1 Hypothèses

Notre hyperviseur inclut non seulement une tâche périodique s'exécutant sur les RMs de tous les clusters mais également le code privilégié des PEs. Le bon fonctionnement de l'hyperviseur repose sur un certain nombre d'hypothèses qui, pour la majorité d'entre elles, sont liées à des configurations matérielles spécifiques.

[2] Au plus proche du matériel, sans système d'exploitation

En premier lieu, nous prenons l'hypothèse que l'adressage des mémoires SRAM locales est configuré au démarrage en mode non entrelacé afin de simplifier le mapping mémoire sur des bancs spécifiques.

En second lieu, nous considérons que les mécanismes matériels de régulation à l'injection sur le NoC sont désactivés. En effet, l'utilisation qui a été faite de ces limiteurs (ρ, σ) dans des travaux précédents [46, 47] est liée à un ordonnancement du NoC asynchrone pour lequel les garanties sur les temps de traversée des paquets sont fournies par Calcul Réseau. Dans notre approche où les conflits sont évités par construction, l'utilisation de ces techniques n'est pas nécessaire.

En troisième lieu, nous prenons pour hypothèse deux configurations spécifiques du contrôleur DDR-SDRAM. D'abord, comme pour les SRAM locales, l'accès à la DDR-SDRAM en adressage entrelacé est interdit pour permettre une allocation simplifiée des bancs mémoire aux partitions. Et deuxièmement, nous désactivons les mécanismes de réordonnancement des accès par le contrôleur afin de simplifier la modélisation de celui-ci. Il est d'ailleurs important de noter que, grâce à la règle 4, les conflits d'accès à la DDR-SDRAM sont résolus par construction et que le réordonnancement en-ligne par le contrôleur afin d'optimiser la performance mémoire moyenne devient alors partiellement inutile.

Enfin, nous prenons l'hypothèse que l'ensemble des instances de l'hyperviseur évoluent avec une notion de temps globale et partagée par tous pour garantir leur synchronie. Ceci est possible nativement avec la version Bostan du KALRAY MPPA®-256 qui intègre au sein de chaque cluster une DSU comportant un compteur qui est garanti d'être matériellement synchronisé avec tous ses homologues sur les autres clusters.

A.6.1.2 Respect des règles du modèle d'exécution

Notre hyperviseur a pour but de garantir en-ligne le respect des 4 règles de notre modèle d'exécution afin d'apporter aux partitions une propriété d'isolation temporelle. Cela est implémenté comme suit:

Règle 1 Afin de garantir le partitionnement spatial à l'intérieur des clusters de calcul, notre hyperviseur utilise plusieurs leviers. Tout d'abord, chaque PE dispose d'une MMU qui permet la virtualisation et la protection mémoire. Lors du démarrage des PEs, ces MMUs sont configurées et figées par notre hyperviseur afin de permettre à chaque PE de n'accéder qu'aux bancs de SRAM alloués à sa partition. De plus, le code utilisateur est limité au mode d'exécution non privilégié des PEs. De ce fait, tout accès d'un PE hors de sa zone mémoire autorisée ou tout autre évènement problématique (trap de division par zéro, appel système, ...) génèrera une exception qui sera traitée en mode privilégié par l'ISR de l'hyperviseur. Dans notre cas, un appel système est considéré comme un comportement anormal car notre paradigme d'utilisation des ressources externes au cluster est complètement passif d'un point de vue applicatif. C'est-à-dire que l'envoi ou la réception de données vers/depuis un cluster distant est orchestré par la règle 3 du modèle d'exécution et n'implique pas la participation active de l'application en-ligne.

Dans notre implantation actuelle, le traitement qui est fait de ces erreurs implique que le PE mis en cause soit éteint par mesure de sécurité. Il pourrait être intéressant de permettre un traitement plus doux de ces erreurs pour certaines catégories d'applications. Nous considérons cela comme une perspective intéressante d'amélioration de notre travail.

Règle 2 Afin de garantir que le NoC est utilisé au travers d'un ordonnancement global dirigé par le temps, chaque instance de l'hyperviseur est en charge de respecter une table d'ordonnancement

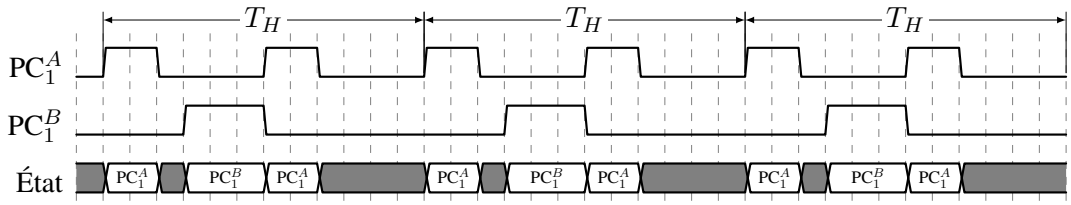


Figure A.10: Exemple d'application d'une table d'ordonnancement NoC

détaillant le travail de l'ensemble de ses PCs sortants. Chaque table est constituée d'une succession d'états de communication correspondant à une hyper-période complète $T_H = ppcm(T(pc))$ où $ppcm()$ renvoie le plus petit commun multiple des périodes des PCs. A chaque activation, l'hyperviseur identifie son état courant dans la table d'ordonnancement et initie des transferts DMAs lorsque cela est nécessaire. Le Tableau A.2 et la Figure A.10 montrent un exemple de table d'ordonnancement et l'application qui en est faite par l'hyperviseur pour 2 PCs PC_1^A et PC_1^B de périodes respectives 6 et 12.

Occupant (de l'interface NoC)	PC_1^A	*Attente*	PC_1^B	PC_1^A	*Attente*
Durée (en nb d'act. de l'hyperviseur)	2	1	3	2	4

Tableau A.2: Exemple de table d'ordonnancement pour PC_1^A et PC_1^B

L'envoi de données au travers du réseau s'effectue par la programmation et la configuration des DMAs de chaque cluster. Sur le KALRAY MPPA®-256, chaque contrôleur DMA exécute un micro-code qui peut être customisé. Dans notre cas, le micro-code des DMAs fait partie de l'hyperviseur et ne peut pas être modifié ou configuré directement par les applications. Ce micro-code est capable d'envoyer plusieurs zones de mémoires non contigües de manière autonome, c'est-à-dire sans nécessiter de travail de la part du RM. Le temps nécessaire à la configuration de celui-ci reste toutefois linéaire en fonction du nombre de zones non contigües à envoyer. Pour n zones mémoire, le temps de configuration imputé au RM croît en $\mathcal{O}(n)$. Pour masquer ce temps, nous utilisons 2 copies du même micro-code. Lorsque l'une s'exécute, le RM configure l'autre. Ainsi, il est possible de changer rapidement de micro-code au début de l'état suivant et de démarrer les transferts sans plus de délais.

Dans notre approche, la vérification du non dépassement des créneaux de communication des PCs est effectuée hors-ligne. Grâce à la règle 3, chaque zone mémoire à envoyer lors de chacune des activations des PCs est fixée hors-ligne. De plus, la règle 2 garantit que les conflits sur le NoC sont évités par construction. Ainsi, avec des modèles relativement simples (c'est-à-dire ne prenant pas les collisions de paquets en compte) il est possible de vérifier que l'ensemble des données à envoyer dans chaque créneau peut l'être dans leur temps imparti. Si tous ces transferts sont vérifiés, il ne reste alors à l'hyperviseur qu'à démarrer les transferts aux moments prévus pour garantir un bon ordonnancement du NoC.

Règle 3 La règle 3 impose que chaque zone mémoire à envoyer au cours de chaque activation des PCs soit connue hors-ligne. Ainsi, chaque PC est associé à une liste de listes de zones mémoire à envoyer. Avant l'activation d'un PC, une de ces listes est attachée à un micro-code un DMA pour permettre son traitement dès l'activation du PC. A l'activation suivante, la liste suivante est transmise. La Figure A.11 montre un exemple de représentation graphique de la liste de listes de zones mémoire associée à un PC. On peut voir qu'à la première activation

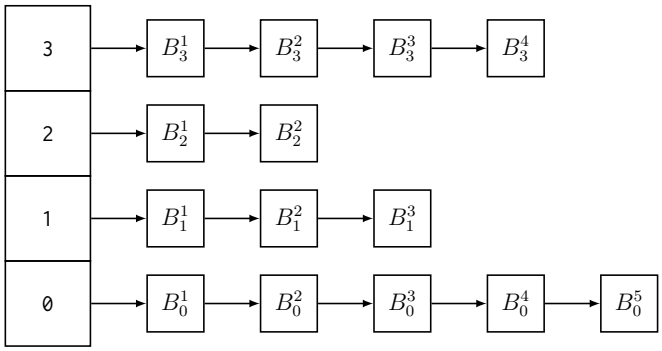


Figure A.11: Exemple de liste de liste de zones mémoires associées à un PC

du PC, les zones B_0^1 à B_O^5 sont envoyées. Lors de la deuxième activation, les zones B_1^1 à B_1^3 sont transmises et ainsi de suite.

Règle 4 La règle 4 impose que les accès à la DDR-SDRAM se fassent sans conflit au niveau des bancs. Ces accès étant rendus possibles aux partitions par l'utilisation de PCs, le calcul de l'ordonnancement du NoC doit s'effectuer avec une attention particulière portée aux PCs en provenance ou à destination des IONs. En calculant un ordonnancement tel que décrit en Section A.4.2, les problèmes de conflits d'accès à la DDR-SDRAM sont résolus par construction et ne nécessitent alors pas d'autre support en-ligne de la part de l'hyperviseur que celui de la gestion adéquate du NoC. En outre, en connaissant à l'avance les zones mémoires visées par les partitions (cf. règle 3) et en prenant pour hypothèse la configuration du contrôleur DDR-SDRAM décrite en Section A.6.1.1, le calcul de bornes sûres sur les temps nécessaires à la réalisation des transactions mémoire est largement simplifié par rapport au cas général, permettant ainsi d'effectuer efficacement la vérification hors-ligne du non dépassement des créneaux de communication des PCs.

A.6.2 Validation expérimentale

Afin de démontrer la capacité de notre modèle d'exécution à apporter une propriété d'isolation temporelle entre partitions concurrentes et de montrer que l'implémentation de celui-ci sur le KALRAY MPPA®-256 est possible, nous avons effectué une validation expérimentale basée sur un cas d'étude académique.

A.6.2.1 Mise en œuvre

Pour montrer l'isolation temporelle entre partitions, nous utilisons le cas d'étude ROSACE [153] comme application de référence aux côtés de laquelle s'exécutera dans une autre partition une application de traitement d'image générant des interférences. Par l'étude du comportement de ROSACE dans plusieurs scénarios de concurrence, nous cherchons à montrer son insensibilité aux interférences générées par la deuxième application.

Le cas d'étude ROSACE est, bien que de taille modeste, représentatif dans sa structure des applications avioniques réelles, et ce notamment par son exécution multi-périodique complexe. Comme cela est montré sur la Figure A.12, ROSACE est composé d'un *contrôleur* longitudinal de vol et d'un *environnement de simulation* qui représente le modèle physique d'un avion. L'ensemble comporte 11 blocs s'exécutant à 50Hz, 100Hz ou 200Hz.

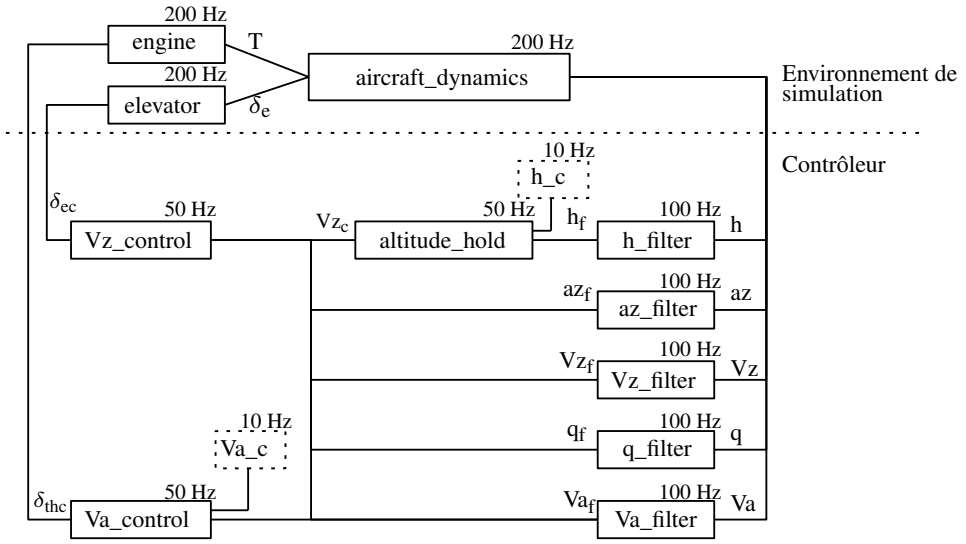


Figure A.12: Contrôleur longitudinal de vol du cas d'étude ROSACE

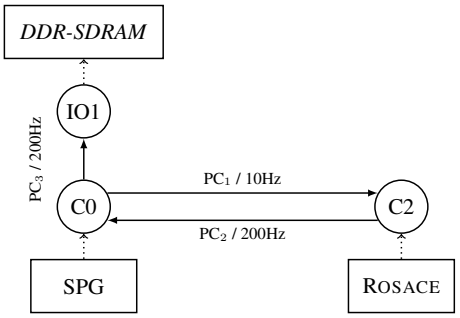


Figure A.13: Budget de la partition de ROSACE avec 2 PNs et 3 PCs

Nous avons fait le choix de placer l'ensemble de ces 11 blocs au sein d'un unique PN pour nos expériences en raison de la faible taille du code. Ce PN utilise 5 PEs sur lesquels l'exécution de ROSACE est parallélisée et 3 bancs de SRAM locale. Pour rendre notre cas d'étude plus complet, nous avons ajouté à celui-ci un générateur de consignes appelé *SPG*. Il est chargé de simuler les commandes du pilote en transmettant en-ligne des consignes d'altitude et de vitesse (h_c et V_{a_c}) au contrôleur. Le SPG est placé dans un second PN connecté à ROSACE par deux PCs pour transmettre les commandes et recevoir les sorties de simulation. Le SPG est également connecté à un ION par un troisième PC afin de loguer en DDR-SDRAM les traces de simulation. Ces 2 PNs et 3 PCs forment une partition dont la structure est présentée sur la Figure A.13

L'application concurrente que nous considérons, appelée *ImgInv*, effectue l'inversion d'une image en noir et blanc comme montré sur la Figure A.14. ImgInv est facilement parallélisable et est particulièrement configurable. L'ajustement de la taille des images à inverser permet de contrôler efficacement la charge de travail pour les PEs, l'empreinte mémoire de la partition et l'utilisation du NoC pour l'envoi et la réception d'images. Par ailleurs, l'implémentation d'ImgInv peut être qualifiée d'aisée.



