Liveness analysis techniques and run-time environment for memory management of dataflow applications

Benjamin Dauphin

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2021IPPAT004

Thèse de doctorat

TELECOM Paris

IP PARIS

Liveness Analysis Techniques and Run-Time Environment for Memory Management of Dataflow Applications

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626 Institut Polytechnique de Paris (ED IP Paris)
Spécialité de doctorat : Réseaux, Informations et Communications

Thèse présentée et soutenue à Sophia Antipolis, le 16 mars 2021, par

Benjamin Dauphin

Composition du Jury :

Laure Petrucci Professeur des Universités, Université Paris 13	Présidente
Robert De Simone Directeur de recherche, INRIA Sophia Antipolis	Rapporteur
Jean-François Nezan Professeur des Universités, INSA Rennes	Rapporteur
Samuel Thibault Maître de conférences, INRIA Bordeaux – Sud-Ouest	Examinateur
Stéphane Mancini Maître de conférences, Université Grenoble Alpes	Examinateur
Andrea Enrici Ingénieur de recherche, Nokia Bell Labs France	Examinateur
Renaud Pacalet Directeur d'étude, Télécom Paris	Directeur de thèse
Ludovic Apvrille Professeur, Télécom Paris	Co-directeur de thèse

Remerciments

Je tiens à remercier toutes les personnes qui ont rendu ce travail possible.

En premier lieu mon directeur de thèse Renaud Pacalet, mon co-directeur Ludovic Apvrille, ainsi qu'Andrea Enrici et Rabéa Ameur-Boulifa. Vos conseils, nos échanges, votre intérêt et votre suivi de ce travail ont façonné ce qu'il est devenu aujourd'hui. Merci pour votre accompagnement.

Je souhaite également remercier les membres du jury. Les relecteurs, Robert de Simone et Jean-François Nezan pour le temps passé à la relecture de ce manuscrit et nos échanges ayant permis son amélioration. De même, je remercie Laure Petrucci, Samuel Thibault et Stéphane Mancini pour l'intérêt porté à mon travail et pour son évaluation.

J'adresse également un remerciement à toute l'équipe du LabSoC, notamment à Minh, Matteo, Emna, Maysam, Letitia, Le Van, Tullio. J'ai vraiment apprécié le temps passé ensemble, nos conversations à la pause déjeuner, et tout ce qui fait la vie de ce laboratoire de recherche.

Enfin, je remercie ma famille pour leur soutien et leurs encouragements toutes ces années, et sans qui je n'aurais jamais pu aller aussi loin.

Résumé substantiel

Cette thèse a été effectuée à Télécom Paris et a été financé par Nokia Bell Labs France. Les Bell Labs sont connus pour leurs recherches dans les domaines de l'informatique et de la télécommunication.

Dans cette thèse sont étudiées différentes techniques visant à la gestion des interblocages et de la saturation des capacités mémoires dans les systèmes embarqués.

Ce travail trouve sa motivation dans la complexification de l'architecture des systèmes informatiques au cours des dernières décennies, notamment avec la généralisation des architectures hétérogènes et Non-Uniform Memory Access (NUMA). Cette évolution se constate dans tous types de systèmes informatiques, de l'embarqué sur Multi-Processor System on a Chip (MPSoC) aux systèmes distribués pour le calcul haute performance (High-Performance Computing). Nous nous intéressons en particulier au problème de la saturation des capacités mémoires dans les systèmes embarqués utilisés pour le traitement numérique du signal (Digital Signal Processing). Nos contributions peuvent toutefois être utilisées pour d'autres types d'applications et de plateformes. Cette thèse apporte trois contributions.

La première contribution de la thèse, présentée dans le chapitre 3, consiste en une technique de prévention des interblocages se basant sur l'étude des cliques dans un type de graphes, les Memory Exclusion Graphs (MEG). Ces graphes représentent les buffers alloués en mémoire et leur possibilité d'allocation simultanée. La recherche de cliques comportant un ensemble de buffer dont l'allocation entrainerait une saturation de ressources mémoires permet de détecter des situations d'interblocages et de les prévenir en ajoutant des contraintes de précédence entre différentes tâches de la charge de travail. Ces contraintes de précédences sont ajoutées entre le consommateur d'un buffer de la clique et le producteur d'un autre buffer de cette même clique. L'ajout de la contrainte de précédence entraine une mise à jour du MEG et la recherche d'une nouvelle clique amenant à la saturation mémoire. Ce processus est répété jusqu'à qu'aucune clique correspondant à une saturation mémoire soit présent dans le MEG, garantissant l'impossibilité de saturation. Dans un objectif de minimisation de l'impact qu'entraine l'ajout d'une contrainte de précédence sur la durée d'exécution de la charge de travail, les tâches visées par cette

contrainte sont sélectionnées en fonction de leur date d'exécution limite.

Dans le chapitre 4 nous présentons la deuxième contribution, qui est une autre approche visant à résoudre les interblocages causés par la mauvaise gestion des ressources mémoires. Il s'agit d'une optimisation de l'analyse de vivacité conventionnellement utilisée pour l'étude de la saturation mémoire, permettant d'analyser des systèmes plus complexes en un temps réduit. Nous définissons un type d'automate appelé automate de contrôle, dont l'ensemble des états est un sous-ensemble de celui de l'automate conventionnel qui représente toutes les possibilités d'ordonnancement. Nous démontrons que l'analyse de ce sous-ensemble d'états est suffisante pour garantir l'absence d'interblocages, sans perte de précision par rapport à l'analyse conventionnelle. Cette réduction dans le nombre d'états à analyser permet de déployer l'analyse sur des systèmes plus complexes que ceux pour lesquelles l'approche conventionnelle est réalisable en un temps donné.

Ces deux contributions sont évaluées en les comparant à un outil issu de l'état de l'art dans le chapitre 5.

Le chapitre 6 détaille notre troisième contribution, correspondant à une mise en œuvre pratique de la deuxième contribution par le développement d'une technique d'évitement des interblocages utilisant les résultats de l'analyse de vivacité optimisée. Cette technique d'évitement a été intégrée à un environnement d'exécution expérimental, et peut également être intégrée à d'autres environnements d'exécutions issus de l'état de l'art. Nous présentons en détail l'architecture de notre environnement d'exécution, conçu pour fortement paralléliser l'exécution des fonctions de contrôle.

Pour conclure le chapitre 7 récapitule les différentes contributions, leurs limites et leurs possibilités d'améliorations. Nous proposons également plusieurs pistes de travaux futurs sur la base des contributions de la thèse.

Contents

Contents 5

List of Figures 8

List of Tables 10

I Background 11

1 Introduction 13

1.1 Modern Computing Platforms 13
1.2 Deadlocks 17
1.3 Problem Statement 18
1.4 Thesis Contribution 20
1.5 Synopsis and Outline 21

2 Related Work 23

2.1 Introduction 23
2.2 Models of Computation 24
2.3 Formalization of Scheduling Problems 34
2.4 Liveness and Deadlocks 38
2.5 Run-Time Environments for Dataflow Applications 45
2.6 Conclusion 48

II Contributions **51**

3 Approximate Deadlock Prevention using Memory Exclusion Graphs **53**
3.1 Introduction . . . 53
3.2 Target Platform . . . 54
3.3 Applications and Workload . . . 55
3.4 Memory Shortage . . . 59
3.5 Conclusions and Limits . . . 68

4 Efficient Liveness Analysis **71**
4.1 Introduction . . . 71
4.2 System models and design assumptions . . . 72
4.3 Schedule Automaton . . . 75
4.4 Control Automaton . . . 81
4.5 Liveness Analysis of Automaton . . . 82
4.6 Evaluation of Schedule and Control Automaton Analysis . . . 85
4.7 Mathematical Formalization . . . 87
4.8 Conclusion . . . 96

5 Experimental Evaluation **97**
5.1 Introduction . . . 97
5.2 Implementation Details . . . 98
5.3 Experimental Setup . . . 99
5.4 Minimal Supported Memory . . . 100
5.5 Permissiveness . . . 101
5.6 Computational Time . . . 102
5.7 Discussion . . . 104

6 Run-Time Environment **107**
6.1 Introduction . . . 107
6.2 Functionalities of a Run-Time Environment . . . 108
6.3 Design Choices to Integrate the Deadlock Handling Policy . . . 110
6.4 Architecture of our Run-Time Environment . . . 112
6.5 Conclusion . . . 129

7 Conclusion **131**
7.1 Contributions of the Thesis 131
7.2 Limitations and Improvements 132
7.3 Future Work and Perspective 133

III Back matter **137**

Acronyms **139**

Bibliography **143**

List of Figures

1.1 5G decoding chain (receiver side) 14
1.2 Example memory architectures 16

2.1 Example of an application Synchronous Data Flow (SDF) graph and its corresponding Acyclic Homogeneous Synchronous Data Flow (AHSDF) graph. 25
2.2 Example of an application Parametrized & Interfaced Synchronous Data Flow (πSDF) graph. S is a configuration actor, B a hierarchical actor . . . 28
2.3 An example Petri net, before and after the firing of a transition. 29
2.4 Reachability graph of example Petri net from Figure 2.3. 30
2.5 Example Petri nets from different classes used to study sequential Resource Allocation System (RAS). 31

3.1 Example logical architecture of a target platform. 54
3.2 Example applications' dependency graphs. 56
3.3 Example workload composed of one iteration of the 4G application and two concurrent iterations of the 5G application 56
3.4 Two example application extensions to a common period. 57
3.5 Example AHSDF and its corresponding Memory Exclusion Graph (MEG). 61
3.6 Example applications, whose concurrent execution may cause a memory shortage 64
3.7 Updated applications, free of memory shortages 65

4.1 The steps of liveness analysis 73
4.2 Target architecture 74
4.3 Example AHSDF for a single-application, with mapping 75

4.4 Schedule automaton for example in Figure 4.3 76
4.5 Control automaton for Figure 4.3 81

5.1 Two example applications used in our experiments. Numbers on edges represent buffer size 99

6.1 Example application and mapping 112
6.2 Software architecture 113
6.3 Finite State Machine of job status. 118
6.4 Finite State Machine of buffer status. 120
6.5 Architecture of the server. Circled numbers correspond to the step in which the event is sent in the example run. 121

List of Tables

1.1 Occurrence of deadlocks in our RAS 19

2.1 Comparison of a selection of deadlock handling tools and methodologies 45
2.2 Run-time environments for task-based and dataflow-based applications . 48

4.1 Evaluation results for a test-bench of 3 workload types 87

5.1 Minimal supported memory size of the approximate approaches, in comparison to theoretical minimum of Contribution 2 101
5.2 Number of allowed schedules by approximate approaches, in comparison to the theoretical maximum of Contribution 2 102
5.3 Computational time of the five approaches: three heuristics from `memDAG`, and our implementation of the first and second contributions 103

Part I

Background

Chapter 1

Introduction

1.1 Modern Computing Platforms

Today's heterogeneous multi-processor platforms embed tens of processing units such as general purpose Central Processing Units (CPUs), Digital Signal Processors (DSPs), Graphics Processing Units (GPUs) and dedicated hardware accelerators (e.g., Field Programmable Gate Arrays (FPGAs)). These platforms are used for real-time dataflow applications with high processing-power requirements in multiple domains: from the Digital Signal Processing (DSP) in wireless communication to image or video analysis in autonomous driving. These applications are subject to very large memory bandwidth and short memory latency constraints that often guide designers to adopt platforms with a Non-Uniform Memory Access (NUMA) architecture in which the memory resource is distributed over multiple, size-limited, physical memories. On one hand, these NUMA architectures offer designers the capability to meet memory bandwidth and latency constraints. On the other hand, memories are limited in size and computing nodes can usually access limited subsets of physical memories. This complicates the search for valid execution orders (schedules) for real-time dataflow applications as it introduces new sources for deadlocks.

This thesis is part of a collaboration between Télécom Paris and Nokia Bell Labs France. As such, the work of this thesis is focused on embedded systems for Digital Signal Processing. This thesis contributions could ideally be placed in the context of O-RAN software [3].

1.1.1 Applications

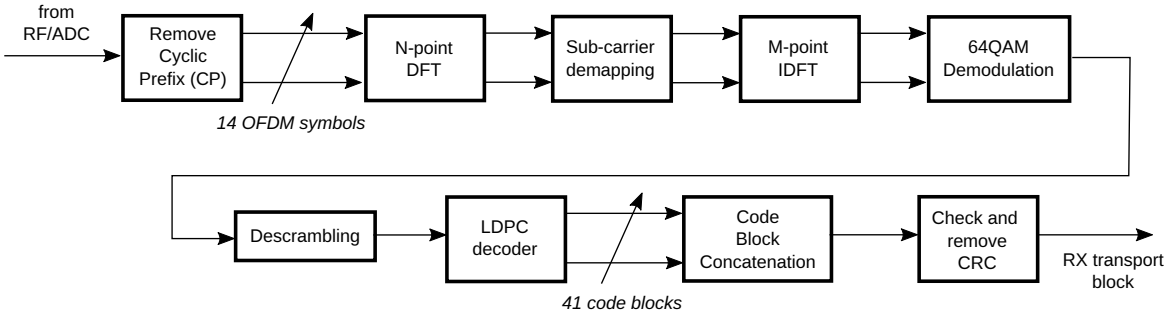


Figure 1.1: 5G decoding chain (receiver side)

Dataflow applications are composed of a set of tasks representing different computations. The tasks are represented as the vertexes of a dependency graphs, in which edges represent precedence constraints and communication between tasks. For example, Figure 1.1 displays a 5G decoding chain in which the `De-scrambling` task will be executed after the `64 QAM Demodulation` task, as the latter produces the input data used by the former.

Applications with multiple iterations can be analyzed as independent applications. It is also possible to analyze multiple iterations at the same time, which is interesting when some dependency constraints exist in (or can be added to) an application graph. For example if a task t in iteration $n + 1$ should not be finished before t in iteration n, analyzing multiple iterations simultaneously reduces the complexity in comparison to treating the two iterations as independent.

1.1.2 Evolution of the Architecture of Computing Systems

The steady increase in the complexity of applications and their performance requirement has led to significant evolution of the architecture of computing systems in the past decades. As they became more complex, modern computing systems also became harder to program.

From Homogeneity to Heterogeneity

The architecture of the first computing platforms, such as the ENIAC and the EDVAC had a single computational unit, allowing exclusively the execution of programs in a

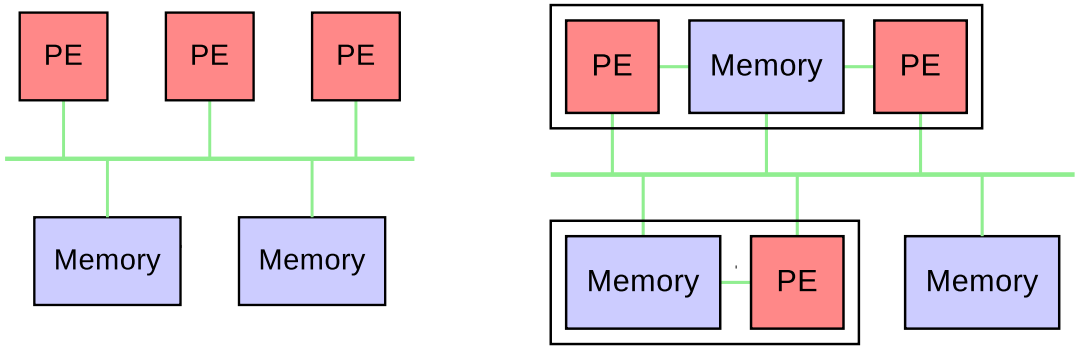
fully sequential manner. The use of multiple computing units on a single platform quickly became an obvious way to increase computing performance, by allowing multiple computations to occur in parallel. This led to the rise of homogeneous architectures, architecture in which all computing units (or Processing Elements (PEs)) have the same capabilities, i.e., any task can run on any PE. Homogeneous architecture can potentially have PEs with different computing speed, such as having one PE being twice as fast as another. But homogeneous architecture have their limits, which have been challenged with the ever increasing demand for computing power. For example, it is not possible to increase the speed of general purpose CPUs indefinitely, physical constraints such as energy consumption and heat dissipation prevent to increase the frequency beyond a threshold. As such, the designers of computing systems have resorted to use more and more heterogeneous architectures over the years. For example heterogeneous embedded systems, called Multi-Processor Systems on a Chip (MPSoCs), feature processors of various types on a same chip. PEs of an MPSoC offer different capabilities, usually featuring some general purpose CPUs, as well as specialized PE (GPUs, hardware accelerators, etc.). This means that not all tasks can run on all PEs, but also that when a task is able to run on different types of PEs, their performance can vary drastically and in a non-linear fashion. The Marvell OCTEON Fusion CNF95xx Family [2] is an example of heterogeneous platform designed for 4G and 5G base stations. For smartphones and tablets platform, we can cite the Samsung Exynos and Qualcomm Snapdragon ranges. The Xilinx Zynq UltraScale+ [72] family of System on a Chip (SoC) is an example heterogeneous platform designed for the domain of FPGA-based Digital Signal Processing.