(a) Avant l'inversion (b) Après l'inversion

Figure A.14: Exemple d'utilisation d'ImgInv

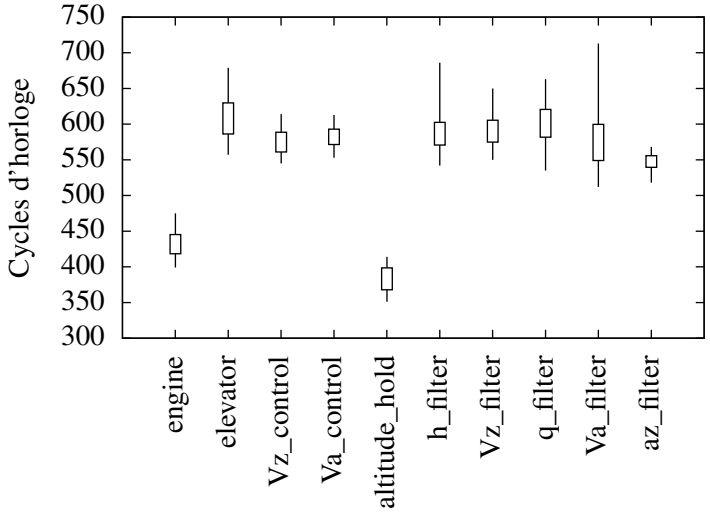


Figure A.15: Temps d'exécution des 11 blocs de ROSACE

A.6.2.2 Scénarios

A partir des deux applications décrites précédemment, nous avons exécuté 3 scénarios permettant de mettre en exergue la propriété d'isolation temporelle entre les deux partitions les hébergeant.

Scénario 1, pas d'interférences Dans un premier temps, nous avons exécuté la partition de ROSACE seule sur le KALRAY MPPA®-256 afin de définir une exécution de référence qui sera par la suite comparée aux différents cas de concurrence. Comme montré sur la Figure A.13, le PN de ROSACE est placé sur le cluster 2 et le PN du SPG est placé sur le cluster 0. Cette exécution nous a permis de valider fonctionnellement notre implémentation de ROSACE en traitant à posteriori les données loguées en DDR-SDRAM. De plus, nous avons pu mesurer les temps d'exécution des 11 blocs comme montré sur la Figure A.15. Sur cette figure les temps d'exécution minimaux et maximaux mesurés sont représentés par les lignes et les encadrés représentent la déviation standard centrée sur la moyenne. Chaque bloc est exécuté plus de 10 000 fois. Les temps d'exécution du bloc "aircraft_dynamics" sont d'en moyenne 59 000 cycles d'horloge et ne sont pas représentés sur la figure pour plus de lisibilité.

Scénario 2, interférences sur le NoC Dans un second temps, nous avons ajouté au scénario 1 une partition contenant 2 instances d'ImgInv. Comme cela est montré sur la Figure A.16, chaque instance est placée dans un PN. Le premier est positionné sur le cluster 3 et le deuxième sur

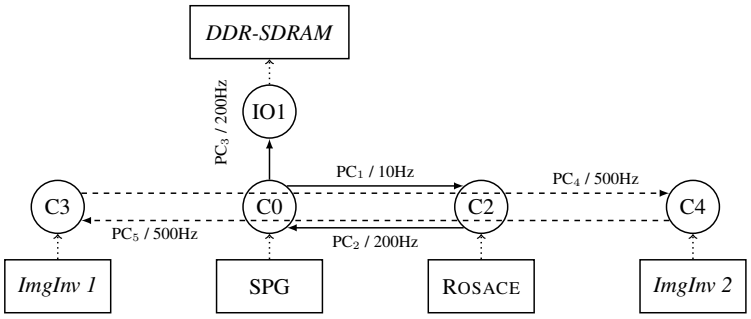


Figure A.16: Configuration du deuxième scénario avec deux instances d'ImgInv partageant des ressources NoC avec ROSACE

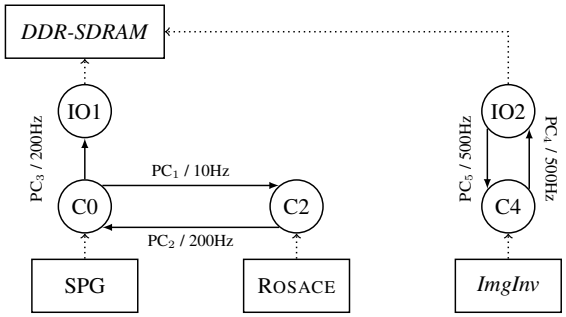


Figure A.17: Configuration du scénario avec une instance d'ImgInv partageant un banc de DDR-SDRAM avec ROSACE

le cluster 4. Ces deux instances sont reliées par deux PCs dont les routes sont délibérément choisies pour partager des ressources NoC avec la partition de ROSACE. Ainsi, la première instance d'ImgInv inverse une image et l'envoie par le NoC. A sa réception, la deuxième instance d'ImgInv inverse l'image et la renvoie à son tour et ainsi de suite. L'intérêt fonctionnel d'un tel travail est clairement faible mais cela permet de générer du trafic sur le NoC et ainsi de vérifier l'évitement des conflits avec ROSACE.

Dans cette configuration, nous avons pu observer que les temps d'exécution des blocs de ROSACE semblent insensibles à la nouvelle source d'interférences. Plus précisément, les meilleurs et pires temps d'exécution des 11 blocs mesurés lors du scénario 1 sont strictement égaux à ceux mesurés lors du scénario 2. De plus, en modifiant temporairement l'hyperviseur pour loguer les dates de réception des paquets reçus depuis le NoC, nous avons pu vérifier que 100% de ceux-ci le sont lors des créneaux d'activation des PCs par lesquels ils sont envoyés. Cela implique qu'aucun paquet n'a subi de retard lié aux communications de la partition concurrente et que l'évitement des conflits sur le NoC semble respecté.

Scénario 3, interférences sur la DDR-SDRAM Dans un dernier temps, nous avons ajouté au scénario 1 une partition contenant 1 instance d'ImgInv. Cette instance est placée dans un PN communiquant avec un ION afin de lire et d'écrire les images depuis/vers la DDR-SDRAM. Comme montré sur la Figure A.17, le PN d'ImgInv est placé sur le cluster 4 et l'ION est placé sur le même cluster IO que celui de ROSACE. De plus, la zone mémoire depuis laquelle les images sont lues et écrites est placée dans le même banc mémoire que celui utilisé par ROSACE pour simuler un cas d'interférence.

Dans cette configuration nous avons pu une nouvelle fois observer l'insensibilité des temps

d'exécution des 11 blocs de ROSACE aux interférences. Comme pour le scénario 2, les pires et meilleurs temps d'exécution de ces blocs sont strictement égaux à ceux mesurés lors du scénario 1. Une nouvelle fois, 100% des paquets émis sur le NoC sont reçus pendant leur créneau d'envoi. Et enfin, en utilisant un *interposeur* DDR-SDRAM pour récupérer de manière non intrusive les traces d'accès à la mémoire externe, nous avons pu observer que les transactions issues des deux partitions ne s'entrelacent pas et s'exécutent lors de périodes bien distinctes.

Avec ces trois scénarios, nous observons que, pour notre cas d'étude, la propriété d'isolation temporelle attendue semble être effective sur le système réel. Bien que n'apportant pas une preuve formelle de correction, ces résultats sont prometteurs et représentent déjà en eux-mêmes un intérêt certain d'un point de vue industriel.

A.6.3 Synthèse

Dans cette section, nous avons décrit l'implémentation de notre modèle d'exécution sur le KALRAY MPPA®-256. Le respect de ses règles est assuré en-ligne par l'usage d'un hyperviseur en charge des différentes configurations matérielles et par des travaux de vérification budgétaire effectués hors-ligne pour garantir que toutes les règles sont respectées, et en particulier l'ordonnancement global du NoC. Nous avons montré avec un cas d'étude académique que la propriété d'isolation temporelle semble se vérifier en pratique, et ce malgré le partage de ressources au niveau du NoC ou de la DDR-SDRAM.

Au cours de cette section, nous avons pris l'hypothèse que les placements et ordonnancements des applications au sein de leurs budgets respectifs étaient connus. Dans le cas d'applications de grandes tailles, un calcul manuel est souvent rédhibitoire. Pour cette raison, nous montrons dans la section suivante comment, en utilisant la programmation par contraintes, il est possible de calculer *automatiquement* et efficacement des placements et ordonnancements d'applications industrielles sur plusieurs clusters du KALRAY MPPA®-256.

A.7 Validation des budgets

Dans cette section, nous détaillons comment l'étape de *Validation* de l'atelier d'intégration (Figure A.9) peut être automatisée en utilisation la programmation par contraintes. L'objectif de cette étape est de déterminer si une application peut s'exécuter correctement à l'intérieur du budget de sa partition. Pour répondre à cette problématique, nous proposons une approche permettant de calculer un placement et un ordonnancement d'une application sur un budget donné tout en respectant ses contraintes fonctionnelles et non fonctionnelles. D'une façon générale, nous considérons les budgets tels que formalisés en Section A.5.1 et le modèle d'applications de la Section A.2.1.

A.7.1 Choix des budgets

A cette étape du processus d'intégration, les budgets sont connus. Le choix d'un budget plutôt qu'un autre peut avoir des conséquences majeures sur l'implémentation et sur les performances d'une application. Nous clarifions ici les hypothèses que nous prenons sur ces budgets et nous proposons des recommandations pour les déterminer en maximisant leurs chances d'être valides.

A.7.1.1 Hypothèses

Chargement du code Nous supposons dans cette section que les applications ne chargent *pas* de code depuis la DDR-SDRAM externe. Cela implique que la totalité de l'application doit être stockée dans les mémoires SRAM locale des clusters de calcul. Cette hypothèse, bien que clairement contraignante, est motivée par deux raisons:

1. La problématique du chargement de code dans les clusters de calcul depuis la DDR-SDRAM est un sujet déjà abordé dans la littérature. Becker *et al.* [146] ont par exemple proposé une méthode d'ordonnancement de *Runnables* AUTOSAR sur un cluster du KALRAY MPPA®-256 en prenant en compte le chargement de leur code par le NoC. Avec notre approche, nous proposons une contribution qui est complémentaire aux travaux déjà effectués et faisant avancer l'état de l'art.
2. Cette hypothèse implique que les applications de grande taille, c'est-à-dire trop grosses pour être stockées dans un seul cluster, soient réparties sur plusieurs. Ainsi, le problème de la gestion du NoC devient prépondérant dans notre cas. Nous voulons par ce travail adresser une problématique purement liée à l'architecture d'un processeur pluri-cœur. Même s'il n'est pas évident que cette approche permette d'exploiter au maximum la capacité de calcul du KALRAY MPPA®-256, elle reste particulièrement intéressante à étudier pour déterminer les solutions permettant la parallélisation massive sur une architecture basée NoC et qui deviendra inévitablement nécessaire à l'avenir compte tenu de la croissance des besoins en puissance de calcul.

Hyperviseur Comme expliqué lors de la section précédente, notre capacité d'implémentation de l'hyperviseur pose quelques contraintes sur les budgets qui peuvent être choisis par les partitions. Tout d'abord, un banc de SRAM locale est réservé au sein de chaque cluster de calcul pour stocker le code de l'hyperviseur, les micro-codes DMA ainsi que les tables d'ordonnancement. Le nombre maximum de bancs de SRAM qu'un PN peut intégrer est donc borné par $N_b(pn) = 15$. De plus, les transferts DMA démarrant à l'activation de l'hyperviseur, les durées, périodes et offsets des PCs doivent être des multiples de la période d'activation de celui-ci. Avec une période d'activation de l'hyperviseur de $5\mu s$ par exemple, un PC de période $20\mu s$ sera défini par $T(pc) = 4$.

A.7.1.2 Contraintes minimales

A cette étape du processus d'intégration, les budgets doivent être connus. Il apparaît toutefois que les définir est souvent une tâche ardue qui peut parfois être contre-intuitive. Pour aider les développeurs d'application à définir la quantité de ressources dont ils ont besoin, nous donnons ici des recommandations sur le choix de ces budgets. Plus précisément, nous exprimons un jeu de conditions nécessaires (mais non suffisantes) qui, lorsque elles sont toutes satisfaites, n'apportent pas une garantie certaine de la validité d'un budget mais qui permettent de gagner en confiance sur la probabilité qu'il le sera.

Contraintes mémoire Les applications ne chargeant pas de code en-ligne depuis la DDR-SDRAM, elles doivent être intégralement stockées dans la SRAM locale des clusters de calcul. A partir de l'empreinte mémoire d'une application, nous pouvons tout d'abord vérifier que les PNs qui lui sont associés comprennent suffisamment de stockage pour la contenir complètement.

Pour cela, la condition A.1 doit être remplie.

$$\sum_{pn \in \mathcal{N}_i} N_b(pn) \times S_{bank}^{SRAM} \geq \sum_{\tau_i^j \in S} M_i^j \tag{A.1}$$

Par ailleurs, sachant que seules 15 bancs de SRAM peuvent être associés à chaque PN, il est impératif que le nombre de PNs compris dans une partition satisfasse la condition A.2.

$$|\mathcal{N}_i| \geq \left\lceil \frac{\sum_{\tau_i^j \in S} M_i^j}{15 \times S_{bank}^{SRAM}} \right\rceil \tag{A.2}$$

Contraintes computationelles Comme pour la mémoire, il est possible de déterminer à partir du profil d'application des contraintes sur la capacité de calcul comprise dans un budget. Tout d'abord, nous pouvons définir le ratio d'utilisation d'une application U par:

$$U = \sum_{\tau_i^j \in S} \frac{C_i^j}{T_i}$$

Sachant que le nombre de PEs sur lesquels s'exécute l'application doit impérativement être plus grand que U, la condition A.3 doit être remplie.

$$\sum_{pn \in \mathcal{N}_i} N_c(pn) \geq \lceil U \rceil \tag{A.3}$$

Par ailleurs, avec 16 PEs par cluster sur le KALRAY MPPA®-256, le nombre de PNs d'un budget doit satisfaire la condition A.4.

$$|\mathcal{N}_i| \geq \left\lceil \frac{U}{16} \right\rceil \tag{A.4}$$

A.7.2 Processus de validation

Notre méthode de validation repose sur l'utilisation de la programmation par contraintes. Cette approche permet de prendre en compte les modèles complexes de l'architecture du KALRAY MPPA®-256 et des applications tout en restant dans un cadre formel.

A.7.2.1 Environnement de modélisation

Plusieurs travaux explorent déjà la problématique du placement et de l'ordonnancement de tâches dépendantes sur des processeurs multi- ou pluri-cœurs en utilisant la programmation linéaire en nombres entiers (ou *ILP* en anglais). De manière générale, ces approches souffrent de difficultés pour traiter des applications de grandes tailles. Ces difficultés sont souvent surmontées en utilisant des heuristiques de placement qui facilitent grandement le passage à l'échelle en abandonnant l'optimalité. Ici, nous utilisons un environnement de modélisation fondé sur la programmation par contraintes et qui a montré son efficacité pratique pour surmonter les difficultés de passage à l'échelle que nous avions également avec une formulation préliminaire basée ILP. Cet environnement est fondé sur la notion de *Variable d'intervalle optionnelle* [157, 158] introduite dans IBM

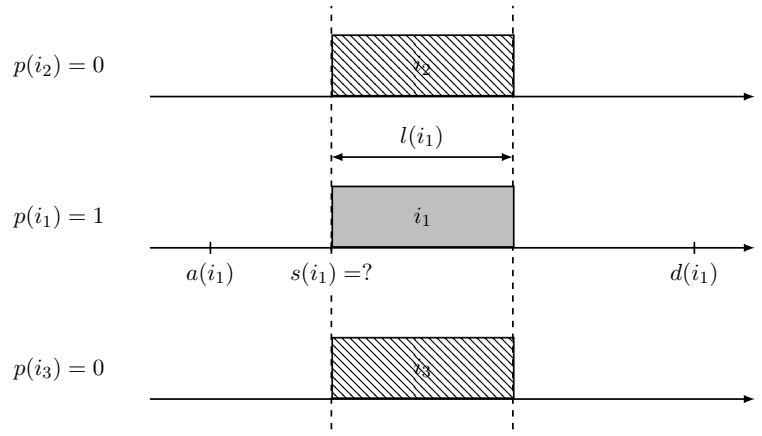


Figure A.18: Exemple de 3 variables d'intervalle optionnelles i_1, i_2 et i_3 où seulement une est présente

ILOG CP Optimizer [159] depuis sa version 2.0. Une *variable d'intervalle*, ici appelée i, représente une activité de durée finie. L'objectif du solveur est de calculer la date de début de i notée $s(i)$. La durée de i est notée $l(i)$. Chaque variable d'intervalle est définie dans une fenêtre temporelle $[a(i), d(i)]$ où $a(i)$ et $d(i)$ représentent respectivement la date d'activation et l'échéance de i. Plus précisément, cela veut dire que tout ordonnancement calculé par le solveur doit vérifier $s(i) \geq a(i)$ et $s(i) + l(i) \leq d(i)$.

Une variable d'intervalle est *optionnelle* si sa présence dans la solution n'est systématiquement requise. Une variable d'intervalle optionnelle possède un attribut booléen $p(i)$ qui indique sa présence ou son absence dans la solution. Une variable absente ne sera alors pas prise en compte dans le jeu de contraintes à satisfaire. Cette notion peut notamment être mise à profit dans le cas où le problème à résoudre comporte non seulement une dimension temporelle (le calcul d'un ordonnancement) mais également une dimension spatiale (le placement d'intervalles sur des ressources). Par exemple, dans le cas où un jeu de tâches devrait être ordonnancé et placé sur un processeur multi-cœur, une formulation naturelle comporterait pour chaque tâche un ensemble de variables d'intervalles optionnelles (n intervalles pour n cœurs) avec la contrainte qu'une et une seule de ces variables soit présente dans la solution finale. Cela encoderait alors l'allocation d'une tâche à un cœur.

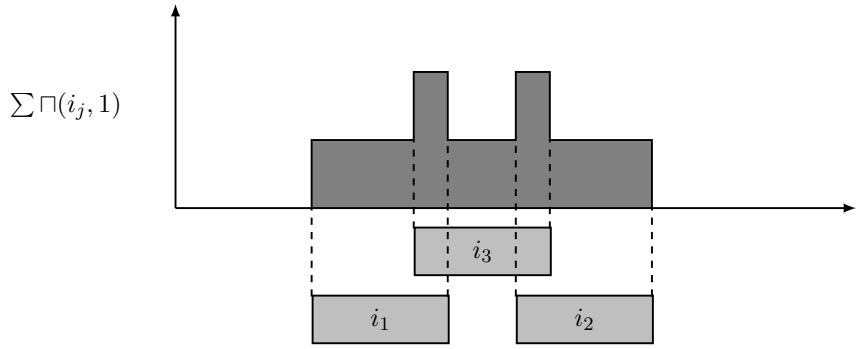
IBM ILOG CP Optimizer permet également l'utilisation du concept de *fonction cumulative*. Elle représente l'usage cumulatif qui est fait d'une ressource par des activités au cours du temps. On les utilise généralement afin de restreindre l'utilisation d'une ressource à des capacités limitées. Dans notre formulation, nous n'utiliserons que les fonctions *portes* notées $\sqcap(i, h)$ avec i une variable d'intervalle et h la quantité de ressource consommée sur la durée de i. La Figure A.19 représente un exemple d'utilisation d'une telle fonction porte avec notamment un cas contraint mettant en œuvre l'exclusion mutuelle des 3 intervalles considérés.

Afin de simplifier les notations, nous utiliserons la notation suivante pour forcer la présence d'une seule variable d'intervalle optionnelle parmi un ensemble $I = \{i_1, \ldots, i_n\}$:

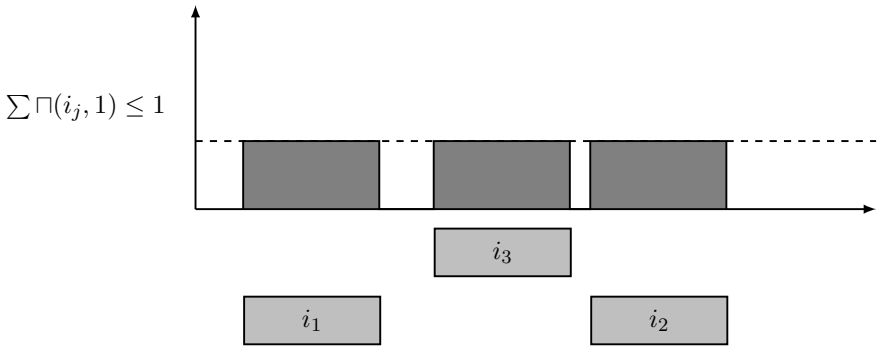
$$\oplus(I) = 1 \Leftrightarrow \sum_{i \in I} p(i) = 1$$

Du plus nous définissons comme suit l'opérateur de précédence entre deux intervalles:

$$i_1 \rightarrow i_2 \Leftrightarrow s(i_1) + l(i_1) \leq s(i_2)$$



(a) Une fonction porte appliquée à i_1, i_2 et i_3 *sans* contrainte



(b) Une fonction porte appliquée à i_1, i_2 et i_3 *avec* contraintes

Figure A.19: Exemple d'utilisation des fonctions portes sur 3 intervalles i_1, i_2 et i_3

A.7.2.2 Formulation du problème

Notre objectif est de calculer un placement et un ordonnancement d'une application $\mathcal{A} =< \tau, \delta >$ sur un budget $< \mathcal{N}, \mathcal{I}, \mathcal{C}, \mathcal{B} >$. L'utilisation de la notion de variable d'intervalle implique de travailler au niveau des instances et pas des tâches elles mêmes. Sur une hyper-période

$$T_H = \underset{\tau_i \in \tau}{\text{lcm}}(T_i)$$

avec T_i la période de τ_i, chaque tâche peut être activée plusieurs fois. Nous parlerons du k-ième *job* de τ_i, noté $\tau_{i,k}$, pour se référer à sa k-ième activation. De même, la k-ième activation de la sous-tâche τ_i^j sera appelée k-ième sous-job, noté $\tau_{i,k}^j$. L'ensemble de tous les sous-jobs exécutés sur une hyper-période est noté S. Les relations de précédences entre sous-tâches d'une même tâche sont dupliquées entre sous-jobs du même job. La k-ième production d'une donnée δ_x par le sous-job $\tau_{i,k}^j$ est notée $\delta_{x,k}$.

La gestion des buffers d'entrée et de sortie de l'application se traduit par des envois ou des réceptions de données vers ou depuis la DDR-SDRAM externe. Ainsi, la gestion de ces données n'est pas différente de celle des données entre sous-tâches à l'exception du fait que les PCs utilisés pour les envoyer sont ceux connectés aux IONs.

Pour notre problème de placement et d'ordonnancement, nous considérons deux types de variables de décision.

Les sous-jobs sur les PNs Nous associons à chaque sous-job $\tau_{i,k}^j$ un ensemble de $|\mathcal{N}|$ variables d'intervalles conditionnelles. On note $j(\tau_{i,k}^j, pn)$ la variable représentant l'allocation de $\tau_{i,k}^j$

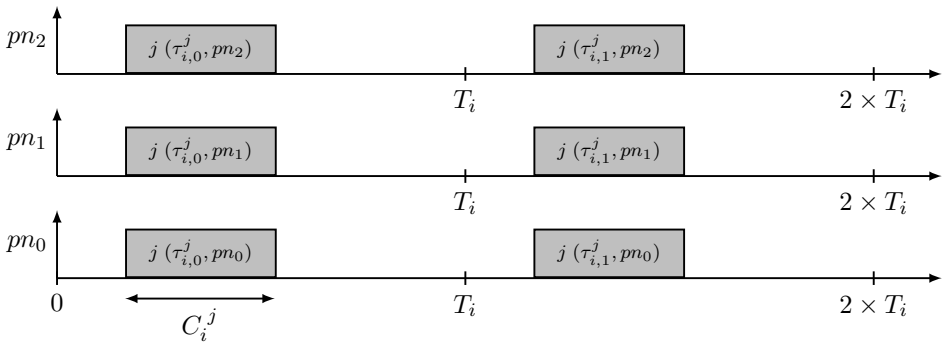


Figure A.20: 6 intervalles représentant 1 sous-tâche τ_i^j avec 2 sous-jobs ordonnançable sur 3 PNs

sur le PN pn. La durée de chacun de ces intervalles est égale au WCET de la sous-tâche correspondante, soit plus formellement $l(j(\tau^j_{i,k}, pn)) = C_i^j$. La fenêtre d'existence de chaque intervalle correspond à la fenêtre d'existence du sous-job associé sur l'hyper-période, c'est-à-dire $[k \times T_i, (k+1) \times T_i]$. Cela est représenté sur la Figure A.20

Les données sur les PCs Nous associons à chaque instance de donnée $\delta_{x,k}$ un ensemble de $|\mathcal{C}|$ variables d'intervalles conditionnelles. On note $d(\delta_{x,k}, pc)$ l'intervalle représentant l'envoi de $\delta_{x,k}$ au travers du PC pc. Lorsque $d(\delta_{x,k}, pc)$ est présente dans la solution finale, cela implique que $\delta_{x,k}$ est envoyée depuis le PN $src(pc)$ à destination du PN $dst(pc)$ au travers de pc. Dans le cas des données, la durée des intervalles les représentant n'est pas liée au temps nécessaire à l'envoi de la donnée sur le NoC. La durée de ces intervalles est fixée comme égale à la largeur d'un créneau d'accès au NoC sur le PC considéré. Plus formellement, on définit $l(d(\delta_{x,k}, pc)) = C(pc)$. La limite sur la quantité de données qui peuvent être envoyées via un PC sera assurée par la suite en utilisant des fonctions cumulatives. La date de début d'une variable d'intervalle $d(\delta_{x,k}, pc)$ encode ici l'allocation de cette donnée à un slot spécifique de son pc. La fenêtre d'existence des intervalles de données est égale à celle de leur sous-job producteur.

A.7.2.3 Contraintes

Notre formulation du problème d'allocation et d'ordonnancement comporte plusieurs familles de contraintes avec des objectifs bien précis. Nous détaillons ces contraintes ici.

Placement des sous-jobs Chaque sous-job ne peut être exécuté que par un seul PN. Ceci se traduit par la contrainte A.5 qui impose qu'une seule variable d'intervalle soit présente pour chaque sous-job.

$$\forall \tau^j_{i,k} \in S, \oplus(\{j(\tau^j_{i,k}, pn) \mid \forall pn \in \mathcal{N}\}) = 1 \tag{A.5}$$

Si les sous-jobs d'une même sous-tâche pouvaient s'exécuter sur différents PNs, c'est-à-dire migrer d'un PN à un autre entre deux activations, cela impliquerait que le code de cette sous-tâche soit dupliqué sur plusieurs PNs et mettrait ainsi de la pression sur une ressource sensible: la SRAM locale. De plus, les données statiques de la sous-tâche devraient être envoyées d'un cluster à l'autre pour garantir une exécution cohérente, et impliqueraient alors davantage de trafic sur une deuxième ressource sensible: le NoC. Pour éviter ces problèmes et simplifier notre formulation, nous interdisons ces migrations en forçant tous les sous-jobs

d'une même sous-tâche à s'exécuter sur le même PN. Cela est traduit par la contrainte A.6.

$$\forall \tau^j_{i,k} \in S,\ \forall pn \in \mathcal{N},\ p(j(\tau^j_{i,k}, pn)) = p(j(\tau^j_{i,(k+1)\%\frac{T_H}{T_i}}, pn)) \tag{A.6}$$

Utilisation des PNs Le nombre de PEs au sein d'un PN étant limité, le nombre de sous-jobs qui peuvent s'exécuter en parallèle sur ce PN doit l'être également. C'est ce qu'exprime la contrainte A.7 en utilisant des fonctions cumulatives.i

$$\forall pn \in \mathcal{N},\ \sum_{\forall \tau^j_{i,k} \in S} \sqcap (j(\tau^j_{i,k}, pn), 1) \leq N_c(pn) \tag{A.7}$$

Par ailleurs, les sous-tâches allouées à un PN doivent être stockables sur celui-ci en intégralité. La contrainte A.8 garantit le respect de cette limitation mémoire. Il peut être remarqué que la contrainte A.8 ne vise que la première activation de chaque sous-tâche (avec $\tau^j_{i,0}$). En effet, les migrations étant interdites, toutes les activations de la même sous-tâche utiliseront le même code.

$$\forall pn \in \mathcal{N},\ \sum_{\tau^j_i \in S} p(j(\tau^j_{i,0}, pn)) \times M^j_i \leq N_b(pn) \times S^{SRAM}_{bank} \tag{A.8}$$

Placement des données Le placement d'une donnée sur un PC dépend du positionnement de sa sous-tâche productrice et de celui de ses sous-tâches consommatrices. Lorsque producteur et consommateur sont sur le même PN, la communication s'effectue par mémoire partagée au sein du cluster et aucun envoi de donnée sur le NoC n'est nécessaire. Dès qu'un consommateur est positionné sur un PN distant, la donnée produite doit être émise sur le PC adéquat. Ceci est capturé par la contrainte A.9.

$$\begin{aligned}&\forall pc \in \mathcal{C},\ \forall \delta_x \in \delta,\\ &\Big(p(j(prod(\delta_{x,0}), src(pc)))\ \wedge \sum_{\tau^j_{i,0} \in cons(\delta_{x,0})} p(j(\tau^j_{i,0}, dst(pc))) \geq 1 \Big) = p(\delta_{x,0}, pc)\end{aligned} \tag{A.9}$$

Les sous-tâches ne pouvant migrer, les données à envoyer doivent toujours l'être sur les mêmes PCs. Ceci se traduit par la contrainte A.10.

$$\forall pc \in \mathcal{C},\ \forall \delta_{x,k} \in \delta,\ p(d(\delta_{x,k}, pc)) = p(d(\delta_{x,k+1}, pc)) \tag{A.10}$$