Memory Architecture

The memory architecture of computing systems depends of the target platform, and can fall into different categories.

In a platform with Uniform Memory Access (UMA) each memory unit can be accessed by all PEs, and the access speed to units are identical for each PE. Figure 1.2a shows the memory system of a UMA platform with multiple memory units.

Figure 1.2b displays an example NUMA platform with multiple memory units. In NUMA architectures the access time between a PE and a memory unit depends



(a) Example Uniform Memory Access platform

(b) Example Non-Uniform Memory Access platform

Figure 1.2: Example memory architectures

on the considered PE/unit pair. Some PEs can even have their memory accesses restricted to only one or a subset of the memory units. A typical example of this is a hardware accelerator PE tightly coupled with its working memory and that cannot access any other memory in the system. Another actor, for instance a Direct Memory Access (DMA) engine, is then used to move data in and out of the PE's working memory.

1.1.3 Software Development for Modern MPSoCs

With the increasing complexity of the architecture of modern MPSoCs, their manufacturers started to provide development tools called Software Design Kits (SDKs). An SDK is a set of tools such as libraries, compilers, run-time environments, etc. aimed at easing the development for the platform. Example commercial SDKs includes those provided by Xilinx [71], Intel [5], or the Data Plane Development Kit (DPDK) [1] among many other. The quality of a platform's SDK is a crucial part of what will make the commercial success or failure of the platform, as it determines the ease of development onto the platform and therefore its adoption. Still, most SDKs for modern MPSoCs do not provide functionalities towards deadlock handling. As such, it is left to the software developers to ensure that the system stays in a safe state or can be recovered from an unsafe state. This lack of deadlock-handling tools in the official SDKs makes harder the task of developing for those platforms. One

of the aims of this thesis is to address this lack of deadlock handling strategy built into the SDKs, by providing design time and/or run-time tools providing deadlock handling to MPSoCs.

1.2 Deadlocks

A deadlock is a situation in which a set of processes are each waiting from another process of the set to take an action in order to be able to continue their execution, such that no process is able to continue. These processes can represent algorithmic computations, but also the communication of data inside a platform for example. Deadlocks occur when the different processes occupy resources (be it memory buffers, communication nodes, etc.) in such a way that no process can get access to all the resources it needs to complete its tasks.

Many techniques have been developed over the years to address the issue of deadlocks. These techniques are usually categorized in three main types.

First, *deadlock prevention* techniques, which aim at ensuring at design-time that deadlock will not occur once the system is running. These techniques are focused on the respect of fixed constraints which can be analyzed statically. There advantage lies in the absence of run-time mechanism. Their main drawback is their high static cost to perform the analysis, or the over-conservativeness leading to reduced run-time flexibility. Deadlock prevention is used in systems which have limited resources such as embedded systems, due to their low run-time overhead, and in safety-critical domains such as avionics, medical devices, etc. where the occurrence of deadlock should be avoided at any cost.

Second, *deadlock detection and recovery* techniques are focused on letting the system run freely, with some sort of monitoring to check periodically if a deadlock occurred. When the monitor detects a deadlock, the system is rolled back to a safe state, by restoring the system to a previous state, or by dropping some computation results (which will have to be re-executed) for example. An important advantage of deadlock detection and recovery is the flexibility given at run-time to perform operations. The recovery mechanism is only used once needed, thus not constraining the execution. Deadlock detection and recovery has some run-time overhead due to the monitoring it requires. Furthermore, the recovery mechanism can be expensive

in itself, or require to execute again some expensive computations whose results have been lost in the process. As such, deadlock detection and recovery is used in systems with sufficient resource to handle the run-time overhead, and for which the occurrence of deadlock is rare enough for the cost of recovery to be acceptable. This is the case for example in some High-Performance Computing (HPC) systems for big data.

Third, *deadlock avoidance* techniques, which guide the execution of the system to ensure at run-time that no decision leading to a deadlock is taken. As for deadlock prevention, and contrary to deadlock detection and recovery, their goal is make sure no deadlocks occur. The difference lies in the fact that deadlock avoidance uses run-time control of the system execution. These techniques are commonly more flexible than deadlock prevention, but do incur some run-time overhead. There are suitable in situations where deadlock detection and recovery is too expensive, by the relative frequency of deadlocks if the system is given full run-time flexibility, or downright impossible by a lack of resources available to perform the recovery.

These three main categories of strategies to address the issue of deadlocks, as well as a selection of existing methods for each category, will be presented in details in next chapter.

1.3 Problem Statement

The work of this thesis is focused on solving the issue of deadlock arising in computing systems and caused by memory shortages, by designing and implementing methods to tackle such deadlocks. This thesis does not address the issue of communication-induced deadlocks, nor aims at addressing deadlocks occurring in different kinds of systems such as distributed systems, High-Performance Computing (HPC), etc.

We target embedded systems such as MPSoCs, that we represent as a Resource Allocation System (RAS). In simple terms, a RAS aims at representing the use of a set of resources by processes, with each resource having a fixed capacity and each process requiring to be allocated a given amount of resources to be executed. RASs are presented in more details in next chapter. In the work of this thesis, we consider three kinds of resources: PEs, communication units, and memory units. Each mem-

ory port is considered as a separate communication resource, and memories are referred only in terms of their capacity (memory space). The capacity of PEs and communication units is 1, the capacity of memory units is their size.

Coffman et al. [19] determined four necessary and sufficient conditions for deadlocks to occur.

- Mutual exclusion: tasks require the exclusive use of a resource, every resource is either assigned to a single task or available.
- Hold-and-wait: a task is holding at least one resource while requesting additional resources.
- No preemption: a resource ownership cannot be suspended or canceled until the end of the task holding the resource.
- Circular wait: there is a set of tasks $\{t_1, t_2, \ldots, t_n\}$ such that t_1 is waiting for a resource held by t_2, t_2 is waiting a resource held by t_3, and so on until t_n waiting for a resource held by t_1.

A summary of the occurrence of these conditions, in our RAS, is presented in Table 1.1.

Condition	Processing	Communication	Memory (infinite)	Memory (limited)
Mutual exclusion	true	true	false	true
Hold-and-wait	false	false	true	true
No preemption	true	true	true	true
Circular wait	false	true	false	true
Deadlock risk	**no**	**no**	**no**	**yes**

Table 1.1: Occurrence of deadlocks in our RAS

In our RAS, mutual exclusion does not occur in the case of (purely theoretical) infinite memory because new buffers can always be allocated in new memory space. The hold-and-wait situation can arise in memory units as some input buffers can be already allocated while others are yet to be. We supposed the absence of preemption for tasks on processing nodes, a very frequent situation in high-end data

processing systems with stringent time constraints (deadlines). There is also no preemption on memory as we decided that allocated buffers can only be freed once their consuming tasks have been executed (there is no mass storage in which to unload excess buffers). There cannot be circular waits for processing nodes as each task is executed on a single PE, and a task requests its processing node only once all the other resources it needs (e.g., in/out buffers) have already been allocated.

Based on the characteristics of our resource allocation problem and the hypothesis above, Table 1.1 allows us to conclude that deadlocks can only arise because of the shared use of memory units of limited capacity. In other words, memory shortages are the source of deadlocks in our target systems.

1.4 Thesis Contribution

The objective of this thesis is to study memory induced deadlocks for dataflow applications running on embedded systems, and to develop and implement methods to avoid running into deadlocks at run-time. The contribution of this thesis is threefold:

1. An approximate deadlock prevention method for systems running applications with a common period (or which can easily be adapted to a common period). This method is based on the analysis of a graph called the Memory Exclusion Graph (MEG) representing the buffers allocated into a memory for the execution of tasks and the exchange of data from producing tasks to consuming tasks. This graph also tells us which buffers could possibly be allocated simultaneously, giving valuable information about the potential of memory shortages. Albeit of limited practical use, this method gave valuable insight to better understand the applications, platforms, and conditions leading to deadlocks.

2. A liveness analysis suitable for systems regardless of the timing constraints of applications, and its derived exact deadlock avoidance mechanism. This analysis and subsequent deadlock avoidance mechanism have been developed thanks to the insight given by the first contribution. They are also not held by the main drawbacks that limited the practical usability of the first approach.

3. The development of a control mechanism to prevent deadlock at run-time using the aforementioned exact deadlock avoidance mechanism. This control mech-

anism has been deployed in a prototype run-time environment which manages the execution of dataflow applications.

It should be noted that although developed in the context of embedded systems for Digital Signal Processing, the contributions could be applied for other types of applications (such as video processing or machine learning) and of target platforms having issues of memory (or other similar resource) shortage.

1.5 Synopsis and Outline

Next chapter (Chapter 2) presents the state of the art in scheduling and deadlock prevention techniques. The following chapters present the contribution. In Chapter 3 the approximate deadlock prevention method is described. Chapter 4 presents the precise liveness analysis. Then Chapter 6 presents how the results from the precise liveness analysis can be used to prevent deadlock at run-time. In Chapter 5 experimental evaluations are conducted to compare the performance of the two deadlock handling mechanisms developed in this thesis with a state-of-the-art deadlock prevention tool. Finally Chapter 7 discusses the contributions and the future work.

Chapter 2

Related Work

2.1 Introduction

In this thesis we look at the deployment of dataflow applications in embedded systems. This ranges from the modeling of the applications when designing the system, to their scheduling and execution onto the target platform, done by a run-time environment. When running such applications it is important to make sure the scheduling is safe, i.e., no memory overflows or shortages, deadlocks, etc. can occur. This raises the need to use liveness analysis and/or deadlock handling mechanism to ensure the safety, which can be critical for embedded systems used for avionics or medical applications, for example.

The goal of this thesis is to develop tools and algorithms that enable the safe deployment of dataflow applications onto heterogeneous and/or NUMA platforms. The remainder of this chapter is structured as follows. Section 2.2 presents different Models of Computation used to represent dataflow applications. Then Section 2.3 presents different ways to model scheduling problems and compute their schedules. Section 2.4 presents multiple strategies that have been developed to analyze the liveness of a system, and addresses the issue of deadlocks that arise when running concurrent computations. Section 2.5 presents various existing run-time environment managing the execution of dataflow applications. Finally, Section 2.6 concludes this chapter.

2.2 Models of Computation

Over the years the complexity of dataflow applications has risen dramatically. To compensate for the continuous increase in the required computing power, platforms have become more parallel. This is shown by the advent of large-scale distributed systems, or of modern Multi-Processor Systems on a Chip (MPSoCs) having an increasing number of cores and hardware accelerators. This increase in the potential for parallelism and the complexity of platforms led to the development of higher level —more abstract—languages and Models of Computation (MoCs) for concurrent programming, to ease the otherwise complex programming. A Model of Computation (MoC) describes the set of rules that governs the execution of programs, by specifying the semantics of its component and how they may interact. The first dataflow MoC was presented by Dennis in 1974 [20]. More modern examples of such high-level MoCs include Petri nets, and dataflow oriented MoCs such as the PRUNE MoC or the widely used SDF and its derivatives. Programming languages (such as C++, Java, etc.) differ from MoCs and are used to implement MoCs. A single programming language can be used to implement different MoCs, for example one could use Java to describe a Turing machine or an execution environment for Petri nets.

2.2.1 Synchronous Data Flow

The Synchronous Data Flow (SDF) [48] is a MoC developed to describe signal processing applications, in order to expose their potential for concurrency, thus enabling an efficient use of parallel computing capabilities. SDF has fixed production and consumption rates when firing actors, and is highly analyzable statically. Formally, in an SDF graph $G = (\mathcal{A}, \mathcal{E})$, the set of nodes $\mathcal{A}$ (called actors) represents tasks interconnected by a set of edges $\mathcal{E}$ that are First In, First Out (FIFO) buffers. In the SDF MoC, an actor starts execution (firing) when its incoming FIFOs contain enough tokens, it cannot be preempted and produces tokens onto its outgoing FIFOs. The number of tokens consumed/produced by each firing is a fixed scalar that is annotated to the graph's edges. As actors have no state in SDF graphs, if enough tokens are available, an actor can start several executions in parallel. For this reason, SDF graphs naturally express the parallelism of signal-processing applications and can be statically analyzed for several types of optimizations (e.g., memory allocation,

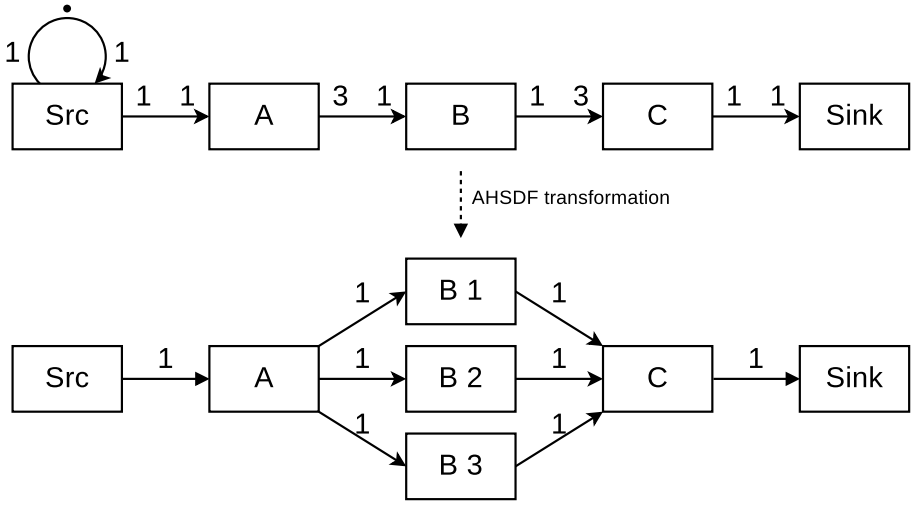


Figure 2.1: Example of an application SDF graph and its corresponding AHSDF graph.

scheduling). An example SDF graph can be seen at the top of Figure 2.1. In this graph, task Src reads and writes to a first FIFO, the self-loop, and writes to a second FIFO. The self-loop allows for at most one concurrent execution of the Src task, since it contains initially one token. Tokens that are present in the initial state of an SDF graph are called delay tokens. The second FIFO is the input of task A. After one execution of Src, the self-loop has again one token, and the FIFO between Src and A also has one token. An execution of task A will consume one token from the Src to A FIFO, and output three tokens to the A to B FIFO (i.e., the edge from A to B).

A subclass of SDF: Acyclic Homogeneous SDF

SDF graphs can have two properties that are useful for their analysis, namely acyclicity and homogeneity. An SDF graph is called *acyclic* if it contains no cycle, i.e. there are no paths in the graph with the same starting and ending nodes. Acyclic graphs are commonly used to isolate a single iteration of the represented application. *Homogeneity* refers to SDF graphs in which the production and consumption rates are equal for each FIFO. Homogeneous SDF graphs fully expose the data parallelism and memory allocation options.

Any SDF graph can be transformed into an Acyclic Homogeneous Synchronous Data Flow (AHSDF), as described in [64]. In a AHSDF graph $H = (\mathcal{T}, \mathcal{B})$, tasks in $\mathcal{T}$ are associated to identical production and consumption rates on FIFO buffers in $\mathcal{B}$. To do so, multiple instances of a same task are created when the production and consumption rates do not match in the original graph. Considering the original SDF graph $G = (\mathcal{A}, \mathcal{E})$. For any FIFO $f \in \mathcal{E}$, with p_f the production rate on f, c_f the consumption rate on f, and $\mathrm{lcm}(a, b)$ being the Least Common Multiple (LCM) of a and b. If p_f and c_f are not equal, $\mathrm{lcm}(p_f, c_f)/p_f$ instances of the producer are created and, conversely, $\mathrm{lcm}(p_f, c_f)/c_f$ instances of the consumer. For example, with a FIFO annotated with $p_f = 200, c_f = 300$ in the original SDF graph, there will be $\mathrm{lcm}(200, 300)/200 = 3$ instances of the producer and $\mathrm{lcm}(200, 300)/300 = 2$ of the consumer in the AHSDF graph. The graph can be made acyclic by removing all FIFOs with delay tokens. The result of this transformation on the example SDF graph can be seen at bottom of Figure 2.1. This transformation is used to expose data parallelism and memory allocation options (Homogeneous SDF), as well as to isolate a single iteration of the algorithm captured by the original SDF graph (Acyclic Homogeneous Synchronous Data Flow).

2.2.2 Cycle-Static Data Flow

The Cycle-Static Data Flow (CSDF) MoC is an extension of SDF first introduced by Bilsen et al. [12]. CSDF graphs are identical to SDF graphs in all aspects but the production/consumption rates on FIFOs. Indeed, in a CSDF graph the constant rate can be replaced by a finite sequence of values. For example, if a FIFO has a production sequence of $(2, 1)$, it means that the producing actor will add 2 tokens on its first firing, and 1 token on its second firing. Once the sequence end has been reached the sequence is repeated, so the third (and every odd-numbered) firing will again produce 2 tokens.