Enfin, la date de début des intervalles de données encode l'allocation d'une donnée à un certain slot de son PC. Pour cela, sa date début doit être alignée avec le début d'un slot du PC en question, comme cela est fait via la contrainte A.11.

$$\forall pc \in \mathcal{C},\ \forall \delta_{x,k} \in \delta,\ s(d(\delta_{x,k}, pc))\ \%\ T(pc) = O(pc) \tag{A.11}$$

Utilisation des PCs La quantité de données qui peut être envoyée au travers des PCs doit être limitée pour correspondre aux capacités des liens NoC et des DMAs. Notre modèle d'exécution garantissant l'absence de conflits sur le NoC, nous pouvons déduire le nombre de flits qui peuvent traverser un PC pc durant sa durée d'activation comme

$$N^{PC}_{flit}(pc) = C(pc) - (R(pc) + 1) \times \Delta_L - R(pc) \times \Delta_S$$

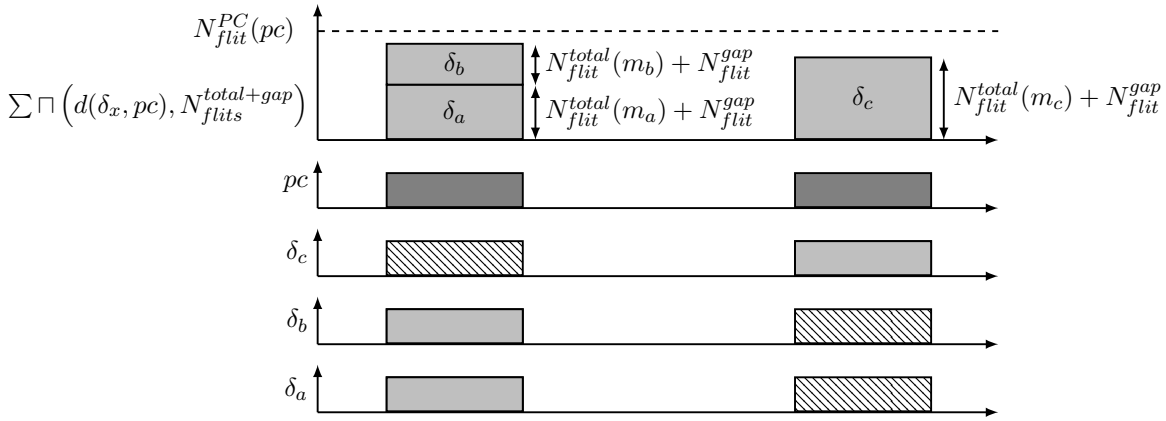


Figure A.21: Limitation de la quantité de données qui peut être envoyée durant l'activation d'un PC en utilisant les fonctions cumulatives sur les variables d'intervalle des données

avec Δ_L la latence d'un lien NoC et Δ_S la latence d'un switch en absence de collision. En notant $N^{total}_{flit}(m_x)$ le nombre de flits nécessaires à l'envoi des m_x octets d'une donnée δ_x, on peut contraindre la quantité de données à envoyer à travers chaque activation d'un PC via la contrainte A.12. Cela est représenté graphiquement sur la Figure A.21.

$$\forall pc \in \mathcal{C}, \sum_{\delta_{x,k} \in \delta} \sqcap\Big(d(\delta_{x,k}, pc), N^{total}_{flit}(m_x) + N^{gap}_{flit}\Big) \leq N^{PC}_{flit}(pc) \tag{A.12}$$

Le terme N^{gap}_{flit} de la contrainte A.12 représente le nombre de flits perdus lors d'un saut du DMA d'une zone mémoire à une autre. En le comptant pour toutes données envoyées, on fait l'hypothèse conservative que les données sont placées dans des zones mémoire non contigües et que le DMA doit sauter de l'une à l'autre lors d'un envoi. Si l'on intégrait au problème d'ordonnancement le calcul des positions de données en mémoire, il serait possible de concaténer des données et d'éviter le surcoût systématique de N^{gap}_{flit}. S'il est clair que cela pourrait permettre de gagner en performance, il est moins évident que l'accroissement de la complexité soit assez faible pour pouvoir traiter des problèmes de grandes tailles avec les machines et les solveurs actuels. L'évaluation de ce compromis apparaît toutefois comme une perspective intéressante d'amélioration de notre travail.

Les DMA n'étant capables de traiter automatiquement qu'une quantité limitée de travaux, le nombre de zones non contigües qui peuvent être émises pendant une seule activation doit être contraint. Avec N^{DMA}_{bufs} le nombre maximum de zones mémoire que le DMA peut traiter de manière autonome, cela est traduit dans la contrainte A.13.

$$\forall pc \in \mathcal{C}, \sum_{\delta_{x,k} \in \delta} \sqcap(d(\delta_{x,k}, pc), 1) \leq N^{DMA}_{bufs} \tag{A.13}$$

Relations de précédence Dans notre modèle d'application, les contraintes de précédences ne sont imposées qu'entre deux sous-tâches échangeant des données. Ces données peuvent être classifiées en deux catégories:

- les données *en avant* sont produites par la sous-tâche parente de la relation de précédence. Plus formellement, avec une précédence (τ^j_i, τ^l_i) où τ^j_i doit s'exécuter avant τ^l_i

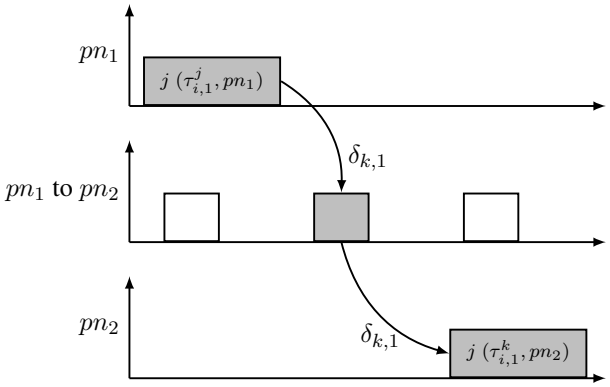


Figure A.22: Intervalle de donnée utilisé comme pivot pour mettre en œuvre une précédence entre deux sous-jobs placés sur 2 PNs distants.

(τ_i^j est donc la sous-tâche parente), δ_x est une donnée en avant si $(\tau_i^j = prod(\delta_x)) \wedge (\tau_i^l \in cons(\delta_x))$;

- les données *en arrière* sont produites par la sous-tâche fille de la relation de précédence. C'est-à-dire que la donnée produite sera consommée lors de la prochaine activation de la sous-tâche parente. Plus formellement, avec une précédence (τ_i^j, τ_i^l) où τ_i^j est parente, δ_x est une donnée en arrière si $(\tau_i^j \in cons(\delta_x)) \wedge (\tau_i^l = prod(\delta_x))$.

Lorsque les deux sous-tâches contraintes par une relation de précédence sont placées sur le même PN, la communication est assurée par mémoire partagée et il suffit de garantir l'ordre d'exécution de ces sous-tâches avec la contrainte A.14.

$$\forall(\tau_{i,k}^j, \tau_{i,k}^l) \in P_{i,k},\ \forall pn \in \mathcal{N},\ j(\tau_{i,k}^j, pn) \rightarrow j(\tau_{i,k}^l, pn) \tag{A.14}$$

Si les deux sous-tâches sont sur des PNs différents, la donnée doit être émise au travers du NoC. Avec des données en avant, nous utilisons l'intervalle de la donnée en question comme pivot de précédence pour garantir la précédence entre les sous-tâches. C'est-à-dire que le sous-job producteur de la donnée est forcé de s'exécuter avant le début du créneau d'envoi de la donnée sur son PC comme cela est fait avec la contrainte A.15. Et ensuite, le sous-job consommateur est quant à lui contraint de démarrer après la fin du slot d'envoi pour garantir la bonne réception de cette donnée au moment de son exécution. Cela est traduit par la contrainte A.16. La Figure A.22 montre une représentation graphique de cette notion de pivot.

$$\forall pc \in \mathcal{C},\ \forall \delta_{x,k} \in \delta,\ j(prod(\delta_{x,k}), src(pc)) \rightarrow d(\delta_{x,k}, pc) \tag{A.15}$$

$$\begin{aligned}&\forall(\tau_{i,k}^j, \tau_{i,k}^l) \in P_{i,k},\ \forall pc \in \mathcal{C},\ \forall \delta_{x,k} \in \delta,\\ &\quad (\tau_{i,k}^j = prod(\delta_{x,k})) \wedge (\tau_{i,k}^l \in cons(\delta_{x,k})) \Rightarrow d(\delta_{x,k}, pc) \rightarrow j(\tau_{i,k}^l, dst(pc))\end{aligned} \tag{A.16}$$

Dans le cas des données en arrière, la donnée consommée par le sous-job parent $\tau_{i,k}^j$ de la précédence est celle produite par le sous-job précédent de sous-tâche fille $\tau_{i,k-1}^l$. Lorsque producteurs et consommateurs sont sur le même PN, la contrainte A.14 suffit à assurer un

comportement correct. En effet, la fenêtre d'exécution de la donnée produite par $\tau^l_{i,k-1}$ termine au début de celle de la fenêtre d'exécution $\tau^j_{i,k}$. En conséquence l'ordre d'émission de consommation est naturellement assuré par construction. Si les deux sous-tâches sont placées sur des PNs différents, il faut garantir que le sous-job parent se termine avant l'envoi de la donnée produite par le sous-job fils afin d'éviter la consommation d'une donnée "trop fraîche". Cela est fait avec la contrainte A.17.

$$\begin{aligned}&\forall(\tau^j_{i,k},\tau^l_{i,k}) \in P_{i,k},\ \forall pc \in \mathcal{C},\ \forall \delta_{x,k} \in \delta,\\ &\quad(\tau^j_{i,k} \in cons(\delta_{x,k})) \wedge (\tau^l_{i,k} = prod(\delta_{x,k})) \Rightarrow j(\tau^j_{i,k}, dst(pc)) \rightarrow d(\delta_{x,k}, pc)\end{aligned} \tag{A.17}$$

Déterminisme Certaines données peuvent être échangées par des sous-tâches non contraintes par une relation de précédence. Dans le cas où un sous-job producteur et un sous-job consommateur s'exécuteraient en même temps, il n'est pas évident que l'ordre de production et de consommation de la donnée soit connu, et il est même possible qu'il varie d'une exécution à l'autre. Nous qualifions cette possible irrépétabilité d'exécution de l'ordonnancement comme non déterministe. Pour éliminer ce non déterminisme et rendre les exécutions répétables, nous forçons l'ordre de production et de consommation des données non contraintes, quel qu'il soit, à rester consistant d'une exécution à l'autre. Quand les sous-tâches productrices et consommatrices sont assignées au même PN, on interdit leur recouvrement temporel par le biais de la contrainte A.18.

$$\begin{aligned}&\forall \delta_{x,k} \in \delta,\ \forall \tau^j_{i,l} \in cons(\delta_{x,k}),\ \forall pn \in \mathcal{N},\\ &\quad(prod(\delta_{x,k}), \tau^j_{i,l}) \notin P_k \wedge (\tau^j_{i,l}, prod(\delta_{x,k})) \notin P_k\\ &\quad\Rightarrow \sqcap(j(prod(\delta_{x,k}), pn), 1) + \sqcap(j(\tau^j_{i,l}, pn), 1) \leq 1\end{aligned} \tag{A.18}$$

Dans le cas où producteurs et consommateurs sont placés sur des PNs différents, c'est le créneau de réception de la donnée et l'exécution du sous-job consommateur qui ne doivent pas être simultanés. Ceci est traduit par la contrainte A.19.

$$\begin{aligned}&\forall \delta_{x,k} \in \delta,\ \forall \tau^j_{i,l} \in cons(\delta_{x,k}),\ \forall pc \in \mathcal{C},\\ &\quad(prod(\delta_{x,k}), \tau^j_{i,l}) \notin P_k \wedge (\tau^j_{i,l}, prod(\delta_{x,k})) \notin P_k\\ &\quad\Rightarrow \sqcap(d(\delta_{x,k}, pc), 1) + \sqcap(j(\tau^j_{i,l}, dst(pc)), 1) \leq 1\end{aligned} \tag{A.19}$$

A.7.3 Résultats expérimentaux

Afin d'évaluer la capacité de notre approche à traiter des problèmes de grande taille, nous avons effectué une série d'expérimentations basées sur une application industrielle d'Airbus. L'objectif de ces expériences est multiple. En effet, nous voulons non seulement être capables de calculer automatiquement l'ordonnancement et le placement de l'application dans son budget mais également étudier l'impact de ce budget sur les performances de notre ordonnanceur et évaluer jusqu'à quel niveau de charge applicative notre méthode est applicable. Ainsi, nous explorons les solutions existantes dans un espace de 3 paramètres:

Charge applicative Afin de simuler une augmentation de charge applicative, nous intégrons un paramètre permettant de raccourcir les périodes des tâches sans changer les durées d'exécution des sous-tâches. Ce faisant, le ratio d'utilisation U augmente et la recherche d'un ordonnancement valide s'avère être de plus en plus difficile. Nous chercherons ici à déterminer

jusqu'à quel niveau de charge applicative nous arrivons à trouver des ordonnancements corrects. Nous réduirons artificiellement les périodes en jouant sur un paramètre k_{div} permettant de réduire la période des tâches avec $\widetilde{T_i} = \frac{256-k_{div}}{256} \times T_i$. Avec $k_{div} \in [0, 255]$, les périodes des applications $\widetilde{T_i} \in [T_i/256, T_i]$.

Réduction du budget Afin de comprendre l'impact de la modification d'un budget sur la performance de notre ordonnanceur, nous expérimentons avec plusieurs configurations de budgets qui sont des variations autour d'un budget originel. Tout d'abord, nous considérons trois configurations différentes de PNs. Avec n_{pn}^{orig} le nombre originel de PNs, nous étudierons des budgets avec des nombre de PNs variant suivant $n_{pn}^{orig} \leq |\mathcal{N}| \leq n_{pn}^{orig} + 2$. Cela nous permettra d'identifier si l'augmentation de la puissance de calcul allouée à la partition (plus de PNs veut dire plus de PEs) permet de supporter une charge applicative supérieure. Nous évaluerons également l'impact des modifications sur les PCs en faisant varier leur durée. Avec $C_{orig}(pc)$ la durée originelle d'un PC, nous testerons trois configurations avec des durées comprises dans $[C_{orig}(pc), C_{orig}(pc) + 2]$.

Configuration des DMAs Enfin, nous évaluons notre cas d'étude avec plusieurs configurations possibles de DMAs. Pour cela, nous faisons varier le nombre de zones de mémoire non contigües N_{bufs}^{DMA} qu'un DMA peut traiter de manière autonome (notamment utilisé dans contrainte A.13). Comme expliqué en Section A.6.1.2, le temps nécessaire pour la configuration du DMA par le RM dépend du nombre de zones à traiter. En réduisant celui-ci, nous réduisons l'autonomie du DMA mais réduisons également le WCET de l'hyperviseur qui peut ainsi être activé plus souvent. En augmentant N_{bufs}^{DMA}, le DMA sera plus autonome mais la période d'activation de l'hyperviseur devra être plus longue et permettra d'exprimer les durées et périodes des PCs à un grain moins fin. En évaluant plusieurs configurations de N_{bufs}^{DMA} comprises dans $[4, 32]$ nous cherchons à identifier le meilleur compromis pour le type d'application considérée.

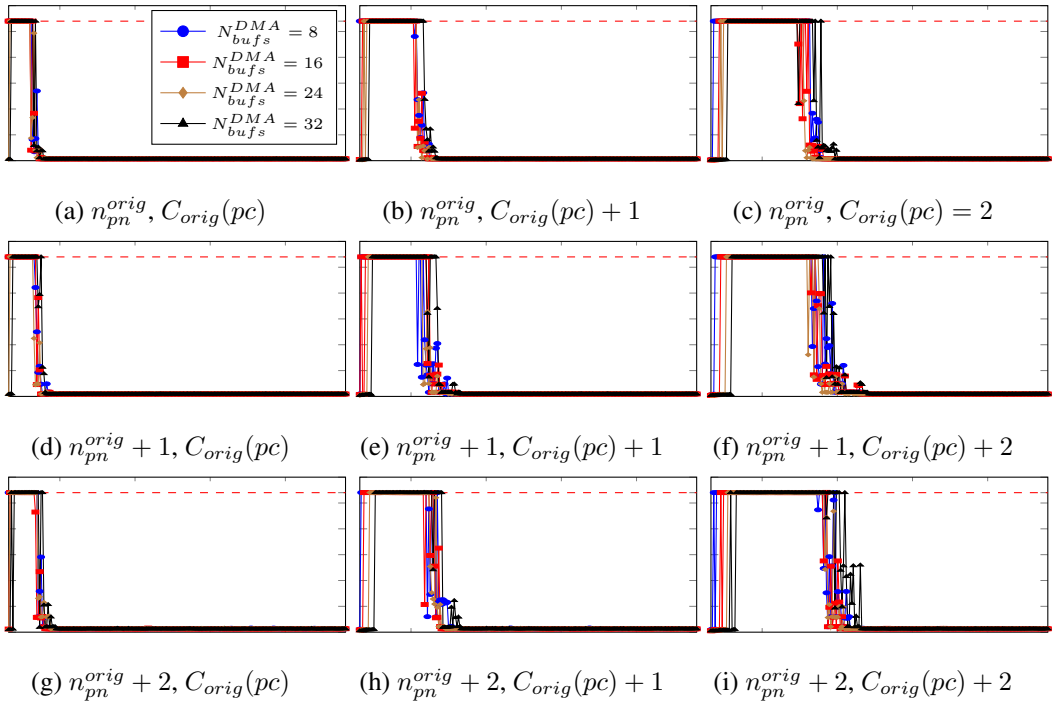
A.7.3.1 Description du cas d'étude

Notre cas d'étude est basé sur une application réelle d'Airbus de grande taille. Elle suit le modèle défini en Section A.2.1 et possède plusieurs tâches aux périodes harmoniques. Sur une hyperpériode, on dénombre environ 100 000 sous-jobs et données produites. Lorsque l'on considère cette application sous la forme d'intervalles, on arrive à plusieurs millions de variables de décisions et de contraintes à satisfaire.

Nous faisons les choix pour les budgets et donc pour l'implémentation de cette application:

1 PE par PN Pour toutes les expérimentations menées sur notre cas d'étude, nous forçons systématiquement le nombre de PEs alloués à chaque PN à être $N_c(pn) = 1$. Cela permet d'une part d'éliminer les interférences entre PEs de la même partition au niveau de la SRAM et donc de réduire les WCETs et d'autre part d'accentuer la nécessité d'une gestion fine du NoC pour la distribution sur plusieurs clusters. Bien que notre formulation du problème permette une parallélisation dans et entre les clusters de calculs, nous concentrons notre étude sur la gestion du NoC qui est le vrai défi nouveau sur une architecture pluri-cœurs.

PCs courts et symétriques Nous choisissons de relier l'ensemble des PNs de notre partition avec des PCs *courts* et *symétriques*. Court signifie que la durée $C(pc)$ est choisie la plus faible petite, c'est-à-dire, de l'ordre de grandeur de la période d'activation de l'hyperviseur. De ce fait, nous espérons réduire les délais NoC subis lors de précédences avec des données en



(a) $n_{pn}^{orig}, C_{orig}(pc)$ (b) $n_{pn}^{orig}, C_{orig}(pc) + 1$ (c) $n_{pn}^{orig}, C_{orig}(pc) = 2$

(d) $n_{pn}^{orig} + 1, C_{orig}(pc)$ (e) $n_{pn}^{orig} + 1, C_{orig}(pc) + 1$ (f) $n_{pn}^{orig} + 1, C_{orig}(pc) + 2$

(g) $n_{pn}^{orig} + 2, C_{orig}(pc)$ (h) $n_{pn}^{orig} + 2, C_{orig}(pc) + 1$ (i) $n_{pn}^{orig} + 2, C_{orig}(pc) + 2$

Figure A.23: Comparaison des temps de calcul en fonction de l'augmentation de la charge applicative, des 9 budgets et des configurations des DMAs

avant qui représentent le véritable coût d'une exécution distribuée. Symétrique signifie que, pour des raisons de simplifications, tous les PCs ont des durées et des périodes égales. Seuls les offsets changent pour éviter les conflits.

A.7.3.2 Analyse des résultats

La Figure A.23 montre les temps de calcul du solveur pour les 9 budgets différents et en fonction de la variation de charge applicative et des configurations DMA. L'abscisse des courbes représente la durée de l'hyper-période. Un point à droite sur une courbe correspond à un calcul avec une hyper-période longue et peu de charge, un point à gauche sur une courbe correspond à un calcul avec une hyper-période courte et donc beaucoup de charge. L'ordonnée indique le temps mis par le solveur pour trouver une solution. Passée une durée maximale de 10 800 secondes (la ligne pointillée horizontale rouge sur les courbes), la recherche de solution est abandonnée et le budget est considéré comme non valide. Les points à droites des courbes ayant une ordonnée faible indiquent qu'avec des charges faibles, des solutions sont trouvées rapidement. Le point le plus à gauche avant le plateau de saturation correspond à la charge maximale pour laquelle une solution a été trouvée. On observe que les meilleures performance sont obtenues avec une durée de PC minimale (courbes (a), (d) et (g)) pour laquelle les latences NoC sont les plus faibles. L'impact du nombre de PNs dans le budget est faible. Cela semble indiquer que le point limitant ici n'est pas simplement la ressource de calcul mais plutôt un goulot d'étranglement sur le NoC.

A.7.4 Synthèse

Dans cette section, nous avons détaillé comment l'étape de Validation peut être automatisée en utilisant la programmation par contraintes. Nous avons clarifié les hypothèses visant à recentrer notre étude sur l'aspect fondamental d'une architecture pluri-cœurs, à savoir la gestion du NoC. Nous avons formulé à l'aide de variables d'intervalle le problème de placement et d'ordonnancement et évalué cette formulation avec une application réelle d'Airbus. Ce résultat montre que la complexité d'une telle architecture de processeur et d'une grande application peut être traitée formellement, y compris avec des applications demandeuses de toujours plus de ressources. Il apparaît cependant que la gestion du NoC reste le goulot d'étranglement limitant les performances de l'approche. L'intégration du calcul du placement des données en mémoire avec celui de l'ordonnancement apparaît comme une bonne opportunité d'amélioration en ce sens.

A.8 Conclusion

D'une manière générale, nous avons adressé par ce travail le problème de l'exécution prédictible sur des processeurs pluri-cœurs. Nous avons proposé un atelier permettant l'intégration d'applications multiples sur de telles architectures. Il repose sur la propriété d'isolation temporelle apportée à des partitions par un modèle d'exécution que nous avons détaillé pour le KALRAY MPPA®-256. Le respect des règles énoncées par ce modèle est garanti en-ligne par un hyperviseur dont nous avons présenté la structure et les mécanismes. En outre, une validation expérimentale a permis de mettre en exergue la propriété d'isolation temporelle attendue sur un benchmark académique réaliste. Enfin, nous avons proposé une approche d'ordonnancement et de placement d'applications sur une architecture de processeur distribuée en tirant parti des fonctionnalités des solveurs modernes et notamment des notions de variable d'intervalle optionnelle et de fonction cumulative. Un cas d'étude réel fourni par Airbus nous a notamment permis de vérifier la capacité de passage à l'échelle sur des applications de grande taille.

L'approche proposée permet de résoudre correctement le problème initialement posé de l'exécution prédictible d'applications parallèles sur des processeurs pluri-cœurs tout en respectant les contraintes avioniques et industrielles posées. Certains points de notre approche nécessitent toutefois des travaux additionnels pour améliorer encore davantage les performances des applications ou pour étendre ce travail à d'autres domaines. Notre modèle d'exécution ainsi que son implémentation impliquent un pré-calcul important et une exécution relativement statique qui peut s'avérer contraignante hors du monde avionique avec des applications dynamiques. La mise en œuvre de mécanismes assouplissant les exécutions impliquerait des modifications relativement profondes de l'approche mais cela représente une opportunité de travail futur très intéressant. Par ailleurs, il semble que le calcul d'ordonnancement et de placement que nous proposons puisse être optimisé davantage en tenant compte des positionnements en mémoire des données. Une modification de notre formulation en ce sens impliquerait probablement une complexification du problème qu'il conviendrait d'évaluer. En outre le pré-calcul ou post-calcul des positionnements en mémoire sont des solutions qui font partie des chemins qui seraient enrichissants à explorer. Enfin, il est probable que de nouveaux types d'applications (d'apprentissage automatique par exemple) viennent à être utilisés massivement dans les systèmes embarqués à l'avenir. L'exécution de ces nouvelles charges de travail sur des architectures distribuées a déjà été étudiée à des échelles différentes, et notamment par des géants du web exploitant un grand nombre de serveurs en réseau. Le portage des techniques mises en œuvre à grandes échelles vers des systèmes embarqués distribués semble alors une opportunité de travail futur très intéressante. Les processeurs pluri-cœurs seraient dans ce contexte, grâce à leur capacité de calcul massivement parallèle, une nouvelle fois de bons candidats

pour la conception de ces systèmes avioniques futurs.

Bibliography

[1] United States General Accounting Office. *PATRIOT MISSILE DEFENSE - Software Problem Led to System Failure at Dhahran, Saudi Arabia*. 1992 (cited on page 19).

[2] Airbus Defence and Space. *Press release: Statement regarding Accident Information Transmission (AIT) to A400M operators as follow up to AOT of 19 May*. 2015 (cited on page 20).

[3] SAE International. *Aerospace Recommended Practices 4754a - Development of Civil Aircraft and Systems*. 2010 (cited on page 20).

[4] Radio Technical Commission for Aeronautics (RTCA) and EURopean Organisation for Civil Aviation Equipment (EUROCAE). *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. 2011 (cited on pages 20, 39, 155).

[5] SAE International. *Aerospace Recommended Practices 4761 - Guidelines And Methods For Conducting The Safety Assessment Process On Civil Airborne Systems And Equipment*. 1996 (cited on page 20).

[6] Gordon Moore. "Progress in Digital Integrated Electronics". In: *IEEE, IEDM Tech Digest* (1975), pp. 11–13 (cited on page 22).

[7] Shekhar Borkar and Andrew A. Chien. "The Future of Microprocessors". In: *Communications of the ACM* 54.5 (2011), pp. 67–77 (cited on pages 23, 48).

[8] Samuel H. Fuller and Lynette I. Millett. *The Future of Computing Performance: Game Over Or Next Level?* National Academy Press, 2011 (cited on page 23).

[9] Michael Bedford Taylor. "Tiled microprocessors". PhD thesis. Massachusetts Institute of Technology, 2007 (cited on pages 23, 48).

[10] Intel Corporation. *Intel Xeon Phi Coprocessor x100 Product Family - Datasheet*. 2015 (cited on pages 24, 156).

[11] Adapteva. *E16G301 Epiphany 16-core Microprocessor - Datasheet* (cited on pages 24, 156).

[12] Khronos OpenCL Working Group. *The OpenCL Specification - Version: 2.2 - Revision: 06*. 2016 (cited on page 24).

[13] NVidia. *CUDA C Programming Guide - Design guide*. PG-02829-001_v8.0. 2016 (cited on page 24).

[14] Intel Labs. *SCC External Architecture Specification (EAS) - Revision 0.94*. 2010 (cited on pages 24, 48, 80, 81, 156, 167).

[15] EZchip. *TILE-Gx72 Processor - Product Brief* (cited on pages 24, 48, 156).

[16] Bob Doud. *Accelerating the Data Plane With the TILE-Mx Manycore Processor*. Linley Data Center Conference. 2015 (cited on pages 24, 156).

[17] Kalray Corporation. *The MPPA hardware architecture*. 2012 (cited on pages 24, 29, 66, 89, 156, 160).

[18] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. "The Worst-case Execution-time Problem - Overview of Methods and Survey of Tools". In: *ACM Transactions Embedded Computing Systems* 7.3 (2008), 36:1–36:53 (cited on pages 39, 44, 45).

[19] Reinhard Wilhelm and Jan Reineke. "Embedded systems: Many cores - Many problems". In: *7th IEEE International Symposium on Industrial Embedded Systems (SIES'12)*. 2012, pp. 176–180 (cited on pages 39, 46).

[20] Reinhard Wilhelm, Daniel Grund, Jan Reineke, Marc Schlickling, Markus Pister, and Christian Ferdinand. "Memory Hierarchies, Pipelines, and Buses for Future Architectures in Time-Critical Embedded Systems". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 28.7 (2009), pp. 966–978 (cited on pages 45, 63).

[21] Thomas Lundqvist and Per Stenstrom. "Timing anomalies in dynamically scheduled microprocessors". In: *20th IEEE Real-Time Systems Symposium (RTSS'99)*. 1999, pp. 12–21 (cited on page 45).

[22] Yau-Tsun Steven Li and Sharad Malik. "Performance Analysis of Embedded Software Using Implicit Path Enumeration". In: *32nd Annual ACM/IEEE Design Automation Conference (DAC'95)*. 1995, pp. 456–461 (cited on page 45).

[23] AbsInt Angewandte Informatik GmbH. *aiT WCET Analyzers*. URL: `https://www.absint.com/ait/` (cited on page 45).

[24] Tidorum Ltd. *Bound-T time and stack analyser*. URL: `http://www.bound-t.com/` (cited on page 45).

[25] IRIT University of Toulouse. *Open Tool for Adaptive WCET Analyses (OTAWA)*. URL: `http://www.otawa.fr/` (cited on page 45).

[26] INRIA Rennes. *Heptane static WCET estimation tool*. URL: `https://team.inria.fr/alf/software/heptane/` (cited on page 45).