It has been demonstrated that CSDF is *not* more expressive than SDF, since any CSDF graph can be transformed into an equivalent SDF graph [60]. The main advantage of CSDF over SDF is that it allows to express complex applications with a smaller graph, and to statically analyze them as is the case with SDF. The counterpart is the higher computational cost of those analyzes.

2.2.3 PRUNE MoC

The PRUNE MoC has been designed to describe the behavior of high-performance signal processing applications. In PRUNE applications are represented by a graph with nodes representing actors (computations), and edges FIFOs allowing the communication of input/output and control data. There exists three types of actors, namely static, dynamic, and configuration actors. There also exists three types of FIFOs, static FIFOs for which the production/consumption rate is constant, dynamic FIFOs that allow this rate to change, and control FIFOs, that manage the rate change of dynamic FIFOs Static actors are similar to SDF actors and always produce and consume the same amount of tokens on their input/output static FIFOs. Configuration actors have one or multiple output control ports connected to the input control port of dynamic actors. Dynamic actors have exactly one control port (linked to a control FIFO in input), are connected to at least one dynamic FIFO, and can be connected to static FIFOs as well. At each firing, they dynamically adjust the consumption or production rate of *all* their dynamic FIFOs, based on the value of the token in their input control FIFO. Similarly to homogeneous SDF graph, FIFOs have a symmetric rate, i.e., the production and consumption rates are equal. But the PRUNE MoC is more expressive than homogeneous SDF graphs, since it allows the dynamic reconfiguration of the FIFO rates. Both ends of a dynamic FIFO should be controlled by the same configuration actor, to ensure this symmetric rate. Since the stated goal of the PRUNE MoC is to allow the static decision on the liveness and memory-boundedness of the modeled applications, a set of design rules must be respected to ensure that these properties are decidable in a finite time, by constraining the control of dynamic actors.

Albeit more expressive than SDF, the contributions of this thesis are not using the PRUNE MoC, as the added expressiveness entails an higher complexity to design applications and analyze them.

2.2.4 πSDF

πSDF is an extension of SDF that allows hierarchical actors and dynamic reconfiguration [23]. A hierarchical actor is an actor whose behavior is itself expressed using a MoC, here via a πSDF graph. πSDF hierarchical actors have been designed to

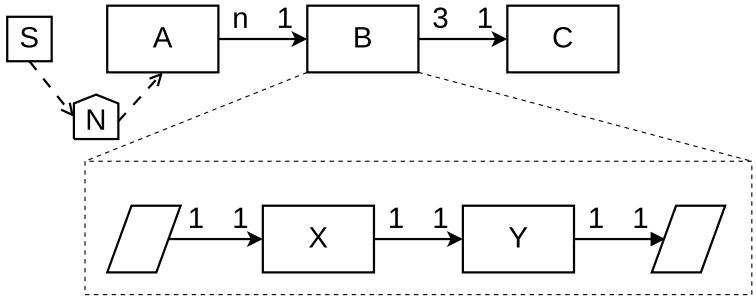
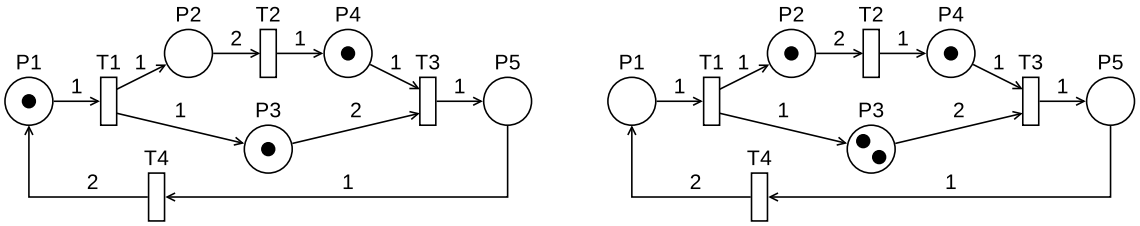


Figure 2.2: Example of an application πSDF graph. S is a configuration actor, B a hierarchical actor

ensure that their internal behavior does not influence the analyzability at the higher level. This is done using interfaces, which behave as circular buffers when more tokens are consumed than produced, and discard excessive tokens have been produced than are consumed. This interfaces behavior has been first defined in the Interface Based-Synchronous Data Flow MoC [62]. This behavior means that in Figure 2.2, the external behavior of actor B is not dependent of the internal behavior of actors X and Y, since they are interfaced (interfaces are represented as parallelograms in Figure 2.2). Here the interface connecting the lower level actor Y to the higher level B will act a circular buffer, since 1 token is produced when 3 are consumed. Hierarchical actors are convenient when designing applications, as they allow to compose and reuse application graphs. Similarly to PRUNE, πSDF has configuration actors that manage the dynamic reconfiguration. The configuration actors are fired once before every iteration of the full πSDF graph, and set parameters which will affect the production/consumption rates of some actors. For example in Figure 2.2, S is a configuration actor that set parameter N, which in turn determines the production rate of actor A.

Similarly to the PRUNE MoC, πSDF extends on SDF to allow dynamic reconfigurations. Again, the added expressiveness is not useful to express the behavior of our target applications.



(a) Example Petri net, with initial marking $M_0 = \{1, 0, 1, 1, 0\}$.

(b) Example Petri net from Figure 2.3a after the firing of $T1$.

Figure 2.3: An example Petri net, before and after the firing of a transition.

2.2.5 Petri Net

A Petri net is graph composed of two types of vertexes, called places and transitions, and directed arcs. A directed arc must connect a place to a transition or a transition to a place, but cannot connect two nodes of the same type. Figure 2.3a shows an example Petri net, in which circles $P1$ to $P5$ are the places, bars $T1$ to $T4$ the transitions, and arrows are the directed arcs. More formally, a Petri net is a directed bipartite graph $N = (P, T, F, M_0)$, in which the set of nodes $P \cup T$ represents places and transitions, with $P \cap T = \emptyset$. Transitions T represent events that can occur, and places P represent conditions such as resource usage. The directed arcs F describe the preconditions and postconditions for each transition, i.e., which conditions must be satisfied before and after the transition for it to be enabled. Places are annotated with a positive integer representing the number of tokens present, which is called the *marking* of the Petri net. M_0 corresponds to the initial marking of the Petri net's places, i.e. the initial state of the Petri net. In Figure 2.3a places $P1$, $P3$, and $P4$ have initially 1 token each. With this initial marking, transition $T1$ is the only enabled. After firing $T1$ we get the marking in Figure 2.3b.

The reachability graph of the example given in Figure 2.3 can be seen in Figure 2.4. Each vertex of the reachability graph corresponds to a possible state of the Petri net and is annotated with the corresponding marking. For example, in Figure 2.4 the uppermost state is annotated with $1, 0, 1, 1, 0$, meaning that places $P1$, $P3$, and $P4$ each have one token while places $P2$ and $P5$ are empty, which corresponds to the marking displayed in Figure 2.3a. The marking of the Petri net in Figure 2.3b corresponds to the leftmost state $(0, 1, 2, 1, 0)$ in the reachability graph

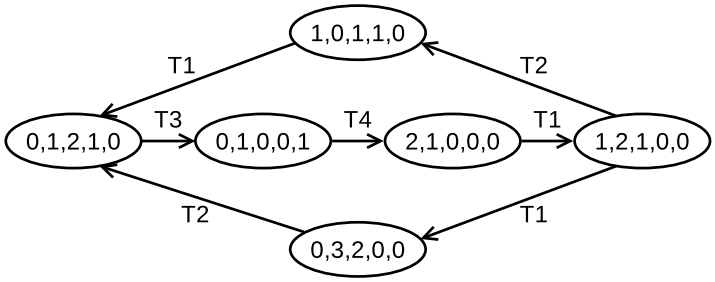


Figure 2.4: Reachability graph of example Petri net from Figure 2.3.

displayed in Figure 2.4. Each transition that is enabled for a given marking in the Petri net is represented by an edge coming from the state representing that marking in the reachability graph. For example, since $T1$ is the only transition enabled with the marking of Figure 2.3a, state $0, 1, 1, 0, 1$ in Figure 2.4 has only one outgoing edge, which corresponds to the firing of $T1$.

Petri nets are commonly used because of their expressiveness and analyzability, making them suitable to model concurrent systems and check if they respect some properties guaranteeing a safe execution, such as liveness and boundedness.

Pure nets — A Petri net is said to be pure if it has no self-loop, i.e., there exists no place that is simultaneously an input and an output of a transition. Pure Petri nets are easier to analyze as they can be represented with an incidence matrix. A Petri net that is not pure cannot be represented by a single matrix. Instead two matrices are necessary, one to represent inbound arcs (going from a transition to a place), and another to represent outbound arcs (going from a place to a transition). The Petri net displayed in Figure 2.3a is pure.

Boundedness — A place in a Petri net is k-bounded if in all states reachable from the initial marking it never has more than k tokens. From the reachability graph in Figure 2.4, we can see that in the example Petri net $P1$ is 3-bounded, $P2$ is 2-bounded, and the remaining places are 1-bounded. As such the example Petri net given in Figure 2.3 is 3-bounded.

Liveness — Four levels of liveness have been defined for the transitions of a Petri net, each level being more restrictive than the last. A transition has liveness level:

- L_1-live: if it can fire in at least one possible firing sequence;
- L_2-live: if $\forall k \in \mathbb{N}$ it can fire k times in at least one possible firing sequence. The

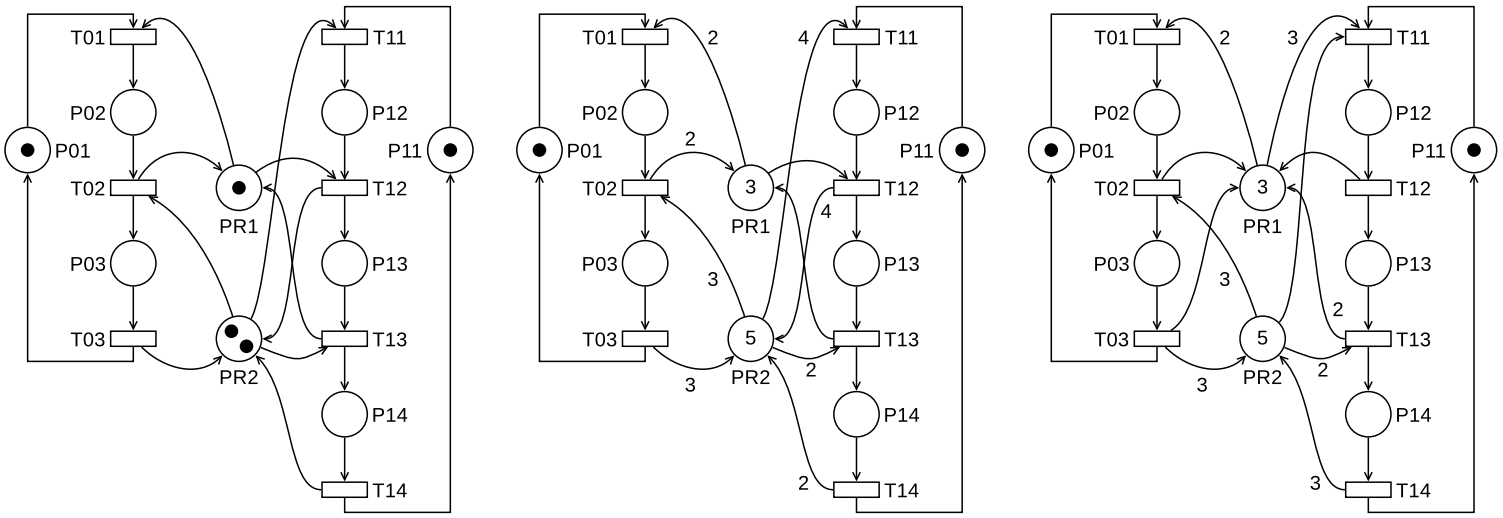


Figure 2.5: Example Petri nets from different classes used to study sequential RAS.

specific sequence can differ for each value of k;

- L_3-live: if it can fire infinitely in at least one possible firing sequence, i.e., there is at least one infinite firing sequence such that the transition can always be fired an infinite number of times;
- L_4-live: if it can fire in every possible state, it is always enabled.

A Petri net is called L_k-live if all its transitions are at least L_k-live. From the reachability graph in Figure 2.4, we can see that all transitions are L_3-live by making the infinite repetition of the sequence $T1, T3, T4, T1, T2$. As such, the example Petri net displayed in Figure 2.3 is itself L_3-live.

Petri nets are used as modeling tool for many systems such as to analyze business processes [10], parallel processing in computing systems, or the behavior of flexible manufacturing systems. Specialized, constrained, types of Petri nets have been developed to represent and analyze specific problems. Each type of constrained Petri net is called a Petri net class.

Classes of Petri Nets

Over the years many classes of Petri nets have been developed. Each class defines a set of constraints over the places and transitions of a Petri net. These constraints

are used to guarantee that some mathematical properties always hold, which make analyzing Petri nets more efficient. A thorough survey of Petri net classes has been made by Liu and Barkaoui [53].

This thesis aims at addressing the issue of deadlocks for some sort of sequential RAS, i.e., a RAS in which resources are reusable. Figure 2.5 shows three classes of Petri nets based on the same principles: there is a set of resources represented as places, and a set of processes represented as a circuit made of places and transitions. Processes can be used to describe the execution of periodic dataflow applications, or the behavior of a flexible manufacturing system for example. For all the Petri nets displayed in Figure 2.5, places $PR1$ and $PR2$ are the resources. The loop made of $P01, T01, P02, T02, P03, T03$ represent a first process, and the loop made of $P11, T11, P12, T12, P13, T13, P14, T14$ represent a second process. The difference between the different classes lies in the edges they allow between the processes and the resource, i.e. in what resource usage they are able to represent.

Lets first present the Generalized Systems of Simple Sequential Processes with Resources (GS^{3}PR) class [52], displayed in Figure 2.5b). A GS^{3}PR Petri net is a pure net made of processes and resources such that each transition have at most one inbound edge connected to a resource, representing the allocation of some of the resource capacity to the process. Conversely, each transition can have at most one outbound edge connected to a resource, representing the deallocation of the resource capacity. Furthermore in a GS^{3}PR, if a process place has an incoming transition allocating some resource, then any outgoing transition must free the same quantity of the same resource. This can be seen in Figure 2.5b) with the two arcs $PR1 \rightarrow T01$ and $T02 \rightarrow PR1$ both having a weight of 2. In other words a process cannot hold onto multiple resources simultaneously, and must free any allocated resource at its next execution step.

Systems of Simple Sequential Processes with Resources (S^{3}PR) [28] (Figure 2.5a) is identical to GS^{3}PR, excepted that there are no weight on edges. As such the S^{3}PR class is said to be *ordinary*, and is more constrained but easier to analyze than GS^{3}PR.

In Figure 2.5c is displayed a net of the S^{4}PR (S^{4}PR) class [68]. S^{4}PR is more permissive than GS^{3}PR, as it lifts the constrains on resource usage. This allows to represent process which can hold onto resources during their execution. In fact,

contrary to S^3PR and GS^3PR, it is possible to express the behavior of a set of AHSDF graphs using an S^4PR net.

There exists a variety of other Petri net classes for sequential RAS beyond those presented here. For example the classes S^*PR (S^*PR) [29] or Processes Competing for Conservative Resources (PC^2R) [54] are more permissive than S^4PR. Since S^4PR is permissive enough to express workloads of applications modeled as AHSDF graphs, we elected not to present them here.

A common approach to enforce liveness in Petri nets is the use of pure net supervisors. Pure net supervisors are sets of places, transitions, and arcs added to a Petri net that is not live in order to produce a new net that is live and pure. Zhong et al. [75] have pointed out that there does not always exist a maximally permissive liveness-enforcing pure net supervisor for a Petri net. This displays the importance of researching non pure net solutions to enforce liveness in a Petri net.

2.2.6 Discussion

The high-level representation of dataflow MoCs enables application and system developers to describe parallelism in streaming applications, such as media processing, in a simple and intuitive way. The different varieties of MoCs existing in the literature each have their own characteristics. These include the execution semantics of the MoC, its expressiveness, analyzability, and restrictions, making them suitable for different kinds of applications and platforms.

In the work of this thesis, SDF is used as the MoC to model the applications to be executed on the target platform. It has been chosen for its high static analyzability associated with a suitable expressiveness for targeted applications, making it possible to minimize the amount of dynamic analysis required, and thus being suitable for embedded systems, where computing resources are limited and best used to compute the actual applications. MoCs that support the reconfiguration of applications, such as πSDF and the PRUNE MoC, have not been considered as the added expressiveness is not necessary to express our target applications. Millo et al. [57] present an interesting work, using the MARTE/CCSL formalism to express the allocation and real-time constraints falling upon SDF applications deployed on embedded systems. The idea of using different formalisms in conjunction to represent applications and

their constraints at different levels of abstraction has also been studied by Arras et al. [7].

2.3 Formalization of Scheduling Problems

Scheduling is a common problem across many sectors. It refers to the need to allocate a limited amount of resources (e.g., materials and machines in a manufacturing plant, PEs in a computing system) in an efficient manner in order to complete a target work while optimizing some objectives (e.g., time, cost, energy consumption).

2.3.1 Types of Scheduling Problems

To describe different kinds of scheduling problems, ranging from construction work, management of a manufacturing plant, to scheduling of applications and tasks in computing systems, a variety of scheduling problems and formalisms have been described. We will present hereafter two of the most common and significant scheduling problems addressed in the literature.

Resource Constrained Project Scheduling Problem

A common problem definition found in the literature is the Resource Constrained Project Scheduling Problem (RCPSP). Problems are defined by a set of activities (or tasks) which have various resource requirements and can be dependent on other activities completion. This problem description is aimed at logistical projects (such as transporting good or infrastructure building), but share common aspects with the scheduling of dataflow applications. Both share a common goal, which is the optimization of the execution of activities or tasks that have specified resource usage. There are different characteristics defining an RCPSP.

Resources — The resources studied in RCPSPs can be of different kind. Renewable resources are resources available in a maximal quantity at any time in the project, such as vehicles in a transportation problem. This is similar to memory resources in our case. Non-renewable resources are available in a limited quantity over the full course of the project: once consumed they are destroyed and cannot be reused at a later point in the project. Doubly constrained resources are non-

renewable resources that also have a maximal simultaneous use like renewable resources.

Objective function — RCPSPs can have different objective functions such as minimizing the cost of the project, its duration, etc.

Activities — Activities can have different characteristics depending on the specific problem. They can be preemptible or non-preemptible (such as in our work). They can by limited to a single mode, or have multiple modes with different duration, resource needs, etc. This can be similar to the execution of tasks in a heterogeneous system, where a task could have different execution time or memory usage depending on its execution node.

Scheduling certainty — Many papers studying RCPSPs assume that creating a basic scheduling table in advance for the whole project is feasible. In practice, variations in the time taken to complete activities, availability of resources, or other external factors can impact the schedule when the project is ongoing. Different kinds of scheduling have been proposed to address the uncertainty that can occur:

- *Reactive scheduling*. Each time an unexpected event occurs, a new schedule is created for the remaining part of the project.
- *Stochastic scheduling*. The duration of activities is assumed to follow some probabilistic rule, known in advance.
- *Fuzzy scheduling* are used when it is not possible to evaluate beforehand the probability distributions of the project's perturbations.
- *Robust (or proactive) scheduling*. Its goal is to create a schedule that is the most robust to events, i.e. to minimize the effect random events can have on the schedule's objectives.
- *Sensitivity analysis* aims at studying how variations in an RCPSP parameters impact, or not, the optimal solution and its cost.

An overview of many RCPSPs has been done by Habibi et al. [34]. In our case, reactive scheduling is not suitable as the cost of computing a complete schedule is high and events occur frequently. More generally, all these approaches aim to address uncertainty when trying to compute a full schedule that is either easily changed

when an unexpected event occurs, or replaceable. For our problem, computing a full schedule is not suitable since events such as an application start, end, or reconfiguration can occur frequently and significantly change the scope of the schedule.

Resource Allocation Systems

Another formalization commonly used is the Resource Allocation System (RAS). Generally speaking, a RAS is formalized by a set R of m different *resource types*: $R = \{R_1, \ldots, R_m\}$, and a set J composed of n *process types* $J = \{J_1, \ldots, J_n\}$. Every resource is characterized by its capacity C_i, a finite positive integer. A process type J_j is characterized by a sequence of request vectors of dimension m: $\langle J_{jk}, k = 1, \ldots, l(j)\rangle$ with $l(j)$ the number of steps for process type J_j, and components $J_{jk}[i], i = 1 \ldots, m$ indicating the amount of units of resource R_i required for the completion of step J_{jk}.

Multiple classes of RASs exist, each having different constraints on the valid resource requests. The most significant classes are the following ones, from the most restrictive to the most permissive:

- **Single-unit RAS:** Request vectors can have a single non-zero component of value one;
- **Single-type RAS:** Request vectors can have a single non-zero component that is a positive integer lesser or equal to the corresponding resource capacity;
- **Conjunctive RAS:** All components of request vectors can take any integer value from 0 to the corresponding resource capacity;
- **Disjunctive/Conjunctive RAS:** For every stage, the vector of dimension m is replaced by a set of vectors such that the step can be completed if any request in that set can be satisfied. Each of those vectors follows the constraints of the request vectors in conjunctive RAS.

The problem of scheduling applications on a multi-core platform can be represented by a conjunctive RAS, or by a disjunctive/conjunctive RAS if there exists multiple implementations of a task having different memory requirements. An example of the latter would be a heterogeneous system in which a task can be executed by

a generic CPU or by a hardware accelerator, and these alternative implementations have different memory footprints.

2.3.2 Computation of Schedules for Multicore Systems

The scheduling of applications running on multi-core platforms is commonly divided in four successive steps, in order:

1. **Extraction:** The first step when scheduling for a multi-core system is to extract the potential parallelism contained within the set of applications to run. This parallelism can come from tasks having no dependency relation, but also from having multiple applications—or iterations of a same application— running concurrently. The extraction step for SDF graphs can be done via its transformation into an AHSDF graph (cf. Subsection 2.2.1).

2. **Mapping:** The mapping corresponds to assignment of tasks to PEs, and of buffers to memories. Task mapping is constrained by the computing capabilities of each PE in heterogeneous architectures, and is commonly done using heuristics with objectives such as minimizing the latency, following real-time constraints [16], maximizing the energy efficiency [41, 46], etc. Buffer mapping in NUMA architectures is usually done to ensure the locality of data, i.e., assigning a buffer to a memory that can be quickly accessed from the PEs on which the tasks using that buffer have been mapped, such as in the work of Drebes et al. [26] for example.

3. **Ordering:** Once tasks have been mapped onto PEs an execution order is computing for each PE. The execution order must respect the dependencies between task, and can also be constrained via deadlines on tasks. It is common to compute the ordering simultaneously with the mapping, since the efficiency of a given mapping is affected by the task ordering.

4. **Timing:** Finally, timing corresponds to the assignment of a specific starting time for the execution of tasks. It depends on various factors such as data availability and system load. Except for the most critical systems, with hard deadlines that must be met in any circumstances, timing is usually done online by the run-time manager or the operating system.

Depending on whether these steps are done statically or online, four main scheduling strategies for multi-core systems can be described [47]:

- **Fully static:** All steps are computed offline, at design or compile time, leading to a single schedule or a handful of schedules out of which one is selected at run time depending on the configuration. It has the advantage of minimizing the run time overhead, but at the cost of a reduced flexibility and a potentially high computational complexity.

- **Self timed:** Only the timing step is computed dynamically. This gives more run time flexibility while keeping the overhead low. This approach is useful for systems where the execution time of tasks is variable.

- **Static assignment:** The extraction and mapping steps are done statically, while the ordering and timing are computed at run time depending on the dynamic conditions in the system.

- **Fully dynamic:** All steps are computed dynamically. This gives the highest flexibility to the system, but also the highest overhead at run-time, making it suitable only for systems where this overhead can be tolerated (typically with not strictly limited computing resources). An example fully dynamic scheduling for homogeneous multi-core systems can be seen in [11].

2.4 Liveness and Deadlocks

2.4.1 Liveness Analysis

A liveness analysis goal is to analyze a system to determine in which situations (or system states) does a given property hold (the system is *live*) or not. The analyzed property can represent for example resource usage (the system is live when there are no resource overflow), or are all buffer reads performed after a corresponding buffer write (the system is live when no incorrect read operation has occurred). The liveness analysis evaluates the given property on the reachable states of the system to determine which states are live or not.

Liveness analysis have many applications such as proving functional correctness, selecting a viable and efficient schedule for a system, or designing run-time monitoring mechanisms to ensure the safe execution of the analyzed system. For example, the results of some liveness analysis can be used to handle deadlocks, usually within deadlock prevention and deadlock avoidance strategies, which are presented in the next subsection.

An example of liveness analysis is the one developed in the PRUNE framework [15]. This analysis is based on the PRUNE MoC, presented in Subsection 2.2.3, which is designed to allow the static analysis of liveness and memory boundedness. As stated in Subsection 2.2.3, this static analysis is made possible via a set of design rules, that have been devised to ensure the consistent behavior of dynamic actors. The PRUNE liveness analysis takes as input a representation of the system composed of an application graph, a platform graph, and a mapping of tasks to PEs. The application graph is similar to one of the input of the first and second contributions of this thesis (presented in Chapter 3 and Chapter 4 respectively), excepted that PRUNE uses the more flexible PRUNE MoC, whereas contributions 1 and 2 use AHSDF graphs. Our contributions also require a mapping of tasks to PEs, but do not require a graph describing the platform. Instead the only necessary information is the size of memories (for contributions 1 and 2), and an estimation of tasks' execution time and communication delays (only for contribution 1). It should be noted that contribution 1 is *not* a liveness analysis but a deadlock handling strategy, which are described in next subsection.

2.4.2 Deadlock Handling Strategies

The occurrence of deadlocks in multi-core systems is a common problem that is widely studied in the literature. Many mechanisms have been designed over the years to tackle the issue of deadlocks, with different strategies and types of deadlocks to remedy, depending on factors such as the requirements of applications, the system design, etc. In this thesis we focus on deadlocks caused by memory shortage, since they are the only kind possible in our target system (as explained in Chapter 1).

In this thesis deadlock handling strategies are classified in three different main

types, namely (i) prevention, (ii) detection and recovery, and (iii) avoidance. This classification is the most commonly used in the literature about deadlocks [19, 49, 65, 66]. These strategies have their own advantages and drawbacks, making them suitable for different kinds of scenarios.

Deadlock Prevention

A common approach to handle deadlock is to statically analyze the system and its possible configurations to make sure that under no circumstances a deadlock would occur. This is called deadlock prevention. The main drawback of this kind of strategy is its high computational cost and/or its restrictiveness.

Example deadlock prevention methods include:

- `memDAG` [56] is a tool developed by the research team behind StarPU [8]. It analyzes a graph derived from a task dependency Directed Acyclic Graph (DAG) to compute its maximal memory footprint. The maximal footprint is given by the topological cut with the highest footprint in the derived graph. If the maximal footprint of the Directed Acyclic Graph (DAG) is greater than the available memory on the system the DAG will run, `memDAG` computes artificial dependencies to prevent memory shortage from occurring. The `memDAG` tool proposes four different heuristics for the computation of artificial dependencies to respect the memory constraint. To avoid creating loops or meaningless dependencies, a dependency will always be added between two vertexes for which no paths previously existed. The `RespectOrder` heuristic has the advantage of never failing, but requires a valid schedule for the specified memory size to be computed beforehand. All the artificial dependencies it adds to the graph will respect the order of the provided schedule. The other three heuristics (`MinLevel`, `MaxSize`, `MaxMinSize`) have the drawback of sometimes failing to provide a valid solution even when one do exists. The purpose of `MinLevel` is to select artificial dependencies that minimize the critical path in the graph, in order to add the smallest overhead to the make-span of the application. `MaxSize` tries to minimize the weight of the next topological cut for each artificial dependency it adds. It adds a dependency by selecting the pair of vertexes which together contribute the most to the weight of the topological cut. `MaxMinSize` also tries to minimize the

weight of the next topological cut. It selects the pair of vertexes for which the minimum contribution among the two vertexes is the largest across all possible pairs. In other word, both vertexes contribute a lot to the weight of the topological cut. To better understand `MaxSize` and `MaxMinSize` let's take a simple example with four vertexes a, b, c, d respectively contributing $20, 5, 10, 10$ to the weight of the topological cut. We also assume that only two dependencies can be added: $b \rightarrow a$ or $c \rightarrow d$, as the other dependencies would create a loop or add no constraint as a path already exists between the vertexes. `MaxSize` will choose $b \rightarrow a$ since $\max(20 + 5, 10 + 10) = 25$ (the sum of the contributions of a and b is the maximal, 25). `MaxMinSize` will choose $d \rightarrow c$ since $\max(\min(20, 5), \min(10, 10)) = 10$ (the minimal contribution from $c \rightarrow d$ is 10, which is larger than the minimal contribution from $b \rightarrow a$ at 5).

- SynDEx [27] is a Computer-Aided Design (CAD) software for the development of real-time applications in embedded systems. SynDEx relies on two input graphs: the acyclic dependency graph of tasks, and the architecture graph representing the targeted platform. Then, by statically analyzing those graphs, it provides a static mapping and a static schedule guaranteed to be deadlock-free (i.e., deadlock prevention).

- PREESM [61] is a prototyping framework for dataflow applications. Its goal is to allow for the fast prototyping and deployment of applications to multi-core DSP systems, with stated goal to enable designers to perform Design Space Exploration (DSE). PREESM generates self-timed, deadlock-free code to run on the targeted system. The absence of deadlocks is ensured through the static analysis of the πSDF application model and the graph representing the target system to guarantee that the execution is memory-bounded and is not susceptible to a memory shortage.

- Stuijk et al., the authors of [67], developed a method for maximizing the throughput of multimedia applications running on MPSoCs, while minimizing buffers' sizes. In order to guarantee a maximal throughput the authors needed to produce deadlock-free schedules. They propose an exact approach which can be used for DSE, providing a Pareto front of memory requirement against max-

imal throughput. They also designed an approximate method with low overestimation of the buffers' sizes required to ensure deadlock freeness.

The first contribution of this thesis (presented in Chapter 3) is also a deadlock prevention method, computing artificial dependencies for dataflow applications running onto embedded systems. It differs from the methods presented above by its use of AHSDF graphs to describe applications. The most similar approach to our first contribution in its principle is `memDAG` [56] as it also computes artificial dependencies, but targets different types of systems. The approach of Stuijk et al. [67] and PREESM [61] have a similar target as contribution 1, as they also target dataflow applications running on embedded platforms, but uses different approaches to prevent deadlocks.

Deadlock Detection and Recovery

A totally different approach to handle deadlocks is the strategy of detection and recovery. The idea behind the approach is to monitor at run time the system in order to detect when a deadlock has occurred. If a deadlock is detected, the system is rolled back to previous *non-deadlocking* state, thus requiring the frequent save of checkpoints. It is efficient for systems where keeping checkpoints and dynamically monitoring the occurrence of deadlocks is (i) possible, and (ii) within a reasonable cost (in terms of time and memory overhead). This makes it suitable for systems where the occurrence of deadlocks is rare, since the restoration of the system to a previous checkpoint is an expensive operation, and with sufficient resources to spare in the monitoring process. This strategy is fully dynamic and useful for systems where static analysis for deadlock freeness would be too expensive in computation time, such as systems running highly reconfigurable applications.

The deadlock detection-based scheduling (DDS) [69] is an algorithm designed to detect deadlocks caused by the saturation of storage in HPC systems. The set of jobs to execute is represented through a weighted directed acyclic graph. Weights represent the estimated size of the input/output files to be written on the storage system. Once a deadlock has been detected, one or multiple jobs are selected to be rolled back and, as such, will have to restart their computation from scratch at a later point. This removal of jobs and their files frees storage capacity, thus enabling the

system to resume computations. Jobs are selected based on the quantity of storage space they currently use and are expected to require to complete their execution, as well as the expected computation time needed to restore their progress if removed.

Deadlock detection and recovery techniques are not suitable for the types of systems studied in this thesis. Indeed, the targeted embedded platforms have not enough resources both due to the limited computing power of such platforms, making the detection of deadlocks too computationally expensive, nor memory resources (with small Random-Access Memories (RAMs) and no mass storage such as hard disk drive) to save a system state for recovery.

Deadlock Avoidance

Deadlock avoidance is a hybrid approach to deadlock handling. This strategy is based on establishing beforehand some properties that, if respected, guarantee the absence of deadlocks. The idea is to then guide at run time the execution of the system to make sure that no decision (be it mapping, ordering, or timing) taken would cause the properties to stop being respected. This strategy has the advantage of a reduced run time overhead in comparison to detection and recovery, while being also usually less computationally expensive than deadlock prevention.

The Banker's algorithm [25] is one of the earliest and most fundamental work in the area of deadlock avoidance. This algorithm has been developed by Edsger Dijkstra in the sixties. It is designed to avoid deadlock in computer systems running concurrent processes. It requires the following information: the *maximal* amount each process may require of each resource, the current amount of *allocated* resources to each process, and the remaining *available* quantities of each resource. Since this algorithm is based on worst case scenarios for the allocation of resources, it usually does not provide maximally permissive solutions.