[27] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. "PathCrawler: Automatic Generation of Path Tests by Combining Static and Dynamic Analysis". In: *5th European Dependable Computing Conference (EDCC'05)*. Springer Berlin Heidelberg, 2005, pp. 281–292 (cited on page 46).

[28] Thomas Lundqvist and Per Stenström. "An Integrated Path and Timing Analysis Method Based on Cycle-Level Symbolic Execution". In: *Real-Time Systems* 17.2-3 (1999), pp. 183–207 (cited on page 46).

[29] Stephen Law and Iain Bate. " Achieving Appropriate Test Coverage for Reliable Measurement-based Timing Analysis ". In: *28th Euromicro Conference on Real-Time Systems (ECRTS'16)*. 2016 (cited on page 46).

[30] Rapita Systems Ltd. *RapiTime*. URL: `https://www.rapitasystems.com/products/rapitime` (cited on page 46).

[31] Steward Edgar and Alan Burns. "Statistical analysis of WCET for scheduling". In: *2nd IEEE Real-Time Systems Symposium (RTSS'01)*. 2001, pp. 215–224 (cited on page 46).

[32] Paul Embrechts, Claudia Klüppelberg, and Thomas Mikosch. *Modelling Extremal Events for Insurance and Finance*. Vol. 33. Springer-Verlag Berlin Heidelberg, 1997 (cited on page 46).

[33] Yue Lu, Thomas Nolte, Ian Bate, and Liliana Cucu-Grosjean. "A trace-based statistical worst-case execution time analysis of component-based real-time embedded systems". In: *2011 IEEE 16th Conference on Emerging Technologies Factory Automation (ETFA'11)*. 2011, pp. 1–4 (cited on page 46).

[34] *Probabilistically Analysable Real-Time Systems (PROARTIS)*. http://www.proartis-project.eu/ (cited on page 46).

[35] *Probabilistic real-time control of mixed-criticality multicore and manycore systems (PROXIMA)*. http://www.proxima-project.eu/ (cited on page 46).

[36] Kostiantyn Berezovskyi, Fabrice Guet, Luca Santinelli, Konstantinos Bletsas, and Eduardo Tovar. "Measurement-Based Probabilistic Timing Analysis for Graphics Processor Units". In: *29th International Conference on Architecture of Computing Systems (ARCS'16)*. 2016, pp. 223–236 (cited on page 46).

[37] Fabrice Guet, Luca Santinelli, and Jérôme Morio. "Probabilistic analysis of cache memories and cache memories impacts on multi-core embedded systems". In: *11th IEEE Symposium on Industrial Embedded Systems, (SIES'16)*. 2016, pp. 131–140 (cited on page 46).

[38] Sebastian Hahn, Jan Reineke, and Reinhard Wilhelm. "Towards Compositionality in Execution Time Analysis: Definition and Challenges". In: *SIGBED Review* 12.1 (2015), pp. 28–36 (cited on page 47).

[39] Pavel Atanassov and Peter Puschner. "Impact of DRAM Refresh on the Execution Time of Real-Time Tasks". In: *IEEE International Workshop on Application of Reliable Computing and Communication*. 2001, pp. 29–34 (cited on pages 47, 59).

[40] Parviz Kermani and Leonard Kleinrock. "Virtual cut-through: a new computer communication switching technique". In: *Computer Networks* 3 (1979), pp. 267–286 (cited on page 48).

[41] Charles L. Dally William J.and Seitz. "The torus routing chip". In: *Distributed Computing* 1.4 (1986), pp. 187–196 (cited on page 48).

[42] Lionel M. Ni and Philip K. McKinley. "A Survey of Wormhole Routing Techniques in Direct Networks". In: *Computer* 26.2 (1993), pp. 62–76 (cited on page 48).

[43] John L. Hennessy and David A. Patterson. *Computer Architecture, Fifth Edition: A Quantitative Approach*. 5th. Morgan Kaufmann Publishers Inc., 2011 (cited on page 49).

[44] Zhonghai Lu, Axel Jantsch, and Ingo Sander. "Feasibility Analysis of Messages for On-chip Networks Using Wormhole Routing". In: *2005 Asia and South Pacific Design Automation Conference (ASP-DAC'05)*. 2005, pp. 960–964 (cited on page 49).

[45] Yue Qian, Zhonghai Lu, and Wenhua Dou. "Analysis of Worst-Case Delay Bounds for On-Chip Packet-Switching Networks". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 29.5 (2010), pp. 802–815 (cited on pages 49, 58).

[46] Benoit Dupont De Dinechin, Duco van Amstel, Marc Poulhies, and G. Lager. "Time-critical computing on a single-chip massively parallel processor". In: *18th Design, Automation & Test in Europe Conference and Exhibition (DATE'14)*. 2014, pp. 1–6 (cited on pages 49, 58, 87, 91, 102, 170, 176).

[47] 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* (2015), pp. 1–51 (cited on pages 49, 58, 78, 79, 87, 91, 102, 165, 170, 176).

[48] JEDEC. *DDR3 SDRAM STANDARD*. 2012 (cited on page 49).

[49] Micron. *4Gb: x4, x8, x16 DDR3L SDRAM Description*. 2011 (cited on page 53).

[50] José Duato. “A New Theory of Deadlock-Free Adaptive Routing in Wormhole Networks”. In: *IEEE Transactions on Parallel and Distributed Systems* 4.12 (1993), pp. 1320–1331 (cited on pages 55, 56).

[51] Éric Fleury and Pierre Fraigniaud. “A General Theory for Deadlock Avoidance in Wormhole-Routed Networks”. In: *IEEE Transactions on Parallel and Distributed Systems* 9.7 (1998), pp. 626–638 (cited on pages 55, 56).

[52] Charles L. Dally William J.and Seitz. “Deadlock-Free Message Routing in Multiprocessor Interconnection Networks”. In: *IEEE Transactions on Computers* 36.5 (1987), pp. 547–553 (cited on page 56).

[53] José Duato. “A Necessary and Sufficient Condition for Deadlock-Free Adaptive Routing in Wormhole Networks”. In: *IEEE Transactions Parallel Distributed Systems* 6.10 (1995), pp. 1055–1067 (cited on page 56).

[54] D. N. Jayasimha, Loren Schwiebert, D. Manivannan, and Jeff A. May. “A Foundation for Designing Deadlock-free Routing Algorithms in Wormhole Networks”. In: *Journal of the ACM* 50.2 (2003), pp. 250–275 (cited on page 56).

[55] Xiaola Lin and Lionel M. Ni. “Deadlock-free Multicast Wormhole Routing in Multicomputer Networks”. In: *18th Annual International Symposium on Computer Architecture (ISCA’91)*. New York, NY, USA: ACM, 1991, pp. 116–125 (cited on page 56).

[56] Loren Schwiebert and D. N. Jayasimha. “A Universal Proof Technique for Deadlock-Free Routing in Interconnection Networks”. In: *7th Annual ACM Symposium on Parallel Algorithms and Architectures*. 1995, pp. 175–184 (cited on page 56).

[57] Benoît Dupont de Dinechin, Yves Durand, Duco van Amstel, and Alexandre Ghiti. “Guaranteed Services of the NoC of a Manycore Processor”. In: *2014 International Workshop on Network on Chip Architectures (NoCArc’14)*. 2014, pp. 11–16 (cited on pages 56, 167).

[58] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*. Springer-Verlag, 2001 (cited on pages 57, 166, 170).

[59] R.L. Cruz. “A calculus for network delay. I. Network elements in isolation”. In: *IEEE Transactions on Information Theory* 37.1 (1991), pp. 114–131 (cited on pages 57, 58, 63, 166, 170).

[60] L. Thiele, S. Chakraborty, and M. Naedele. “Real-time calculus for scheduling hard real-time systems”. In: *2000 International Symposium on Circuits and Systems (ISCAS’00)*. Vol. 4. 2000, pp. 101–104 (cited on page 57).

[61] Yue Qian, Zhonghai Lu, and Wenhua Dou. “Analysis of Communication Delay Bounds for Network on Chips”. In: *2009 Asia and South Pacific Design Automation Conference (ASP-DAC’09)*. 2009, pp. 7–12 (cited on page 58).

[62] Jia Zhan, Nikolay Stoimenov, Jin Ouyang, Lothar Thiele, Vijaykrishnan Narayanan, and Yuan Xie. “Designing Energy-efficient NoC for Real-time Embedded Systems Through Slack Optimization”. In: *50th Annual Design Automation Conference (DAC’13)*. 2013, 37:1–37:6 (cited on page 58).

[63] Hui Zhang. “Service disciplines for guaranteed performance service in packet-switching networks”. In: *Proceedings of the IEEE* 83.10 (1995), pp. 1374–1396 (cited on page 58).

[64] Zheng Pei Wu, Yogen Krish, and Rodolfo Pellizzoni. “Worst Case Analysis of DRAM Latency in Multi-requestor Systems”. In: *34th Real-Time Systems Symposium (RTSS’13)*. 2013, pp. 372–383 (cited on page 59).

[65] Yiqiang Ding, Lan Wu, and Wei Zhang. “Bounding Worst-Case DRAM Performance on Multicore Processors”. In: *Journal of Computer Science and Engineering* 7.1 (2013), pp. 53–66 (cited on page 59).

[66] Scott Rixner, William J. Dally, Ujval J. Kapasi, Peter Mattson, and John D. Owens. “Memory Access Scheduling”. In: *SIGARCH Computer Architecture News* 28.2 (2000), pp. 128–138 (cited on page 59).

[67] Hyoseung Kim, Dionisio de Niz, Björn Andersson, Mark Klein, Onur Mutlu, and Ragunathan Raj Rajkumar. “Bounding memory interference delay in COTS-based multi-core systems”. In: *20th Real-Time and Embedded Technology and Applications Symposium (RTAS’14)*. 2014, pp. 145–154 (cited on page 59).

[68] Heechul Yun, Rodolfo Pellizzoni, and Prathap Kumar Valsan. “Parallelism-Aware Memory Interference Delay Analysis for COTS Multicore Systems ”. In: *27th Euromicro Conference on Real-Time Systems (ECRTS’15)*. 2015, pp. 184–195 (cited on page 59).

[69] Niladrish Chatterjee, Naveen Muralimanohar, Rajeev Balasubramonian, Al Davis, and Norman P. Jouppi. “Staged Reads: Mitigating the Impact of DRAM Writes on DRAM Reads”. In: *18th International Symposium on High-Performance Computer Architecture (HPCA’12)*. IEEE Computer Society, 2012, pp. 1–12 (cited on page 59).

[70] B. Bhat and F. Mueller. “Making DRAM Refresh Predictable”. In: *22nd Euromicro Conference on Real-Time Systems (ECRTS’10)*. 2010, pp. 145–154 (cited on page 59).

[71] Quentin Perret, Pascal Maurère, Éric Noulard, Claire Pagetti, Pascal Sainrat, and Benoit Triquet. “Predictable composition of memory accesses on many-core processors”. In: *8th Conference on Embedded Real Time Software and Systems (ERTS’16)*. 2016 (cited on pages 60, 85, 93, 146).

[72] Kees Goossens, Arnaldo Azevedo, Karthik Chandrasekar, Manil Dev Gomony, Sven Goossens, Martijn Koedam, Yonghui Li, Davit Mirzoyan, Anca Molnos, Ashkan Beyranvand Nejad, Andrew Nelson, and Shubhendu Sinha. “Virtual Execution Platforms for Mixed-Time-Criticality Systems: The CompSOC Architecture and Design Flow ”. In: *ACM SIGBED* 10.3 (2013) (cited on pages 61–63).

[73] E. Kasapaki and J. SparsØ. “Argo: A Time-Elastic Time-Division-Multiplexed NOC Using Asynchronous Routers”. In: *20th IEEE International Symposium on Asynchronous Circuits and Systems (ASYNC’14)*. 2014, pp. 45–52 (cited on pages 61, 63).

[74] Y. Krishnapillai, Zheng Pei Wu, and R. Pellizzoni. “A Rank-Switching, Open-Row DRAM Controller for Time-Predictable Systems”. In: *26th Euromicro Conference on Real-Time Systems (ECRTS’14)*. 2014, pp. 27–38 (cited on pages 61, 64).

[75] Stephen A. Edwards and Edward A. Lee. “The Case for the Precision Timed (PRET) Machine”. In: *44th Annual Design Automation Conference (DAC’07)*. New York, NY, USA: ACM, 2007, pp. 264–265 (cited on page 61).

[76] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee. “Predictable Programming on a Precision Timed Architecture”. In: *2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES’08)*. New York, NY, USA: ACM, 2008, pp. 137–146 (cited on page 61).

[77] SPARC International Inc. *The SPARC Architecture Manual - Version 8* (cited on page 61).

[78] Blair Fort, Davor Capalija, Zvonko Vranesic, and Stephen D. Brown. “A Multithreaded Soft Processor for SoPC Area Reduction ”. In: *IEEE International Symposium on Custom Computing Machines*. 2006, pp. 131–142 (cited on page 61).

[79] Martin Labrecque and J. Gregory Steffan. “Improving Pipelined Soft Processors with Multithreading”. In: *2007 International Conference on Field Programmable Logic and Applications*. 2007, pp. 210–215 (cited on page 61).

[80] Nicholas Jun Hao Ip and Stephen A. Edwards. “A Processor Extension for Cycle-Accurate Real-Time Software”. In: *International Conference Embedded and Ubiquitous Computing*. 2006 (cited on page 61).

[81] Luca Benini, Alberto Macii, Enrico Macii, and Massimo Poncino. “Increasing Energy Efficiency of Embedded Systems by Application-Specific Memory Hierarchy Generation”. In: *IEEE Desgin and Test* 17.2 (2000), pp. 74–85 (cited on page 61).

[82] Michael Zimmer, David Broman, Chris Shaver, and Edward A. Lee. “FlexPRET: A Processor Platform for Mixed-Criticality Systems”. In: *20th IEEE Real-Time and Embedded Technology and Application Symposium (RTAS’14)*. 2014 (cited on page 61).

[83] Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanovic. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.0*. Tech. rep. Electrical Engineering and Computer Sciences - University of California at Berkeley, 2014 (cited on page 61).

[84] Martin Schoeberl, Sahar Abbaspour, Benny Akesson, Neil Audsley, Raffaele Capasso, Jamie Garside, Kees Goossens, Sven Goossens, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Alexander Jordan, Evangelia Kasapaki, Jens Knoop, Yonghui Li, Daniel Prokesch, Wolfgang Puffitsch, Peter Puschner, André Rocha, Cláudio Silva, Jens Sparsø, and Alessandro Tocchi. “T-CREST: Time-predictable Multi-Core Architecture for Embedded Systems ”. In: *Journal of Systems Architecture* 61.9 (2015), pp. 449–471 (cited on pages 61, 62).

[85] Martin Schoeberl, Pascal Schleuniger, Wolfgang Puffitsch, Florian Brandner, Christian W. Probst, Sven Karlsson, and Tommy Thorn. “Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach”. In: *First Workshop on Bringing Theory to Practice: Predictability and Performance in Embedded Systems (PPES’11)*. Grenoble, France, 2011, pp. 11–20 (cited on page 61).