Reveliotis et al. [63] developed a deadlock avoidance technique for RAS. Their method allows for the computation of deadlock avoidance policies in polynomial time. These policies are shown to be exact (i.e., only forbid deadlocking schedules) in large subset case of single-unit RAS. Their approach can also be used for the more generic single-type RAS and conjunctive RAS, albeit without maximal permissiveness of the provided deadlock avoidance policy.

Thanks to the exact liveness analysis, presented in Chapter 4 of this thesis, an *exact* (i.e., maximally permissive) deadlock avoidance policy has been designed, and is deployed in the run-time environment presented in Chapter 6. This maximal permissiveness of this method gives more run-time flexibility for scheduling applications in comparison to the deadlock avoidance method presented in the preceding paragraphs.

2.4.3 A Comparison of Existing Methodologies for Handling Deadlocks

There exists a variety of methodologies to address the issue of deadlocks in computing systems. They vary with regard to their target system and problem specification. We further define the solution provided by a methodology to be *maximally permissive* if only schedules that effectively lead to deadlock are forbidden, while all theoretically valid schedules are allowed. In the opposite case, if some valid schedule cannot be selected due to the constraints imposed by the methodology, we say that the provided solution is *approximate*. Table 2.1 summarizes the characteristics of the different methodologies presented earlier in this section.

Table 2.1 presents a comparison of different deadlock handling tools and algorithms, including which strategy they use, their targeted system, and whether they are maximally permissive (i.e., allow *all* schedules) or not. Depending on the characteristics of the targeted systems, many deadlock handling strategies have been developed. For example, the use of Petri nets led to the development of many deadlock handling techniques for flexible manufacturing systems, be it prevention [9, 43, 51], detection and recovery [18, 30, 74], or avoidance [17, 31, 73]. An extensive survey of Petri net based techniques has been made by Hou and Barkaoui [42].

Deadlock detection and recovery strategies have the advantage of requiring the less amount of computation, but are only viable if the occurrence of deadlocks can be monitored at run-time and a recovery mechanism can be put in place. This is only suitable in systems for which deadlocks are a rare occurrence and the recovery a possible and not too expensive possibilities. As such, deadlock detection and recovery are very infrequently deployed in embedded systems, as resources are scarce and performance critical.

Table 2.1: Comparison of a selection of deadlock handling tools and methodologies

Methodology	Strategy	Problem specification	Target system	Maximally permissive
`memDAG` [56]	prevention	dependency graph	HPC	no
SynDEx [27]	prevention	acyclic dependency graph	real-time/embedded	no
PREESM [61]	prevention	πSDF	embedded	no
Stuijk et al. [67]	prevention	CSDF/SDF	embedded	yes & no[1]
DDS [69]	detection and recovery	acyclic dependency graph	HPC	yes
Banker's algorithm [25]	avoidance	see above	generic computing	no
Reveliotis et al. [63]	avoidance	Conjunctive RAS	flexible manufacturing systems	no
Contribution 1	prevention	AHSDF	embedded	no
Contribution 3	avoidance	AHSDF	embedded	yes

[1] A maximally permissive solution is provided, as well as an approximate heuristic

Deadlock prevention solutions offer the advantage of not requiring dynamic computations, which is very valuable in embedded systems with limited computing power. Their drawbacks are their high static computational cost, which might render them not suitable for large workloads, or their otherwise limited permissiveness.

Deadlock avoidance strategies are usually more permissive than deadlock prevention, at the expense of some run-time overhead. They may be deployed in embedded systems, provided that their dynamic overhead is limited. Deadlock avoidance strategies are usually preferred over deadlock prevention in embedded systems when they allow for more flexibility for the scheduling.

2.5 Run-Time Environments for Dataflow Applications

Run-time environments are used to manage the execution of dataflow applications at run time. They are used in a wide variety of computing systems, ranging from

embedded systems, general purpose computing, real-time computing and HPC. A run-time environment should provide memory allocation and management functionalities, which may include memory defragmentation, cache management, and/or paging in some systems. It also manages the dynamic part of the scheduling which, depending on the scheduling strategy, can include (1) the timing of tasks, (2) their ordering, (3) the mapping of tasks onto PEs and buffers to memories, and (4) the extraction of potential parallelism. Run-time environments with more functionalities tend to incur a higher run time overheard, and are mostly used in high-end systems such as the ones used for HPC. On the other hand, embedded systems with their limited capabilities commonly use run-time environment providing a more reduced set of functionalities, to minimize the run time overhead.

Another advantage of run-time environments is that they can be used to hide away the complexity of the underlying hardware. Over the years commercially available platforms became more complex in order to improve performance, once the pace of increase in raw performance of generic CPUs (for example by rising their frequency) slowed down, faced with constraints such as manufacturability, reliability, or energy consumption. For example specialized PEs such as DSPs for signal processing and GPUs for graphic computing have been designed. This led to heterogeneous computing architecture, in which tasks can only be performed on some PEs or have varying performance depending on their mapping, as well as the rise of Non-Uniform Memory Access (NUMA) architectures, since those specialized PEs often used dedicated memories for their computations.

Example run-time environments for applications based on task/dataflow models include:

- StarPU [8] is a run-time environment targeting heterogeneous platforms for High-Performance Computing (HPC). StarPU maps and schedules tasks at run time.

- The XKaapi [33] run-time environment targets HPC platforms with heterogeneous architecture composed of multiple CPUs and GPUs. It superseded the KAAPI run-time environment developed by the same research team [32]. The computations to be executed are represented via tasks, which are mapped and scheduled at run time. Furthermore, XKaapi includes a *work-stealing* mecha-

nism, that allows an idle PE to steal a task initially slated for execution onto another PE. This mechanism leads to an improvement in performance by minimizing the idleness of PEs, and has been shown to be as efficient as the default configuration of the StarPU scheduler [50].

- PRUNE [15] is a hybrid run-time environment for dataflow applications running on heterogeneous platforms. It uses an extension of SDF where connections between actors can change at run-time, allowing for the representation of reconfigurable applications while still being statically analyzable for memory boundedness and deadlock freeness. The PRUNE run-time environment does not provide a deadlock handling system *per se*. Instead, it relies on the static liveness analysis (presented in Subsection 2.4.1 of this chapter) to ensure that the running system (composed of an application graph, a platform graph, and a mapping) is memory-bounded and deadlock-free, thus removing the need to use any run-time deadlock handling strategy.

- The HTGS Model-Based Engine (HMBE) [70] targets homogeneous multi-CPUs platforms. It is a high-level (model-based) abstraction on top of the Hybrid Task Graph Scheduler (HTGS) Application Programming Interface (API) [13, 14]. It should be noted that this underlying API supports heterogeneous, multi-CPUs multi-GPUs platforms. HMBE provides a run-time environment with a dynamic mapping and scheduling of tasks. Tasks are assigned dynamically to worker thread, which run concurrently on the PEs of the targeted platform.

- SPIDER [39, 40, 58] is a run-time environment for real-time dataflow applications running onto heterogeneous platforms. Applications are described using the πSDF formalism, allowing reconfiguration at run-time. SPIDER performs mapping and scheduling decisions at run-time, in a centralized fashion.

We presented here only model-based run-time environment for dataflow applications. There also exists many APIs which provide run-time environments when implementing dataflow application at the language-level, such as the aforementioned HTGS [13, 14], as well as Charm++ [44], Qilin [55], and Harmony [24] among others.

Table 2.2 displays a comparison of different existing run-time environments. Out of the presented environments, HMBE [70] and *Contribution 3* of this thesis (pre-

Table 2.2: Run-time environments for task-based and dataflow-based applications

Feature	StarPU	XKaapi	PRUNE	HMBE	SPIDER	*Contribution 3*
Heterogeneity	Yes	Yes	Yes	No	Yes	Yes
Dynamic mapping	Yes	Yes	Yes	No	Yes	No
Dynamic scheduling	Yes	Yes	Yes	Yes	Yes	Yes
Deadlock handling	Yes	No	No[1]	No	No	Yes
NUMA support	Yes	Yes	undef.	Yes	Yes	Yes
Reconfigurable applications	Yes	Yes	Yes	No	Yes	No
Targeted system	HPC	HPC	general	real-time	real-time	embedded

[1] PRUNE run-time is used to execute applications which are guaranteed to be deadlock-free, via the PRUNE static liveness analysis.

sented in Chapter 6) have a scheduling of the *static assignment* kind, whereas the scheduling done by the other environments are *fully dynamic*. The *third contribution* of this thesis differs from existing run-time environments by the following: SPIDER [39] offers no deadlock handling mechanism, HMBE [70] does not support heterogeneous architectures, and the other available run-time environments are not suitable for embedded systems, as they target platforms with larger computing resources such as HPC systems.

2.6 Conclusion

There exists many Models of Computation which can be used to represent dataflow applications. These MoCs have different properties with regard to their expressiveness (what they can or cannot represent) and their analyzability (what properties do they allow to analyze, and with which complexity). In the context of this thesis, we settled on using the SDF and AHSDF formalisms, as they are the less complex MoCs allowing to represent our target systems: dataflow applications running onto embedded systems.

Contributions 1 and 2 of this thesis aim at providing deadlock handling mechanism for dataflow applications running onto heterogeneous architecture, while contribution 3 provides a run-time environment satisfying all our criteria. The aim of this thesis is to enable the deployment of dataflow applications onto heterogeneous, NUMA platforms, and ensure their safe (non-deadlocking) execution. As such, there

are currently no run-time environments satisfying these criteria as described at the end of Section 2.5 of this chapter. This raises the need to either extend an existing run-time environment targeting embedded systems, or to design our own run-time environment. Chapter 6 discusses in more details whether and how the deadlock avoidance strategy derived from contribution 2 could be deployed onto existing run-time environments, and describes a new run-time environment satisfying all our criteria.

Part II

Contributions

Chapter 3

Approximate Deadlock Prevention using Memory Exclusion Graphs

3.1 Introduction

The occurrence of memory overflow in embedded systems leads to data corruption and prevents the correct execution of applications. Because of this, in most embedded systems, a hardware-software combination prevents memory corruption by guaranteeing that data buffers are never consumed before they are produced and never overwritten before they have been consumed by all consumers. In turn, these mechanisms can lead to memory shortage situations where the execution can progress only after buffers have been allocated in memories that cannot accommodate them. Many solutions exist in generic PCs to counteract memory shortages and avoid the deadlocks they cause at run time. This is the case for example of deadlock detection and recovery, which reverts the system to a previously saved safe state, or of the offloading, which temporarily moves data buffers into some high capacity storage. Deploying these solutions in embedded systems is often not possible, due to their tight requirements on the cost, physical size, and performance. A solution fitting those requirements is the use of a static memory usage analysis, to guarantee

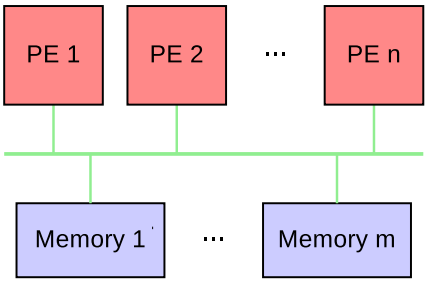


Figure 3.1: Example logical architecture of a target platform.

at design time that no memory shortage will occur once the system is online, thus removing the need to deploy costly run-time fixes.

In this chapter we present the **first contribution** of this thesis, a method to prevent deadlocks resulting from memory shortages. It is based on the analysis, at design time, of the potential for memory shortage when running a set of applications, (hereafter referred as a *workload*) onto a target platform. The analysis uses a graph that represents the allocations of buffers and whether said buffers can or cannot be allocated simultaneously. It allows to detect potential memory shortages and to prevent them by adding precedence constraints between tasks in the workflow.

The remainder of this chapter is structured as follows: Section 3.2 presents the characteristics and hypotheses made on the target platforms, and Section 3.3 the modeling of the workload and the constraints on the applications its represent. The memory usage analysis and its derived deadlock prevention method are presented in Section 3.4. Finally, Section 3.5 concludes this chapter. The evaluation of this contribution is deferred to Chapter 5.

3.2 Target Platform

The memory usage analysis presented in the chapter applies to platforms such as the one presented in Figure 3.1. The platform has three types of architecture elements: computing resources, the platform memories, and the interconnect linking them together. They are displayed respectively in red, blue, and green in Figure 3.1. It should be noted that Figure 3.1 shows a simple example, but the communication

links between computing resources and memories can follow more complex structure, such as in NUMA architectures (cf. Figure 1.2b).

Computing resources are a set of n Processing Elements (PEs) responsible for the execution of tasks. A PE can be any sort of computing unit such as a CPU, a GPU, a hardware accelerator, etc. Therefore PEs are allowed to provide widely different capabilities, meaning that the target platform is not limited to homogeneous architectures, heterogeneous architectures are also supported. The following hypothesis is made with regard to the computation behavior of PEs:

- H_1: Each PE can execute only one task at a time, without preemption, until the task's completion.

For example under assumption H_1 each core of a multicore CPU should be considered as its own PE.

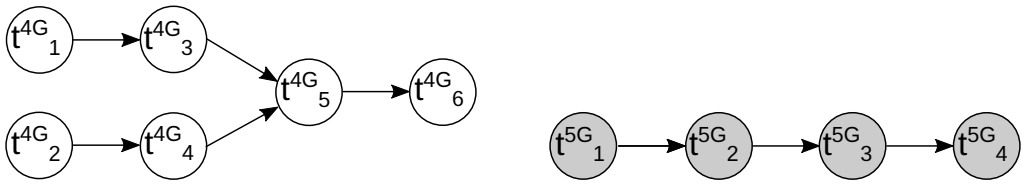
The target platform has a set of m memories in which buffers are stored. Each memory of the target platform has a specified size, expressed in data units. In the following all memory sizes are expressed in the same data unit. Platforms with a single memory are usually Uniform Memory Access platforms, but might as well be Non-Uniform Memory Access platforms, for example if the interconnect is not symmetrical for different PEs. Platforms with multiple memories are often NUMA, but can also be UMA.

Studying interconnects is outside the scope of this thesis. As such it is supposed that the interconnect is reliable, by which is meant the following:

- H_2: There is no risk of deadlock or livelock on the interconnect;
- H_3: There is no loss of data during transfer (for example between a memory and the cache of a PE).

3.3 Applications and Workload

The applications selected for execution on the target platform are modeled by dependency graphs, such as the ones displayed in Figure 3.2. Vertexes represent tasks, i.e., computations to be executed, and edges represent Input/Output buffers used for the communication between tasks. Edges are annotated with a positive integer that is the size of the buffer in memory (not represented in Figure 3.2). The memory needs of a task are considered in terms of Input/Output buffers (i.e., where data



(a) Example dependency graph of a 4G application.

(b) Example dependency graph of a 5G application.

Figure 3.2: Example applications' dependency graphs.

are stored, before and after processing). For systems where it is relevant to also consider the internal memory of tasks (e.g., to store intermediate results), requests for this additional memory can be simply added to those for the Input/Output (I/O) buffers in the model.

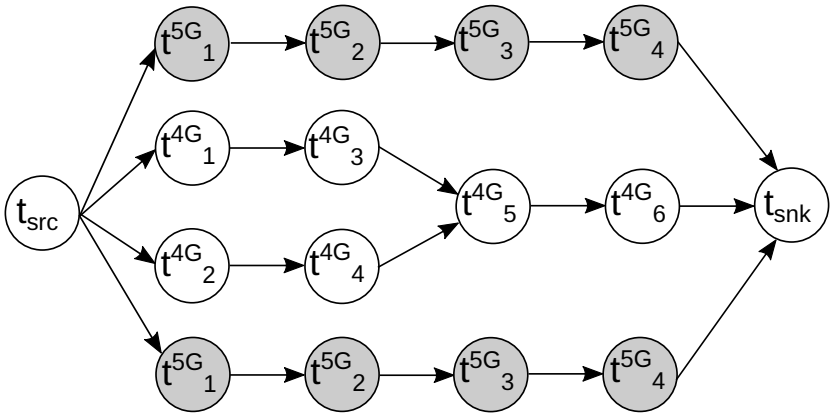
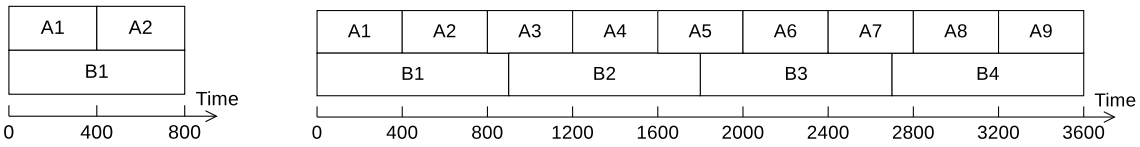


Figure 3.3: Example workload composed of one iteration of the 4G application and two concurrent iterations of the 5G application

A platform executes a **workload** that is modeled with an annotated dependency graph (a Directed Acyclic Graph (DAG)), as illustrated by Figure 3.3. The workload graph is the union of each application's dependency (AHSDF) graph, where artificial source and sink tasks connect the source and sink vertexes, respectively, of each application. The source and sink incident edges have null weights. Note that a dependency graph does not specify any execution semantics for its vertexes. Therefore, to capture multiple concurrent executions of periodic applications, a separate copy of the application dependency graph should be added in the workload graph for each period that can run concurrently. This can be seen in Figure 3.3, which



(a) Application A has a period of 400, application B of 800.

(b) Application A has a period of 400, application B of 900.

Figure 3.4: Two example application extensions to a common period.

represents a workload composed of two concurrent iterations of the 5G application (Figure 3.2b) and one iteration of the 4G application (Figure 3.2a).

In the context of the contribution presented in this chapter, applications in a workload graph should have a defined deadline, and share the same period. The requirement to share multiple periods is necessary as the analysis is based on timing informations deduced from a global deadline. In other words, the timing information of a running application are also dependent on the other running applications. When multiple applications do not share the same period, it is possible to *extend* applications, i.e., to represent multiple successive (non concurrent) iterations of the applications in order to get to a common period. The minimal possible value for this common period is the LCM of the application periods. For example in Figure 3.4a, application A is extended to two iterations to reach a common period of 800. In Figure 3.4b, A is extended to 9 iterations and B to 5, to reach a common period of $\mathrm{lcm}(400, 900) = 3600$. Hence the application periods have an important effect on the extension process, and therefore on the size of the workload graph. The best-case scenario for this extension to a common period is when the base applications have *harmonic* periods. Two applications are said to have harmonic periods when the period of application with the largest period is a multiple of the period of the application with the shortest period. More generally, a set of applications is said to be harmonic if the least common multiplier of the application periods is equal to the maximal application period of that set. For example the applications in Figure 3.4a are harmonic, since the period of application B is double the period of application A. Applications in Figure 3.4b are *not* harmonic, which leads to a larger common period and therefore a larger number of successive iterations to represent in the workload graph.

The workload graph $H = (\mathcal{T}, \mathcal{B})$ is similar to the applications' dependency graphs. It has a global deadline, which represents the time at which the sink task should have been completed. Vertexes $\mathcal{T}$ are the tasks that execute on the PEs of the platform. Each vertex t is annotated with the following:

- The *mapping*, i.e., which PE of the target platform should execute the task represented by t. Tasks are statically mapped to a PE in the target platform and cannot migrate at run time. This is necessary to compute the due-date which is described bellow;
- A Worst-Case Execution Time (WCET) denoted $wcet_t$ of the task represented by t on that PE. The WCET is the maximal amount of time the task is guaranteed to require for its complete execution.
- The *due-date* dd_t of the task represented by t. The due-date metric is an upper-bound on the completion time of a task which, if missed, will cause a deadline miss for said task or at least one of its descendants. The due-date is obtained by subtracting from the global deadline a lower-bound estimation on the execution time of the critical path from t to the sink. We compute the due-date using Algorithm 1, first described by Adyanthaya et al. [6]. The due-date is equal to or tighter than the conventional deadline, as it takes into account the mapping information and how multiple tasks assigned to the same PE will not be able to run in parallel. If the due-date is negative, it means that all possible schedules will cause a deadline miss.

Edges $\mathcal{B}$ denote precedence constraints and the Input/Output task buffers that are allocated in memory. As is the case for application graphs, each edge is annotated with a positive integer that is the buffer size (not represented on Figure 3.3).

The computation of due-dates is performed by on the workload graph in a reverse topological order, i.e., each task get its due-date computed after the due-dates of all its successors have already been computed. Each task's due-date is computing following Algorithm 1. It is obtained by computing a local due-date for the task for each PE, and then taking the minimal value across all PEs (hence the tightest due-date). The due-date of the workload sink is defined to be equal to the global deadline.

All the hypotheses made on the workload are recapitulated bellow.

- H_4: All applications have a deadline, and share the same period (or have been extended to share the same period).

input
  $H = (\mathcal{T}, \mathcal{B})$; // workload graph
  n; // number of PEs in targeted platform
  t; // task for which to compute the due-date
output
  dd_t; // due-date of task t
define
  $H.succ_k^{ordered}(t)$; // list of immediate successor tasks of t in H mapped to PE k and ordered by descending due-date

$dd_t \leftarrow \infty$;
for $k \leftarrow 1$ **to** n **do**
  $local\_dd_t \leftarrow \infty$;
  for $t' \in succ_k^{ordered}(t)$ **do**
    $local\_dd_t \leftarrow \min(dd_{t'}, local\_dd_t) - wcet_{t'}$;
  end
  $dd_t \leftarrow \min(dd_t, local\_dd_t)$
end

Algorithm 1: Compute the due-date of a task.

- H_5: Tasks are statically mapped to a PE in the target platform and cannot migrate at run time.
- H_6: All tasks have a defined *due-date* and *WCET*.

3.4 Memory Shortage

A memory is at risk of shortage when the memory occupation is at or near the capacity of the memory. In conventional Personal Computers (PCs) or HPC systems, some data buffers can be offloaded from the (local) memory onto a large capacity storage system. This has an impact on performance as these type of storage, be it a large capacity RAM or a hard-drive, usually have orders of magnitude higher access times, but it has no impact on the ability to properly finish the execution of the application. This offloading approach is usually not practical in the kind of platforms targeted in this thesis, where there is no such large capacity storage available for offloading and/or no time for these data transfers. As such, the occurrence of memory shortage is a serious problem that prevents the correct execution of applications

and leads to deadlocks if not mitigated. This led to the design of a memory usage analysis, which is the first contribution of this thesis.

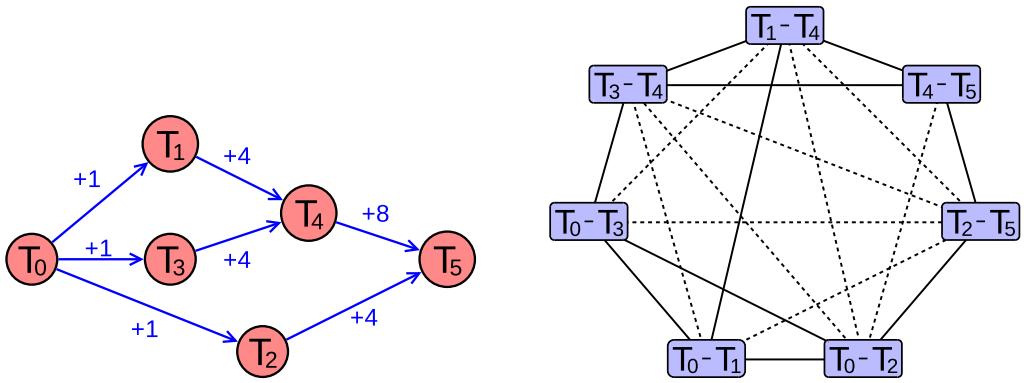
3.4.1 Memory Exclusion Graph

The optimization of memory management is a great concern in embedded systems. For example Hadj Salem, Kieffer, and Mancini studied the issue of memory management in embedded vision systems [35–38]. In 2015, Desnos et al. introduced the Memory Exclusion Graph (MEG) [21], a type of graph computed from an AHSDF graph and used to analyze its memory allocations and requirements. In [22] MEGs have been used to minimize the memory footprint of dataflow applications by studying memory reuse opportunities when executing actors, such as storing the output data of an actor at the same location as some of its input data that will not be used again by the application.

A MEG is an undirected weighted graph where vertexes represent indivisible memory objects that correspond to communication buffers in a SDF graph, the working memory of SDF actors, and feedback FIFOs that store initial tokens in a SDF graph. Edges in a MEG represent exclusion relations, i.e., the impossibility to share physical memory as the buffers can be allocated simultaneously. More formally, a MEG is an undirected weighted graph $M = (V, E, w)$ where:

- V is the set of vertexes. Each vertex represents a buffer of the AHSDF graph.
- E is the set of edges representing the memory exclusions, i.e. the impossibility to share memory. We denote $(v_0, v_1) \in E$ the edge between vertexes v_0 and v_1 of V.
- $w : V \rightarrow N$ is a function giving $w(v)$, the weight of a vertex v. The weight of a vertex corresponds to the size of the associated buffer.

Two buffers are in a relation of memory exclusion, whereby they cannot be allocated at overlapping locations in memory, if there exists a schedule of the AHSDF graph such that both buffers would be allocated simultaneously. Some exclusions can arise from the properties of the buffers, such as input and output buffers of a task being allocated simultaneously when that task is executing. Others come from the parallelism of an application, as is the case with buffers used by tasks that can be executed concurrently.



(a) Example AHSDF graph.

(b) Corresponding MEG, dashed edges are exclusion edges caused by parallelism.

Figure 3.5: Example AHSDF and its corresponding MEG.

The construction of a MEG is based on the following hypotheses over the memory management of the target platform:

- H_7: Output and working buffers of tasks are allocated instantaneously, immediately before the task starts execution.
- H_8: A buffer is deallocated when it becomes dead, i.e., as soon as the last live consumer task has completed.
- H_9: A buffer is never deallocated if it is alive, i.e., if there exists at least one live task that requires it and did not yet complete.

The left-hand side of Figure 3.5 shows an example AHSDF graph, and the right-hand side its corresponding MEG.

Edges represented by continuous lines in Figure 3.5b correspond to buffers used by the same task and that therefore must be present in memory simultaneously. They represent mandatory exclusions that will occur in all schedules, such as the exclusion between buffers $T_0 - T_2$ and $T_2 - T_5$ since both buffers are necessary for the execution of T_2.

Edges represented by dashed lines in Figure 3.5b correspond to buffers used by tasks that can execute concurrently and therefore may be required simultaneously. They represent exclusions that may or may not occur depending on the scheduling

of the AHSDF graph. This is the case of the exclusion between buffers $T_0 - T_2$ and $T_1 - T_4$, caused by the potential parallelism between T_2 and either of T_1 and T_4, that will not occur if T_2 is not run concurrently to both T_1 and T_4.

Finally, in Figure 3.5b there are no exclusion edges between $T_0 - T_1$ and $T_4 - T_5$ on one hand, and $T_0 - T_3$ and $T_4 - T_5$ on the other hand. This is the case because all tasks requiring $T_0 - T_1$ or $T_0 - T_3$ (i.e., T_0, T_1, and T_3) must have been completed before any task requiring $T_4 - T_5$ (i.e., T_4 and T_5) can start their execution.

From this overview of MEGs, it can be deduced that some of its exclusions, those of the second type, can be removed. Consequently the maximal memory footprint can be reduced by limiting the parallelism between some concurrent tasks.

So far, we presented MEGs as computed from an unscheduled workload graph. In this case the MEG is called a pre-scheduling MEG, and will represent the memory exclusions from all possible schedules. Instead, if a MEG is computed from a fully scheduled workload graph it is called a post-scheduling MEG, and will represent only the memory exclusions that *do* occur when executing the workload with the specified schedule. The set of edges in a post-scheduling MEG will always be a subset of the set of edges in the pre-scheduling MEG of the same workload. More precisely, all edges of the first type will be part of any post-scheduling MEG, but edges of the second type may be removed. It should be noted that it is not always possible to find a schedule such that all edges of the second type would be removed. Even when possible, a schedule leading to the removal of all edges of the second type necessarily corresponds to a sequential execution of the applications composing the workload graph, and is therefore of little practical interest. For example, in Figure 3.5 it is not possible to remove simultaneously the edges $(T_0 - T_1, T_3 - T_4)$ and $(T_0 - T_3, T_1 - T_4)$. Indeed, removing the former requires to have terminated the execution of T_1 before starting the one of T_3, whereas removing the latter requires the opposite, meaning that no schedule can lead to a post-scheduling MEG in which both edges are removed.

3.4.2 MEG Analysis

Memory Exclusion Graphs can be used to analyze the memory allocation and requirements, and as such, to analyze whether memory shortages might occur. The

shortage-prevention technique presented here uses this fact to look if memory shortages are a potential problem and, if so, annotate the workflow graph with artificial dependencies until it safe, shortage-free, execution can be guaranteed. This means that there exists no possible scheduling of the annotated workload graph leading to shortage, therefore removing the need for any dynamic recovery or deadlock prevention mechanism.

The memory usage analysis is based on the analysis of cliques in the MEG computed from the workload graph, as a clique represent of set of buffer that might be stored in the same memory of the system at the same time. As such, if all buffers constituting a clique cannot fit inside the memory a shortage could occur, meaning that the allocation of at least one of these buffers could stall indefinitely, leading to a deadlock in the system.

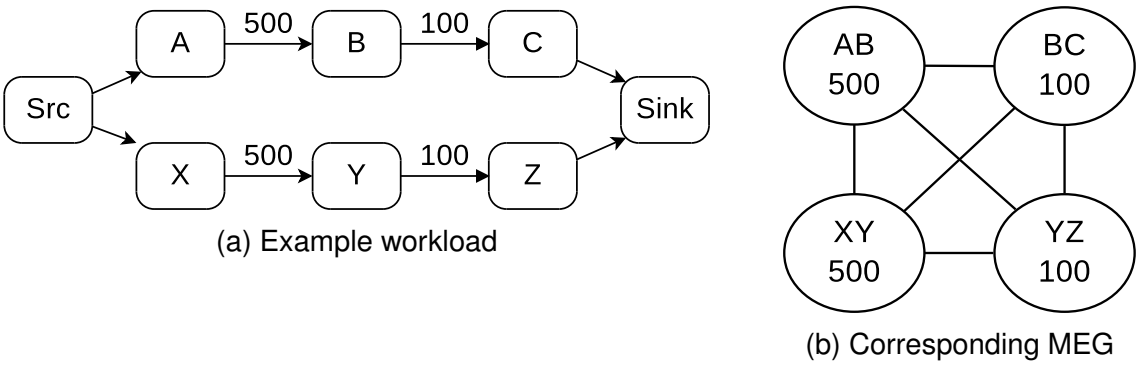
Since a MEG is used to represent memory exclusion relations in a single memory, the analysis of multi-memory systems is done by applying the analysis to the MEG of each memory, until no MEGs are left with clique not fitting inside their corresponding memory.

Input

Different types of information serve as input: (i) one or more application graphs annotated with real-time constraints (tasks' deadlines) and (ii) mapping information that assigns tasks to PEs, specifies the size of the memories, associates estimates of the tasks' WCET.

Application model — Each application SDF graph is first transformed, as described in Subsection 2.2.1, into a directed Acyclic Homogeneous Synchronous Data Flow (AHSDF) graph $H = (\mathcal{T}, \mathcal{B})$. Tasks in $\mathcal{T}$ are associated to identical (homogeneous) production and consumption rates on FIFOs buffers in $\mathcal{B}$. The result of this transformation on the example SDF graph can be seen at bottom of Figure 2.1. This transformation is necessary to expose data parallelism and memory allocation options (Homogeneous SDF) as well as to isolate one iteration of the algorithm captured by the original SDF graph (Acyclic Homogeneous SDF).

Platform model and mapping information — A generic logical architecture of the target platforms can be seen in Figure 3.1. These platforms are composed of a set of n Processing Elements (PEs) and m memories.



(a) Example workload

(b) Corresponding MEG

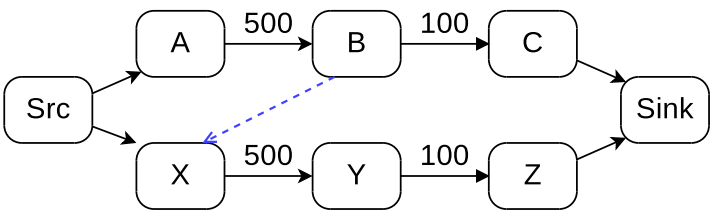
Figure 3.6: Example applications, whose concurrent execution may cause a memory shortage

3.4.3 Memory Shortage Prevention

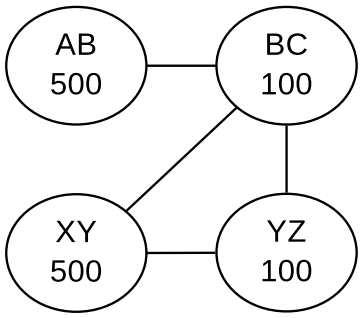
As described in [21], AHSDF graphs can be updated with scheduling constraints. These constraints have the effect of removing exclusion relations (edges) in the corresponding MEG. The analysis is composed of two phases. First, for each memory a MEG is created from the AHSDF graph of all applications, and pruned of all exclusions that are not possible under the timing constraints and task WCETs. The second phase iteratively removes potential memory shortages seen in the MEGs by adding a precedence constraint using a heuristic, preventing the simultaneous allocation of the set of involved buffers. Of course, the new precedence constraint must be selected in such a way that the AHSDF graph remains acyclic, since violating this characteristic would prevent the full execution of the workload. If no such precedence constraint is found by the analyzing algorithm, the memory shortage prevention fails. This can happen if the memory size is smaller than the required minimum to perform the workload execution, or if previously added constraints do not allow the heuristic to add a valid new precedence constraint.