[86] Martin Schoeberl. “Time-predictable Cache Organization”. In: *Firstr International Workshop on Software Technologies for Future Dependable Distributed Systems (STFSSD’2009)*. 2009, pp. 11–16 (cited on page 62).

[87] T. Ungerer, F. Cazorla, P. Sainrat, G. Bernat, Z. Petrov, C. Rochange, E. Quinones, M. Gerdes, M. Paolieri, J. Wolf, H. Casse, S. Uhrig, I. Guliashvili, M. Houston, F. Kluge, S. Metzlaff, and J. Mische. "Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability". In: *IEEE Micro* 30.5 (2010), pp. 66–75 (cited on page 62).

[88] T. Ungerer, C. Bradatsch, M. Gerdes, F. Kluge, R. Jahr, J. Mische, J. Fernandes, P. G. Zaykov, Z. Petrov, B. Böddeker, S. Kehr, H. Regler, A. Hugl, C. Rochange, H. Ozaktas, H. Cassé, A. Bonenfant, P. Sainrat, I. Broster, N. Lay, D. George, E. Quiñones, M. Panic, J. Abella, F. Cazorla, S. Uhrig, M. Rohde, and A. Pyka. "parMERASA – Multi-core Execution of Parallelised Hard Real-Time Applications Supporting Analysability". In: *2013 Euromicro Conference on Digital System Design (DSD'13)*. 2013, pp. 363–370 (cited on page 62).

[89] Infineon. *TriCore 1, 32-bit Unified processor Core*. 2008 (cited on page 62).

[90] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. "Hardware Support for WCET Analysis of Hard Real-time Multicore Systems". In: *SIGARCH Computer Architecture News* 37.3 (2009), pp. 57–68 (cited on page 62).

[91] 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* 14.1 (2009), 2:1–2:24 (cited on page 62).

[92] K. Goossens, J. Dielissen, and A. Radulescu. "AEthereal network on chip: concepts, architectures, and implementations". In: *IEEE Design & Test of Computers* 22.5 (2005), pp. 414–421 (cited on page 63).

[93] Benny Akesson, Kees Goossens, and Markus Ringhofer. "PREDATOR: A Predictable SDRAM Memory Controller". In: *5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07)*. 2007, pp. 251–256 (cited on page 63).

[94] Bruce Jacob, Spencer Ng, and David Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., 2007 (cited on page 63).

[95] Benny Akesson, Liesbeth Steffens, Eelke Strooisma, and Kees Goossens. *Real-Time Scheduling of Hybrid Systems using Credit-Controlled Static-Priority Arbitration*. Tech. rep. NXP Semiconductors, 2007 (cited on page 63).

[96] Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, and Mateo Valero. "An Analyzable Memory Controller for Hard Real-Time CMPs". In: *IEEE Embedded Systems Letters* 1.4 (2009), pp. 86–90 (cited on page 63).

[97] Jan Reineke, Isaac Liu, Hiren D. Patel, Sungjun Kim, and Edward A. Lee. "PRET DRAM Controller: Bank Privatization for Predictability and Temporal Isolation". In: *7th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'11)*. 2011, pp. 99–108 (cited on page 64).

[98] Leonardo Ecco and Rolf Ernst. "Improved DRAM Timing Bounds for Real-Time DRAM Controllers with Read/Write Bundling". In: *IEEE Real-Time Systems Symposium (RTSS'15)*. 2015, pp. 53–64 (cited on page 64).

[99] Stuart Fisher. *Certifying Applications in a Multi-Core Environment: The World's First Multi-Core Certification to SIL 4*. 2014 (cited on pages 64, 164).

[100] International Electrotechnical Commission (IEC). *IEC 61508 : Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems* (cited on page 64).

[101] Xavier Jean, Daniel Faura, Marc Gatti, Laurent Pautet, and Thomas Robert. “Ensuring robust partitioning in multicore platforms for IMA systems”. In: *31st IEEE/AIAA Digital Avionics Systems Conference (DASC’12)*. 2012, 7A4–1–7A4–9 (cited on pages 65, 66, 164).

[102] Xavier Jean. “ Maîtrise de la couche hyperviseur sur les processeurs multi-coeurs COTS dans un contexte avionique ”. PhD thesis. Telecom ParisTech, 2015 (cited on pages 65, 164).

[103] Heechul Yun, Gang Yao, R. Pellizzoni, M. Caccamo, and Lui Sha. “MemGuard: Memory bandwidth reservation system for efficient performance isolation in multi-core platforms”. In: *19th Real-Time and Embedded Technology and Applications Symposium (RTAS’13)*. 2013, pp. 55–64 (cited on pages 65, 67, 165).

[104] Adam Kostrzewa, Sebastian Tobuschat, Rolf Ernst, and Selma Saidi. “Safe and dynamic traffic rate control for networks-on-chips”. In: *2016 Tenth IEEE/ACM International Symposium on Networks-on-Chip (NOCS’16)*. 2016, pp. 1–8 (cited on page 66).

[105] 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: *17th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’11)*. 2011, pp. 269–279 (cited on pages 66, 165).

[106] Emiliano Betti, Stanley Bak, Rodolfo Pellizzoni, Marco Caccamo, and Lui Sha. “Real-Time I/O Management System with COTS Peripherals”. In: *IEEE Transactions on Computers* 62.1 (2013), pp. 45–58 (cited on page 67).

[107] Guy Durrieu, Madeleine Faugère, Sylvain Girbal, Daniel Gracia Pérez, Claire Pagetti, and Wolfgang Puffitsch. “Predictable Flight Management System Implementation on a Multicore Processor”. In: *7th Conference on Embedded Real Time Software and Systems (ERTS’14)*. 2014 (cited on pages 67, 165).

[108] Texas Instruments. *TMS320C6678 - Multicore Fixed and Floating-Point Digital Signal Processor*. 2014 (cited on pages 67, 80, 165, 167).

[109] Victor Jegu, Benoît Triquet, Frédéric Aspro, Frédéric. Boniol, and Claire Pagetti. *Method and device for loading and executing instructions with deterministic cycles in a multicore avionic system having a bus of which the access time is not predictable*. 2012. URL: `http://www.google.com/patents/US20120084525` (cited on pages 68, 165).

[110] Angeliki Kritikakou, Claire Pagetti, Matthieu Roy, Christine Rochange, Madeleine Faugère, Sylvain Girbal, and Daniel Gracia Pérez. “ Distributed run-time WCET controller for concurrent critical tasks in mixed-critical systems ”. In: *22nd International Conference on Real-Time Networks and Systems (RTNS’14)*. 2014 (cited on page 68).

[111] Rohan Tabish, Reanto Mancuso, Saud Wasly, Ahmed Alhammad, Sujit S. Phatak, Rodolfo Pellizzoni, and Marco Caccamo. “A Real-Time Scratchpad-Centric OS for Multi-Core Embedded Systems”. In: *22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’16)*. 2016, pp. 1–11 (cited on pages 68, 165).

[112] NXP Semiconductors. *MPC5777M Microcontroller Data Sheet*. 2016 (cited on page 68).

[113] Sanjoy Baruah, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Leen Stougie, and Andreas Wiese. “A generalized parallel task model for recurrent real-time processes”. In: *33rd IEEE Real-Time Systems Symposium (RTSS’12)*. 2012, pp. 63–72 (cited on pages 71, 75).

[114] Karthik Lakshmanan, Shinpei Kato, and Ragunathan (Raj) Rajkumar. “Scheduling Parallel Real-Time Tasks on Multi-core Processors”. In: *31st IEEE Real-Time Systems Symposium (RTSS’10)*. 2010, pp. 259–268 (cited on page 71).

[115] Abusayeed Saifullah, Kunal Agrawal, Chenyang Lu, and Christopher Gill. “Multi-core Real-Time Scheduling for Generalized Parallel Task Models”. In: *32nd IEEE Real-Time Systems Symposium (RTSS’11)*. 2011, pp. 217–226 (cited on page 71).

[116] Jorgen Bang-Jensen and Gregory Z. Gutin. *Digraphs: Theory, Algorithms and Applications*. 2nd. Vol. 2.1 Acyclic Digraphs. Springer Publishing Company, Incorporated, 2008 (cited on page 72).

[117] Mark D. Hill and Michael R. Marty. “Amdahl’s Law in the Multicore Era”. In: *IEEE Computer* 41.7 (2008), pp. 33–38 (cited on page 73).

[118] Houssine Chetto, Maryline Silly, and T. Bouchentouf. “Dynamic scheduling of real-time tasks under precedence constraints”. In: *Real-Time Systems* 2.3 (1990), pp. 181–194 (cited on pages 73–75).

[119] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. “Complexity of Machine Scheduling Problems”. In: *Studies in Integer Programming*. Ed. by B.H. Korte "P.L. Hammer E.L. Johnson and G.L. Nemhauser. Vol. 1. Annals of Discrete Mathematics. Elsevier, 1977, pp. 343 –362 (cited on page 74).

[120] Joseph Y-T Leung and Jennifer Whitehead. “On the complexity of fixed-priority scheduling of periodic, real-time tasks”. In: *Performance evaluation* 2.4 (1982), pp. 237–250 (cited on page 74).

[121] J.D. Ullman. “NP-complete scheduling problems”. In: *Journal of Computer and System Sciences* 10.3 (1975), pp. 384 –393 (cited on page 74).

[122] Manar Qamhieh, Frédéric Fauberteau, Laurent George, and Serge Midonnet. “Global EDF Scheduling of Directed Acyclic Graphs on Multiprocessor Systems”. In: *21st International Conference on Real-Time Networks and Systems (RTNS’13)*. 2013, pp. 287–296 (cited on page 74).

[123] Manar Qamhieh, Laurent George, and Serge Midonnet. “A Stretching Algorithm for Parallel Real-time DAG Tasks on Multiprocessor Systems”. In: *22nd International Conference on Real-Time Networks and Systems (RTNS’14)*. Oct. 2014, pp. 13–22 (cited on page 74).

[124] Jing Li, Kunal Agrawal, Chenyang Lu, and Christopher Gill. “ Analysis of Global EDF for Parallel Tasks ”. In: *25th Euromicro Conference on Real-Time Systems (ECRTS’13)*. 2013 (cited on page 75).

[125] Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Sebastian Stiller, and Andreas Wiese. “Feasibility Analysis in the Sporadic DAG Task Model”. In: *25th Euromicro Conference on Real-Time Systems (ECRTS’13)*. 2013, pp. 225–233 (cited on page 75).

[126] Emmanuel Grolleau and Annie Choquet-Geniet. “Ordonnancement de tâches temps réel en environnement multiprocesseur à l’aide de réseaux de Petri”. In: *Real-Time Systems (RTS’01)*. 2001 (cited on page 75).

[127] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. “Optimal Scheduling Using Priced Timed Automata”. In: *ACM SIGMETRICS Performance Evaluation Review* 32.4 (Mar. 2005), pp. 34–40 (cited on page 75).

[128] Julie Baro, Frédéric Boniol, Mikel Cordovilla, Eric Noulard, and Claire Pagetti. “Off-line (Optimal) Multiprocessor Scheduling of Dependent Periodic Tasks”. In: *27th Annual ACM Symposium on Applied Computing (SAC’12)*. 2012, pp. 1815–1820 (cited on page 75).

[129] Frédéric Boniol, Pierre-Emmanuel Hladik, Claire Pagetti, Frédéric Aspro, and Victor Jégu. “A Framework for Distributing Real-Time Functions”. In: *6th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS’08)*. 2008, pp. 155–169 (cited on page 75).

[130] Yu-Kwong Kwok and Ishfaq Ahmad. “Static Scheduling Algorithms for Allocating Directed Task Graphs to Multiprocessors”. In: *ACM Comput. Surv.* 31.4 (Dec. 1999), pp. 406–471 (cited on page 76).

[131] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. New York, NY, USA: W. H. Freeman & Co., 1979 (cited on page 76).

[132] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. “A Comparison of List Schedules for Parallel Processing Systems”. In: *Commun. ACM* 17.12 (1974), pp. 685–690 (cited on pages 76, 77).

[133] I. Ahmad, Yu-Kwong Kwok, and Min-You Wu. “Analysis, evaluation, and comparison of algorithms for scheduling task graphs on parallel processors”. In: *2nd International Symposium on Parallel Architectures, Algorithms, and Networks*. 1996, pp. 207–213 (cited on page 76).

[134] Thomas L. Casavant and Jon G. Kuhl. “A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems”. In: *IEEE Transactions on Software Engineering* 14.2 (1988), pp. 141–154 (cited on page 76).

[135] Hesham El-Rewini, Theodore G. Lewis, and Hesham H. Ali. *Task Scheduling in Parallel and Distributed Systems*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994 (cited on page 76).

[136] Apostolos Gerasoulis and Tao Yang. “A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors”. In: *Journal of Parallel and Distributed Computing* 16.4 (1992), pp. 276 –291 (cited on page 76).

[137] Behrooz Shirazi, Mingfang Wang, and Girish Pathak. “Analysis and Evaluation of Heuristic Methods for Static Task Scheduling”. In: *Journal of Parallel and Distributed Computing* 10.3 (Oct. 1990), pp. 222–2232 (cited on page 76).

[138] C. L. McCreary, A. A. Khan, J. J. Thompson, and M. E. McArdle. “A comparison of heuristics for scheduling DAGs on multiprocessors”. In: *8th International Parallel Processing Symposium*. 1994, pp. 446–451 (cited on page 76).

[139] C. Q. Yang and B. P. Miller. “Critical path analysis for the execution of parallel and distributed programs”. In: *8th International Conference on Distributed Computing Systems*. 1988, pp. 366–373 (cited on page 76).

[140] E.G. Coffman and J.L. Bruno. *Computer and job-shop scheduling theory*. A Wiley-Interscience publication. Wiley, 1976 (cited on pages 76, 77).

[141] Donald K. Friesen. “Tighter Bounds for LPT Scheduling on Uniform Processors”. In: *SIAM J. Comput.* 16.3 (June 1987), pp. 554–560 (cited on page 77).

[142] R.L. Graham, A.H.G.R. Kan, and Erasmus University Rotterdam. Econometric Institute. *Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey*. Reprint series / Erasmus University Rotterdam. Erasmus University, 1979 (cited on page 77).

[143] B. Kruatrachue and T. G. Lewis. *Duplication Scheduling heuristic, A New Precedence Task Scheduling for parallel System*. Tech. rep. 1987 (cited on page 77).

[144] Min-you Wu and Daniel D. Gajski. "Hypertool: A Programming Aid for Message-Passing Systems". In: *IEEE Transactions on Parallel and Distributed Systems* 1 (1990), pp. 330–343 (cited on page 77).

[145] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. "Scheduling of Mixed-criticality Applications on Resource-sharing Multicore Systems". In: *11th ACM International Conference on Embedded Software (EMSOFT'13)*. 2013, 17:1–17:15 (cited on pages 78, 166).