As explained above, a memory shortage may occur if a circular wait happens due to a memory being unable to host a buffer. This situation can be detected by inspecting cliques in a MEG $M = (V, E, w)$. A clique of an undirected graph is defined as a subset of the vertexes such that every two distinct vertexes in the clique are adjacent. A clique C of M is thus such that:

$$C \subseteq V, \ \forall \ v_0 \in C, v_1 \in C, v_0 \neq v_1, (v_0, v_1) \in E$$



(a) Example workload, updated with a scheduling constraint (dashed line)



(b) Updated MEG

Figure 3.7: Updated applications, free of memory shortages

The weight of C is the sum of the weights of its vertexes:

$$w(C) = \sum_{v \in C} w(v)$$

It defines the maximum amount of memory that must be allocated to store the buffers in C. This is regardless of the scheduling policy for the tasks that produce/consume the buffers represented by vertexes of a MEG. Therefore, if the weight of a clique exceeds the memory capacity, a memory shortage may occur at run time. Such cliques will be referred as *oversized cliques* hereafter. We define a minimal oversized clique as an oversized clique such that removing any buffer from it would lead to a clique that is not oversized.

In case of pre-scheduling MEGs, the presence of an oversized clique is a necessary (but not sufficient) condition for a memory shortage to occur. For post-scheduling MEGs, the presence of an oversized clique becomes a necessary and sufficient condition. For example, let's consider the scenario where two applications can run simultaneously, as shown in Figure 3.6a that represents the corresponding cluster of partitioned AHSDF graphs. These applications run on a platform with one PE and a memory of capacity 1000. Executing these applications will lead to a memory shortage if producer tasks A and X are scheduled to run before both consumer tasks B and Y. This is the case because neither output buffer BC nor YZ can be allocated. This is visible in the corresponding MEG (Figure 3.6b) as the clique $\{AB, BC, XY\}$ (and its symmetric equivalent $\{AB, XY, YZ\}$) is of size 1100, thus larger than the available memory capacity.

```
1  input
2  |  M;                                   // Memory Exclusion Graph to analyze
3  |  mem_size;                            // size of the memory
4  output
5  |  OC;          // minimal oversized clique of M (or empty set if none found)
6  define
7  |  size(clique);                        // sum of the sizes of all buffers in clique
8  OC ← ∅;                                                        // initialization
9  max_cliques ← find_maximal_cliques(M);
10 for clique ∈ max_cliques do
11 |  if size(clique) > mem_size then
   |  |  // reduce to minimal oversized
12 |  |  forall buffer ∈ clique do
13 |  |  |  if size(clique \ {buffer}) > mem_size then
14 |  |  |  |  clique ← clique \ {buffer};
15 |  |  |  end
16 |  |  end
17 |  |  OC ← clique;
18 |  |  break;
19 |  end
20 end
```

Algorithm 2: Find a minimal oversized clique in a MEG

The mechanism to prevent memory overflows is based on finding minimal oversized cliques in an unscheduled MEG. A possible way to find such cliques is to use the algorithm described in Algorithm 2. To find a minimal oversized clique the first step is to look for a maximal clique of the MEG that is oversized, and then remove vertexes from it until a minimal oversized clique is obtained. A maximal clique is a clique to which no vertexes can be added without resulting in a graph that is not a clique. If there is an oversized clique in the MEG, a scheduling constraint is added to the workload graph, from the consumer task of a buffer in the clique to the producer task of another buffer of the clique (while ensuring that it would not create a loop in the workload graph). The MEG is then updated to remove exclusions links not applicable with the scheduling constraint in place. The algorithm looks specifically for minimal oversized cliques because removing a minimal oversized clique also removes any larger clique containing all buffers of that minimal oversized clique, thus reducing the

number of scheduling constraints added, and the number of times the algorithm must be executed. The additional scheduling constraint corresponds to the dashed link in Figure 3.7a, and the updated MEG is shown in Figure 3.7b. Adding scheduling constraints prevents from having all the producer tasks of buffers which are part of the clique from being executed while none of the corresponding consumers have. This avoids having to fit all the buffers of the oversized clique in memory at the same time, thus preventing the shortage. In this example, the prevention of memory shortage is visible in the updated MEG in Figure 3.7b, as the oversized clique $\{AB, BC, XY\}$ is no longer present. The remaining maximal cliques $\{AB, BC\}$ and $\{BC, XY, YZ\}$ are not oversized, meaning that there are no oversized cliques left in the MEG.

Since finding the cliques in a MEG is computationally expensive (the clique decision problem is NP-complete [45]), the proposed analysis to prevent memory overflow should be executed statically. The minimal oversized clique returned by Algorithm 2 is dependent on the order of selection of the maximal cliques (line 10), and on the order in the buffers from said clique will be explored (line 12). This variability in the returned minimal oversized clique in turn impacts the artificial constraint which is added, thus having a potential effect on the quality of allowed schedules, but this aspect has not been studied in this thesis. It should also be noted that Algorithm 2 is only one way to find minimal oversized cliques. Another possible way would be to construct a minimal oversized clique by selecting buffers one by one, each additional buffer being in exclusion relations with all previously selected buffer. Buffers could be selected by their size (from the largest to the smallest) or by the due-date of their producer for example. This process would be repeated until either a minimal oversized clique is found, or, if not, a selected buffer would be removed to continue the exploration, as removing a buffer can allow adding buffers that were not selectable previously. Again, studying alternative algorithms to find minimal oversized clique has not been studied during this thesis.

The MEG analysis described here should be made on the worst-case scenario for the system, where all applications in the workload run simultaneously, to produce correct results. This is a limitation since the clique finding is NP-complete. There are potential solutions to reduce the size of the problem. A first option is to define a set of worst-case scenarios, where each scenario only has a proper subset of all applications running concurrently. With this approach, it should be forbidden at run-time

to run simultaneously a set of applications that is not covered by one of the studied scenarios, i.e., the set of running applications must be included or equal to the application set of at least one scenario. This makes sense and should definitely be used in situations where there are known incompatibilities between two or more applications that completely forbid to run them simultaneously. Examples of this can be found when two different modes (e.g. size of a modulation, rate of a channel decoding...) of the same application are represented as two independent applications. Depending on the currently active mode, only one or the other application variant will run but not several. Of course this approach is limited by the total number of combinations to analyze, that increases rapidly if too many applications have too many variants. A second approach is to divide the system into two or more subsystems and analyzing them separately, each having its own dedicated part in the memories of the target platform. The drawback of this option is the underuse of memory resources, since one subsystem cannot use memory space that other second subsystem might not be using at a given instant. On the other hand, this approach has the advantage of partially lifting the same period requirement, since each subsystem can have an independent period. This means that applications from different subsystems are no longer required to be extended to a common periods, but applications that are part of the same subsystem are still subject to this extension process if they do not share the same period. This second approach requires to carefully select the application sets and memory sizes of the different subsystems to minimize memory underuse.

3.5 Conclusions and Limits

In this chapter the **first contribution** of this thesis was presented, a new approach to prevent memory shortages and the deadlocks they cause when scheduling periodic dataflow applications on parallel and heterogeneous platforms. The additional cost of these computations means that they should be executed during the design phase of the system. The memory usage analysis is based on the computation of cliques in Memory Exclusion Graphs, and produces artificial dependencies that constrain the scheduler to only select safe, non-deadlocking schedules, with minimal *dynamic* computational overhead.

The technique to prevent memory shortages presented in this chapter is limited

for its practical use by multiple factors. One factor is the constraint on application periods being identical or extended greatly reduces the application sets for which this technique can be used, since the extension process can lead to a large increase in the size of the system to analyze. A second lies with the quality of the proposed solution, which is greatly dependent on (1) the order in which oversized cliques are selected, (2) the order in which buffers are removed from oversized cliques to build minimal ones, and (3) on which artificial dependency is finally added to the AHSDF graph to remove the considered minimal oversized clique. Finally, the execution time of the memory shortage analysis is quite high for an approximate method, as is shown in the experimental evaluation of Chapter 5. With these limitations another method that is less restrictive on the sets of application it can support, and a behavior not dependent on its specific implementation has been developed. This second method is presented in next chapter (Chapter 4), and has been implemented in a run-time environment described in Chapter 6.

Chapter 4

Efficient Liveness Analysis

4.1 Introduction

In this chapter the **second contribution** of this thesis is presented. It is a liveness analysis targeting platforms regardless of their memory architecture (UMA, NUMA). This method is based on the analysis of an automaton, called the control automaton, whose state-space is a subset of the state space of the conventional automaton representing all possible schedules. This chapter demonstrates that analyzing this subset of the full state-space is sufficient to guarantee the absence of deadlocks, without any loss in the precision of the analysis in comparison to analyzing the full state-space. The reduction of the state-space to analyze makes it possible to run an exact deadlock prevention analysis in a reasonable amount of time for larger systems than would be possible using the conventional approach.

In comparison to the method presented in the previous chapter this method can be used to produce exact deadlock handling strategies, that is to say it can be used to forbid only schedules leading to a deadlock while allowing all safe schedules. Furthermore, this method does not need a fully static mapping for tasks as long as the mapping of buffers is fixed and known in advance. This method is also independent of timing information, making it suitable for sets of applications for which the extension to a common period is too expensive, or where tasks can be mapped onto different PEs leading to different execution times (heterogeneous architecture). The purpose of this method is to be as flexible as possible, leaving large possibilities with

regard to the implementation of the scheduler. As such the scheduler could be static or dynamic, is not constrained in its objectives (e.g., maximizing the throughput, minimizing the memory footprint), and is prevented from selecting only the schedules actually leading to a deadlock.

The overall flow of the analysis is illustrated in Figure 4.1. The analysis is divided in three steps:

(i) An automaton representing the scheduling state-space of a dependency graph is computed.

(ii) This automaton is pruned of states overflowing at least one memory.

(iii) The final automaton, representing all possible (but potentially deadlocking) schedules, is analyzed to extract all *safe* non-deadlocking schedules.

The difference between the conventional approach and the contribution of this thesis lies in the replacement of the conventional schedule automaton by a significantly smaller automaton –the control automaton– without any loss in the precision of the analysis. Note that it is possible to merge the first two steps and directly compute an automaton pruned of overflowing states and their inaccessible successors.

The remainder of this chapter is structured as follows: First, Section 4.2 presents design assumptions and the characteristics of the target platforms. Then Section 4.3 presents the conventional schedule automaton used in liveness analysis, and Section 4.4 the modeling of the workload and the constraints on the applications it represent. The liveness analysis itself, and how it can be used to design an exact deadlock prevention strategy, is presented in Section 4.5. An evaluation of the gain in performance when analyzing a control automaton instead of the conventional schedule automaton is conducted in Section 4.6. The mathematical proof justifying the use of control automaton without any loss in generality is presented in Section 4.7. Section 4.8 concludes the chapter.

4.2 System models and design assumptions

A generic instance of the targeted logical **architectures** is shown in Figure 4.2. This architecture is composed of a set of Processing Elements (PE, e.g., CPUs, GPUs, hardware accelerators), local and global memory units connected by a set of buses and interconnects. Direct Memory Access (DMA) engines can be used to transfer

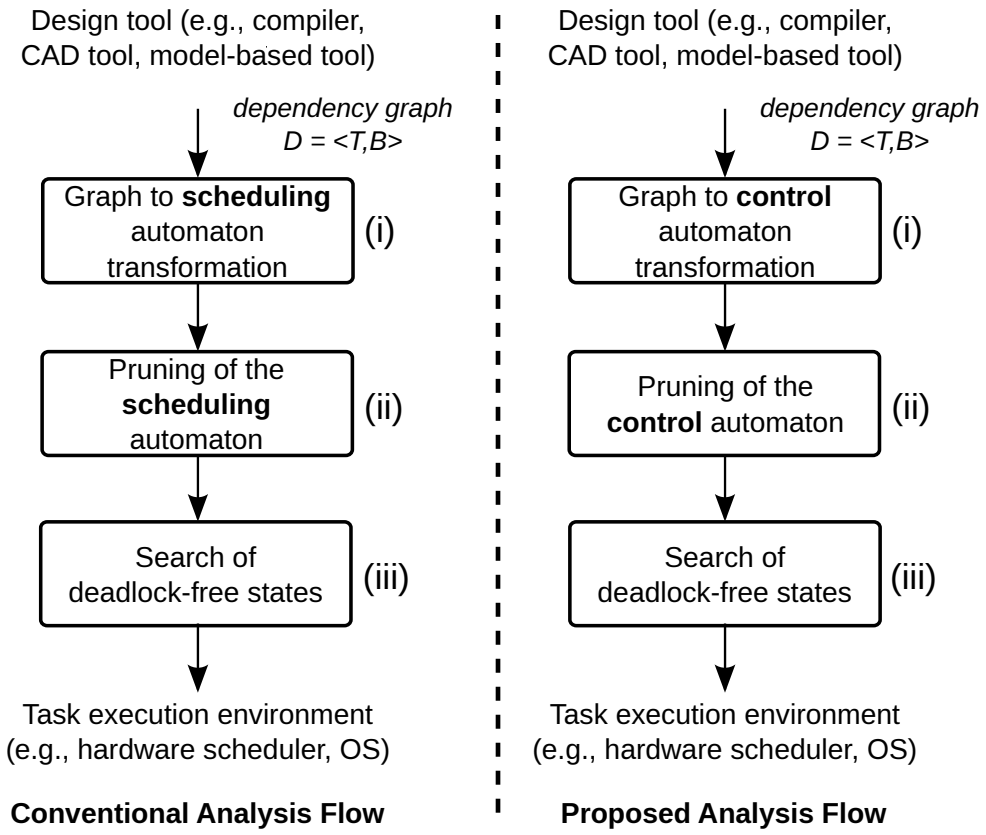


Figure 4.1: The steps of liveness analysis

data between memories. Memories are considered at a logical (abstract) level, regardless their actual implementation (e.g., SRAM, DRAM). Memories are limited in size (fixed size) and shared by tasks at run-time; we do not consider cache memories as, by design, a cache cannot overflow. Caches are thus considered as invisible sub-parts of their host PEs. It should be noted that our contribution is not based on any assumptions on the memories of an architecture, e.g., ports, access restrictions, access times of memories.

The **workload** (i.e., set of applications) running on the target platform is represented by an acyclic dependency graph, $H = (\mathcal{T}, \mathcal{B})$ (with $\mathcal{T}$ set of tasks, $\mathcal{B}$ set of precedence constraints and input/output buffers), as is the case for the first contribution. An example workload can be seen in Figure 3.3 from Chapter 3. Our contribution is *not* based on any hypothesis on the timing characteristics of tasks or communications, like worst-case execution or communication times, deadlines, etc.

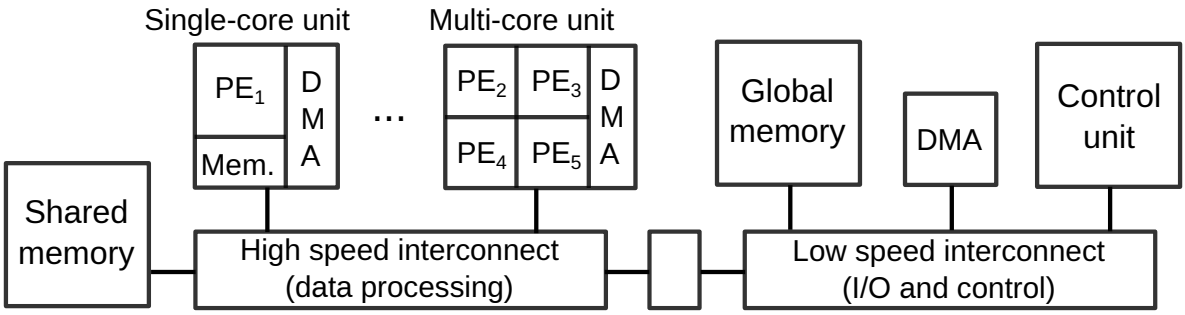


Figure 4.2: Target architecture

This is an important difference in requirements from the first contribution. This timing independence gives more flexibility to the system at run-time, since it allows applications to have independent periods and tasks to have multiple implementations with different timing characteristics depending on the PE executing the task at run-time. If such timing characteristics are available, however, they can be used to further reduce the computational cost of solutions based on our results. One could, for instance, apply one form or another of critical path analysis to compute earliest and latest tasks' start and end times, deduce that some tasks can or cannot be simultaneously active, and further prune the state space of unreachable states.

This contribution is valid for scheduling and memory-management decisions that are taken both on-line and off-line, locally for a specific PE or globally for an entire platform. In comparison to the method presented in the previous chapter, some assumptions are lifted or changed.

All *application* hypotheses (H_4, H_5, and H_6) are lifted: timing information of tasks are not needed, and application periods are no longer required to be identical. This removes the need to extend applications to a common period, thus avoiding to expend the workload graph to analyze.