[146] Matthias Becker, Dakshina Dasari, Borislav Nicolic, Benny Åkesson, Vincent Nélis, and Thomas Nolte. "Contention-Free Execution of Automotive Applications on a Clustered Many-Core Platform". In: *28th Euromicro Conference on Real-Time Systems (ECRTS'16)*. 2016 (cited on pages 79, 80, 123, 124, 166, 183).

[147] Thomas Carle, Manel Djemal, Dumitru Potop-Butucaru, and Robert De Simone. "Static mapping of real-time applications onto massively parallel processor arrays". In: *14th International Conference on Application of Concurrency to System Design (ACSD'14)*. 2014, pp. 112–121 (cited on pages 80, 166).

[148] Wolfgang Puffitsch, Éric Noulard, and Claire Pagetti. "Off-line mapping of multi-rate dependent task sets to many-core platforms". In: *Real-Time Systems* 51.5 (2015), pp. 526–565 (cited on pages 80, 81, 124, 167).

[149] EZchip. *TILE-Gx36 Processor - Product Brief* (cited on pages 80, 167).

[150] Silviu S. Craciunas and Ramon Serna Oliver. "Combined task- and network-level scheduling for distributed time-triggered systems". In: *Real-Time Systems* 52.2 (2016), pp. 161–200 (cited on pages 81, 82).

[151] SAE International. *AS6802: Time-Triggered Ethernet* (cited on page 82).

[152] Wolfgang Puffitsch, Rasmus Bo Sørensen, and Martin Schoeberl. "Time-division Multiplexing vs Network Calculus: A Comparison". In: *23rd International Conference on Real Time and Networks Systems (RTNS'15)*. 2015, pp. 289–296 (cited on pages 91, 170).

[153] Claire Pagetti, David Saussie, Romain Gratia, Éric Noulard, and Pierre Siron. "The ROSACE case study: From Simulink specification to multi/many-core execution". In: *20th Real-Time and Embedded Technology and Applications Symposium (RTAS'14)*. 2014, pp. 309–318 (cited on pages 109, 112, 178).

[154] Mitra Nasri, Morteza Mohaqeqi, and Gerhard Fohler. "Quantifying the Effect of Period Ratios on Schedulability of Rate Monotonic". In: *24th International Conference on Real-Time Networks and Systems (RTNS'16)*. 2016, pp. 161–170 (cited on page 116).

[155] Morteza Mohaqeqi, Mitra Nasri, Yang Xu, Anton Cervin, and Karl-Erik Årzén. "On the Problem of Finding Optimal Harmonic Periods". In: *24th International Conference on Real-Time Networks and Systems (RTNS'16)*. 2016, pp. 171–180 (cited on page 116).

[156] Raul Gorcitz, Emilien Kofman, Thomas Carle, Dumitru Potop-Butucaru, and Robert De Simone. “On the Scalability of Constraint Solving for Static/Off-Line Real-Time Scheduling”. In: *13th International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS’15)*. 2015, pp. 108–123 (cited on page 124).

[157] Philippe Laborie and Jérôme Rogerie. “Reasoning with Conditional Time-intervals”. In: *21st International Florida Artificial Intelligence Research Society Conference (FLAIRS’08)*. 2008 (cited on pages 125, 126, 184).

[158] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. “Reasoning with Conditional Time-Intervals. Part II: An Algebraical Model for Resources”. In: *22nd International Florida Artificial Intelligence Research Society Conference (FLAIRS’09)*. 2009 (cited on pages 125, 126, 184).

[159] IBM ILOG. *CPLEX Optimization Studio*. http://www.ibm.com/software/integration/optimization/cplex-optimization-studio/. 2014 (cited on pages 125, 185).

[160] C. Lekkeikerker and J. Boland. “Representation of a finite graph by a set of intervals on the real line”. In: *Fundamenta Mathematicae* 51.1 (1962), pp. 45–64 (cited on page 129).

[161] Fanica Gavril. “Algorithms for Minimum Coloring, Maximum Clique, Minimum Covering by Cliques, and Maximum Independent Set of a Chordal Graph”. In: *SIAM Journal on Computing* 1.2 (1972), pp. 180–187 (cited on page 129).

[162] Quentin Perret, Pascal Maurère, Éric Noulard, Claire Pagetti, Pascal Sainrat, and Benoit Triquet. “Temporal isolation of hard real-time applications on many-core processors”. In: *22nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS’16)*. 2016 (cited on page 146).

[163] Quentin Perret, Pascal Maurère, Éric Noulard, Claire Pagetti, Pascal Sainrat, and Benoit Triquet. “Mapping hard real-time applications on many-core processors”. In: *24th International Conference on Real-Time Networks and Systems (RTNS’16)*. 2016 (cited on page 147).

[164] Joël Goossens. “Scheduling of Offset Free Systems”. In: *Real-Time Systems* 24.2 (2003), pp. 239–258 (cited on page 150).

[165] Jan Korst, Emile H. L. Aarts, Jan Karel Lenstra, and Jaap Wessels. “Periodic Multiprocessor Scheduling”. In: *International Conference on Parallel Architectures and Languages Europe (PARLE’91)*. 1991, pp. 166–178 (cited on page 150).

[166] Omar Kermia. “Ordonnancement temps réel multiprocesseur de tâches non-préemptives avec contraintes de précédence, de périodicité stricte et de latence”. Thèse de doctorat dirigée par Sorel, Yves Informatique Paris 11 2009. PhD thesis. 2009, 1 vol. (208 p.) (Cited on page 150).

[167] Ahmad Al Sheikh. “Resource allocation in hard real-time avionic systems. Scheduling and routing problems”. Theses. INSA de Toulouse, 2011 (cited on page 150).

[168] Jeffrey Dean and Sanjay Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *6th Symposium on Operating System Design and Implementation (OSDI’04)*. 2004 (cited on page 150).

[169] *FP7 Certification of Real-Time Applications Designed for Mixed-Criticality (CERTAINTY)*. http://www.certainty-project.eu/ (cited on page 163).

List of Figures

1.1 Comparison of federated and IMA architectures 22
1.2 Logarithmic comparison of transistor count, frequency, power, performance and number of cores in micro-processors between 1985 and 2010 (extracted from [8]) . 23

2.1 Example of application model 30
2.2 Architecture of the KALRAY MPPA®-256 30
2.3 Architecture of compute clusters in the KALRAY MPPA®-256 31
2.4 Hierarchical SRAM arbiter of KALRAY MPPA®-256 32
2.5 DDR-SDRAM arbiter of the KALRAY MPPA®-256 34
2.6 NoC switch of the KALRAY MPPA®-256. The local interface is not depicted for clarity. 36
2.7 Example of remote memory operations on a simplified model of the KALRAY MPPA®-256 37

3.1 Variability on the execution times of a program 44
3.2 WCET estimation procedure using static analysis techniques 45
3.3 Example of a 4x4 tiled many-core processor with a 2D-mesh Network on Chip. . . 48
3.4 Example of two packets of four flits crossing a wormhole-switched network 50
3.5 Architecture of a DRAM bank. 51
3.6 Simplified state machine of a DRAM bank access protocol. 51
3.7 DRAM commands under open-page row-buffer management policies for requests of example 7 54
3.8 Example of deadlock on a wormhole-switched NoC. 56
3.9 Example of two cumulative flows with graphical representation of the backlog and the delay 58
3.10 Block diagram of the PRET architecture (extracted from [76]). 61
3.11 Block diagram of the TCREST architecture (extracted from [84]). 62
3.12 Block diagram of the Merasa architecture (extracted from [87]). 62
3.13 Block diagram of a CompSOC instance (extracted from [72]). 63
3.14 Example of Deterministic Adaptative Scheduling with τ_1 and τ_2 non critical tasks and τ_3 and τ_4 critical tasks. 65
3.15 Architecture of MARTHY (extracted from [101]) 66
3.16 Example of execution with MARTHY with two tasks τ_1 and τ_2 running on two cores 66
3.17 Architecture of Memguard (extracted from [103]) 67
3.18 Example of schedule following PREM with two tasks τ_1 and τ_2 67
3.19 Example of distributed run-time WCET control with one non-critical task τ_1 and 2 critical tasks τ_{C2} and τ_{C3} 68

4.1 Example of a DAG task τ_1 with 10 sub-tasks . . . 72
4.2 Example of task modeled using Petri nets . . . 76
4.3 Example of schedule produced by HLFET on two cores. The sub-tasks of the critical path are colored in gray. . . . 77
4.4 Example of FTTS schedule on 2 cycles (extracted from [47]). . . . 79
4.5 Read, write and execute phases of Runnables (extracted from [146]). . . . 79
4.6 Example of scheduling table produced by LoPhT (extracted from [147]). . . . 80
4.7 Example of *pull* (steps ① and ②) and *push* (steps ③ and ④) operations on the Intel SCC [14] (extracted from [148]). . . . 81
4.8 Example of TTEthernet network with 5 end-systems and 2 switches (extracted from [150]). . . . 81

5.1 Example of application of the Rule 1 of the execution model on simplified compute cluster architecture with 2 partitions . . . 90
5.2 Example of application of the Rule 2 of the execution model on simplified NoC with 2 communications competing for a shared NoC resource ② → ⑤ . . . 91
5.3 Example of application of the Rule 4 of the execution model on simplified DDR-SDRAM arbiter with 3 masters P_A, P_B and P_C competing to access the memory . 92

6.1 Integration of partitions on the KALRAY MPPA®-256 work-flow . . . 96
6.2 Example of schedule computed during the *Validate* phase. . . . 98

7.1 Example of local SRAM partitioning using virtual memory protection . . . 104
7.2 Example of application of a NoC scheduling table . . . 105
7.3 Example of list of buffer queues associated to a PC . . . 106
7.4 Example of relative offsets for managing distributed memory . . . 107
7.5 Longitudinal flight controller of the ROSACE case study . . . 109
7.6 Schedule of ROSACE blocks in 5 threads . . . 110
7.7 Budget of the ROSACE's partition with 2 PNs and 3 PCs . . . 110
7.8 Example of application of ImgInv . . . 111
7.9 Measures of the execution times of ROSACE . . . 113
7.10 Configuration for the scenario 2 with two instances of ImgInv sharing NoC resources with ROSACE . . . 114
7.11 Configuration for the scenario 3 with one instance of ImgInv sharing a DDR-SDRAM bank with ROSACE . . . 115
7.12 Example of buffer queues for non-optimized memory mapping . . . 116
7.13 Example of buffer queues for optimized memory mapping . . . 117
7.14 Execution times of the DMA micro-code in function of the buffers queues. . . . 118
7.15 Measured WCET of the hypervisor versus the size of the DMA buffer queue . . . 119

8.1 Example of three conditional time-intervals i_1, i_2 and i_3 with only one present in the final solution . . . 125
8.2 Example of three intervals associated with pulse cumulative functions . . . 126
8.3 6 interval variables representing 1 sub-task τ_i^j with 2 sub-jobs that may be scheduled on 3 PNs . . . 128
8.4 Example of sub-job interval presence following the constraint on migration of sub-jobs between PNs . . . 129
8.5 Limitation of the amount of data sent during the activation of a PC using cumulative functions on data intervals . . . 131

8.6 Data interval enforcing the precedence between sub-jobs . . . 132
8.7 Comparison of computation times in function of the load increase with different budgets. The x axis represents the length of the application's hyperperiod (small value means high utilization) and the y axis represents the time used by the solver (timeout at 10,800 seconds) . . . 135
8.8 Total size of data per PC slot (1/2) . . . 137
8.9 Total size of data per PC slot (2/2) . . . 138
8.10 Number of data per PC slot (1/2) . . . 139
8.11 Number of data per PC slot (2/2) . . . 140

A.1 Exemple de modèle d'application . . . 158
A.2 Architecture du KALRAY MPPA®-256 . . . 160
A.3 Architecture d'un cluster de calcul du KALRAY MPPA®-256 . . . 161
A.4 Politique d'arbitrage vers la SRAM locale sur le KALRAY MPPA®-256 . . . 162
A.5 Exemple d'accès mémoires distants sur le KALRAY MPPA®-256 . . . 168
A.6 Exemple d'application de la règle 1 du modèle d'exécution sur une architecture simplifiée de cluster de calcul avec 2 partitions . . . 170
A.7 Exemple d'application de la règle 2 du modèle d'exécution avec 2 communications partageant un lien NoC ② → ⑤ . . . 171
A.8 Exemple d'application de la règle 4 du modèle d'exécution sur un arbitre DDR-SDRAM simplifié et 3 masters P_A, P_B et P_C en concurrence pour accéder à la mémoire . . . 171
A.9 Atelier d'intégration de partitions multiples sur le KALRAY MPPA®-256 . . . 173
A.10 Exemple d'application d'une table d'ordonnancement NoC . . . 177
A.11 Exemple de liste de liste de zones mémoires associées à un PC . . . 178
A.12 Contrôleur longitudinal de vol du cas d'étude ROSACE . . . 179
A.13 Budget de la partition de ROSACE avec 2 PNs et 3 PCs . . . 179
A.14 Exemple d'utilisation d'ImgInv . . . 180
A.15 Temps d'exécution des 11 blocs de ROSACE . . . 180
A.16 Configuration du deuxième scénario avec deux instances d'ImgInv partageant des ressources NoC avec ROSACE . . . 181
A.17 Configuration du scénario avec une instance d'ImgInv partageant un banc de DDR-SDRAM avec ROSACE . . . 181
A.18 Exemple de 3 variables d'intervalle optionnelles i_1, i_2 et i_3 où seulement une est présente . . . 185
A.19 Exemple d'utilisation des fonctions portes sur 3 intervalles i_1, i_2 et i_3 . . . 186
A.20 6 intervalles représentant 1 sous-tâche τ_i^j avec 2 sous-jobs ordonnançable sur 3 PNs 187
A.21 Limitation de la quantité de données qui peut être envoyée durant l'activation d'un PC en utilisant les fonctions cumulatives sur les variables d'intervalle des données . 189
A.22 Intervalle de donnée utilisé comme pivot pour mettre en œuvre une précédence entre deux sous-jobs placés sur 2 PNs distants. . . . 190
A.23 Comparaison des temps de calcul en fonction de l'augmentation de la charge applicative, des 9 budgets et des configurations des DMAs . . . 193

List of Tables

1.1 Description of the Design Assurance Levels from the ARP-4761 [5] 20

2.1 Example of sub-tasks parameters 29

3.1 Example of timing constraints extracted from the Micron 4GB MT41K512M8-125 DDR3 SDRAM module documentation [49]. 53
3.2 Example of DRAM commands issued under open-page and close-page row-buffer management policies 55

4.1 Example of adjusted timing parameters for DAG tasks using the algorithm described in [118] 75
4.2 Example of schedule produced by HLFET on two cores 78

5.1 Notations and values of the KALRAY MPPA®-256 hardware parameters 89

7.1 Example of scheduling table of PC_1^A and PC_1^B 105

8.1 Reminder on notations defined throughout the dissertation 122

A.1 Paramètres des sous-tâches 159
A.2 Exemple de table d'ordonnancement pour PC_1^A et PC_1^B 177