The *mapping* hypothesis (H_5) is replaced by:

- H_{10}: Buffer mapping is defined statically. Task mapping can be defined dynamically and task migration across PEs is allowed, as long as it is compatible with the static buffer mapping.

Preemption is allowed (H_1 is lifted), and the following *scheduling* assumption is added:

- H_{11}: Task starts or terminations are atomic events that cannot occur simultane-

ously. This is a very common hypothesis in formal automaton analysis where events have no duration and can thus not overlap.

All *memory management* assumptions (H_7, H_8, and H_9) are still effective.

The next section presents the making of conventional pruned scheduling automaton from dependency graphs, that is, steps (i) and (ii) of the *conventional* analysis flow in Figure 4.1, and a property of those automaton that makes possible their reduction to smaller automaton for the purpose of memory usage analysis.

4.3 Schedule Automaton

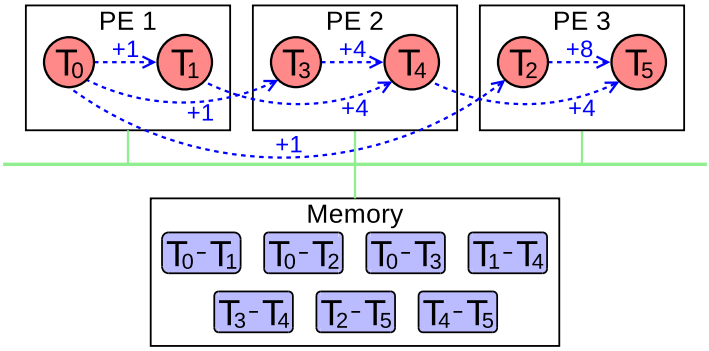


Figure 4.3: Example AHSDF for a single-application, with mapping

For the sake of simplicity, lets consider an example with a single-application, and a target platform composed of a single memory unit of size 15 data units and 3 PEs. In the following all memory sizes are expressed in data units. In Figure 4.3, identifiers denote the mapping of buffers and tasks, and dotted arrows the data dependencies. Note that only the mapping of buffers is required with this analysis. The mapping of tasks does not have to be determined statically and only needs to be compatible with the buffer mapping.

The possible schedules for the application set are represented by a directed acyclic **schedule automaton** $G = (Q, E)$, where Q is the set of possible states for the system, and E the set of possible transition between those states, Figure 4.4. This automaton corresponds to the automaton, that is well-known in the literature [63], and that captures all operations in a RAS. The schedule automaton is conceptually similar to the reachability graph of Petri nets (cf. Figure 2.4), presented in Chapter 2.

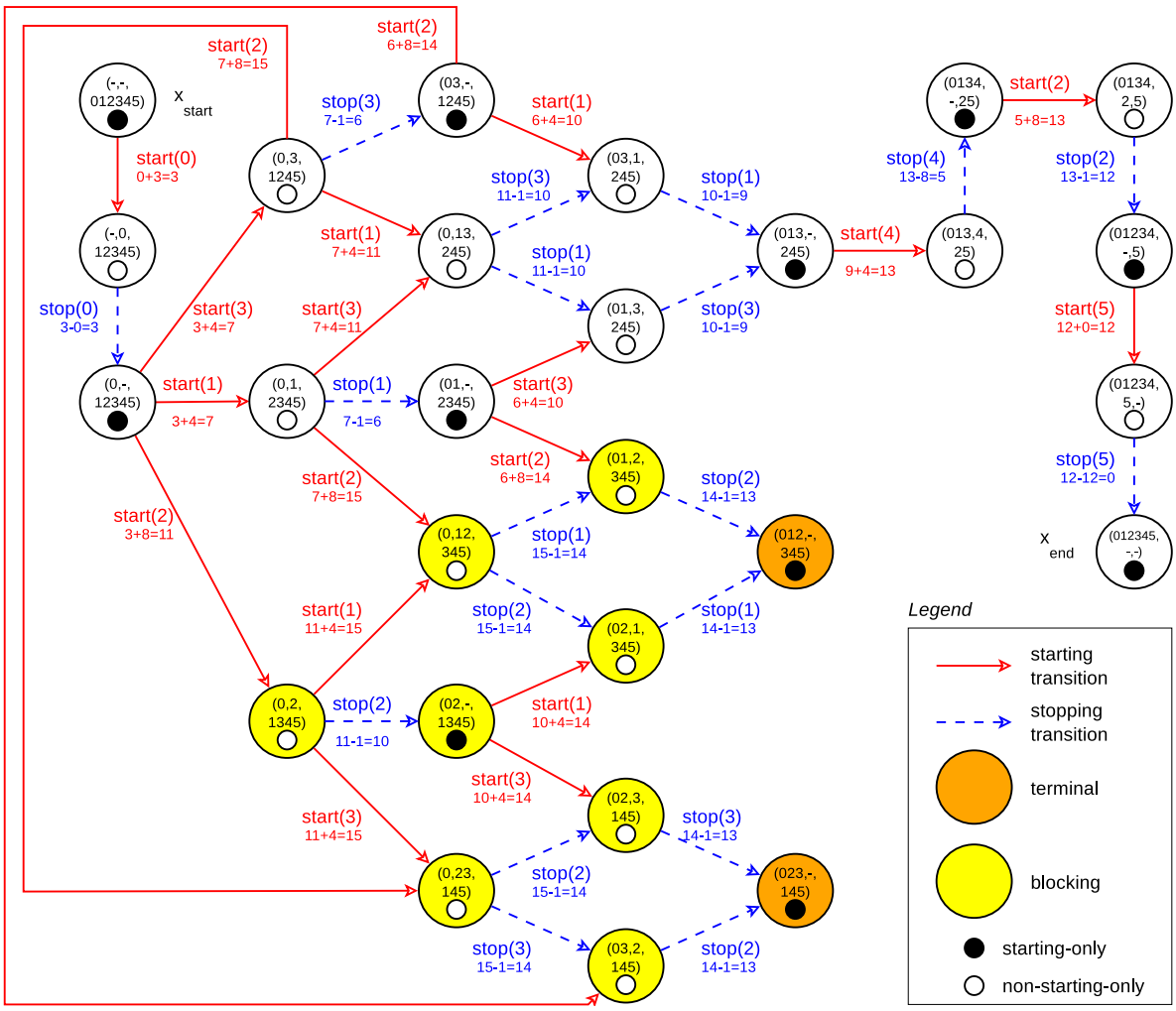


Figure 4.4: Schedule automaton for example in Figure 4.3

Both the schedule automaton and the reachability graph are used to represent the possible states of an underlying system, and in which ways this system can transition from one state to another. Our schedule automaton is a straightforward instance of the generic automaton in [63], where one single resource type (memory) is considered. Furthermore, states that correspond to schedules where at least one memory is overflowed are pruned.

States in Q represent a possible configuration—a step in a schedule—for the system under study. We denote a state x with a tuple (TC_x, TR_x, TF_x). TC_x, TR_x and TF_x are, respectively, the lists of identifiers for the tasks that have already

Completed, are currently **R**unning and will execute in the **F**uture. For sake of legibility tasks in RAS states will be referred to by their indexes only (e.g., the set comprised of T_1 and T_2 will be represented as 12), and empty sets are denoted by $-$. Because of hypothesis H_{11} in Section 4.2, each transition in G represents a state transition where exactly one task either starts or terminates. A label on a transition indicates which task t is started or terminated and the variation of the memory usage in the system. This is expressed by labels in the form $a \pm b = c$ where a is the memory occupancy in the edge start state, $\pm b$ is the memory footprint variation due to the allocation or deallocation of buffers when starting or stopping task t, and c is the memory available in the edge destination state. For example, in Figure 4.4 the transition from state $(0, -, 12345)$ to $(0, 1, 2345)$ has the label `start(1) 3+4=7`. It represents the starting of task T_1, from an initial memory footprint of 3, allocating 4 (since the size of the output buffer of T_1 is 4), and leading to a final memory footprint of 7.

A scheduling automaton is made by representing the possible states of a RAS. In Figure 4.4 we can see a scheduling automaton for the RAS described in Figure 4.3. The initial state x_{start} has all tasks in TF: $(-, -, 012345)$. Initially, only T_0 can be executed, leading to states $(-, 0, 12345)$ and then $(0, -, 12345)$. From here, three tasks have their input requirements satisfied: T_1, T_2, and T_3. This gives the three outgoing transitions and their respective destination states. This process can be pursued until all possible states and transitions have been created. An example state that is impossible, therefore pruned (and not present in Figure 4.4) is $(0, 123, 45)$. This state would require $1 + 1 + 1 + 4 + 4 + 8 = 19 > 15$, thus overflowing the capacity of the memory. For multi-application systems, a schedule automaton is the Cartesian product of the individual application schedule automaton, pruned of vertexes (and their incident transitions) that overflow at least one memory. In target platforms with more than one memory, that is, the general case, the variations of memory occupancy are labeled with memory identifiers:

```
start(1) Mem3(3+4=7) Mem1(0+5=5)...
```

The pseudo-code to create a schedule automaton pruned of overflowing states from an AHSDF graph is given in Algorithms 3, 4 and 5 where $cntEvt$ is a map which keys are task start or end events and which values are the count of other events

that must happen before the key can be selected: if $cntEvt[start(t)] = 5$ there are 5 events that must happen before task t can start. Highlighted lines show the differences with Algorithm 6 that will be presented later. For AHSDF representing multiple concurrent applications (or instances of a same application), the schedule automaton of the system can be created by exploring the full AHSDF graph or, alternatively, by making the Cartesian product of the respective application schedule automaton, pruned of the states overflowing at least one memory unit. The latter approach tends to be more efficient to compute the schedule automaton of the whole system, as it reduces the state space via the pruning of each application schedule automaton and limits redundant computations, especially for application sets where some applications having multiple concurrent iterations.

```
1  input
2  |  H = (T, B);                          // AHSDF of applications
3  |  M;                                   // vector of memory units' sizes
4  output
5  |  G = (Q, E);                          // pruned schedule automaton of H
6  define
7  |  H.predecessors(t);                  // set of immediate predecessor tasks of t in H
8  |  H.successors(t);                    // set of immediate successor tasks of t in H
9  |  H.occupancy(x)[i];                  // occupancy of memory i in state x
10 cntEvt ← emptyMap;                     // initialize map
11 x_start ← (∅, ∅, T);                   // initial state: all tasks to be in executed in future
12 Q ← {x_start};                         // initial state set: start only
13 E ← ∅;                                 // initial edge set: empty
14 foreach t ∈ T do                       // Initialize cntEvt
15 |  cntEvt[start(t)] ← |H.predecessors(t)|;
16 |  cntEvt[end(t)] ← 1;
17 end
18 explore(x_start, cntEvt);              // recursively explore-build automaton graph
```

Algorithm 3: Compute schedule automaton from an AHSDF graph

Before stating our theorem, we introduce the following definitions. In automaton G, we define a state x as *terminal* if it has no successor, e.g., $(012, -, 345)$, orange states in Figure 4.4 and Figure 4.5. A state is *blocking* if there is no path from x to the final state x_{end} (the state representing the completion of all tasks), e.g.,

```
1  Function explore(x, cntEvt) is
2      input
3          x = (tc, tr, tf);                                   // current state
4          cntEvt;                                             // current map
5      foreach (a, n) ∈ cntEvt do
6          if n = 0 then                                       // event can happen
7              y ← getNextState(x, a);                         // x successor by a
8              if ∀i ∈ {1, ..., m}, H.occupancy(y)[i] ≤ M[i] then   // no overflow
9                  newEdge ← (x, a, y);
10                 E ← E ∪ {newEdge};
11                 if y ∉ Q then                               // y not yet explored
12                     Q = Q ∪ {y};
13                     newCntEvt ← cntEvt.copy();
14                     newCntEvt.remove(a);
15                     t ← task(a);
16                     if type(a) = end then
17                         foreach t' ∈ H.successors(t) do
18                             newCntEvt[start(t')] ← newCntEvt[start(t')] − 1;
19                         end
20                     else if type(a) = start then
21                         newCntEvt[end(t)] ← newCntEvt[end(t)] − 1;
22                     explore(y, newCntEvt);
23                 end
24             end
25         end
26     end
27 end
```

Algorithm 4: explore function for the *schedule* automaton

```
1  Function getNextState(x, a) is
2      input
3          x = (tc, tr, tf);                          // source state
4          a;                                         // action from source state
5      output
6          y;                                         // x successor state by action a
7      t ← task(a);
8      if type(a) = start then
9          y ← (tc, tr ∪ {t}, tf \ {t});
10     else if type(a) = end then
11         y ← (tc ∪ {t}, tr \ {t}, tf);
12     return y
13 end
```

Algorithm 5: $\mathrm{getNextState}$ function

$(0, 2, 1345)$, yellow states in Figure 4.4 and Figure 4.5. A *starting* transition is one that corresponds to a task starting execution and the allocation of its output buffers. Conversely, a *stopping* transition is one where a task terminates and some of its input buffers may be deallocated if they are not needed anymore. A state is *starting-only* if all its outgoing transitions are *starting* transitions (or it is a terminal without outgoing transitions); it is *non-starting-only* if at least one of its outgoing transitions is a *stopping* transition.

A relevant property of a *non-starting-only* state $x = (TC_x, TR_x, TF_x)$ is that it has a unique successor *starting-only* state $y = (TC_x \cup TR_x, -, TF_x)$ that is reachable by at least one path composed exclusively of stopping transitions. We name this state y the *control* state of x. It only differs from x in that all tasks running in x are completed in y. The y state is reachable from x because as $x \in Q$ it does not overflow any memory and a stopping transition never increases the occupancy of memories; it can only decrease it if some input buffers of the stopped task are not needed anymore. A more formal proof of this lemma will be given in Section 4.7. Example: in Figure 4.4, state $(023, -, 145)$ is the *control* state of states $(0, 23, 145)$, $(02, 3, 145)$, and $(03, 2, 145)$.

Theorem 1. *Let $G = (Q, E)$ be a schedule automaton, let x be a* non-starting-only *state and let y its control state. x is blocking if and only if y is blocking.*

Proof. Consider there exists a path π from a *non-starting-only* state x to the final state in G, $\pi = x, \ldots, x_{end}$, there is also an equivalent path π' from the corresponding *control* state y to the final state in G, $\pi' = y, \ldots, x_{end}$. Along path π' the same tasks as in π are started or stopped, in the same order, except for the tasks that are currently running in x and already completed in y. For these tasks there is a stopping transition in π but no transition in π'. At each state in π' the memory consumption of any memory is less than or equal to that in the equivalent state in π. Hence, if there exists a path from x to x_{end} there exists also a path from y to x_{end}. Conversely, if y is a blocking state, x must be blocking too. And as y is a reachable successor of x, if y is non-blocking, x is non-blocking too. ■

The formal statement of the theorem and its proof are deferred to Section 4.7. **This is a powerful theorem: it implies that studying only a limited set of states is sufficient to detect deadlocks.** In the next section this theorem is used to develop a new type of automaton that is equivalent for memory usage analysis, but smaller with regard to the number of states and transitions and thus faster to analyze. This corresponds to steps (i) and (ii) of the *proposed* analysis flow in Figure 4.1.

4.4 Control Automaton

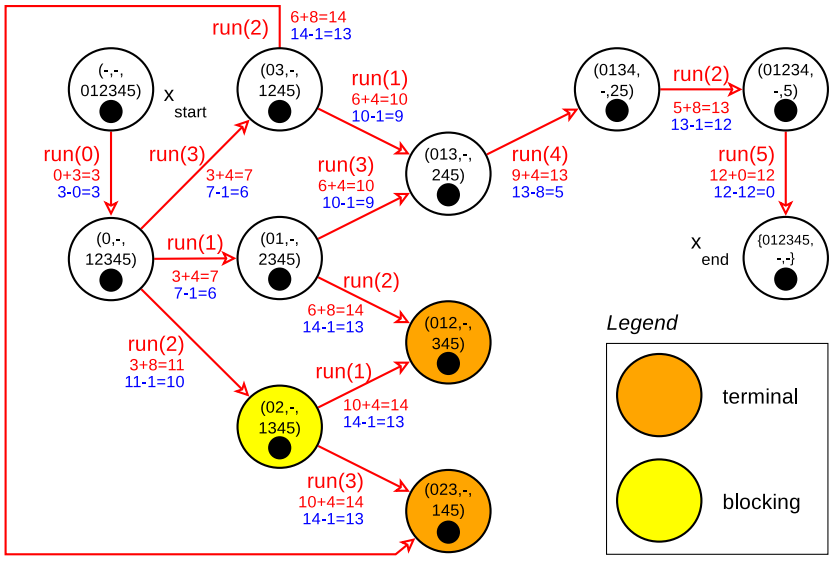


Figure 4.5: Control automaton for Figure 4.3

Figure 4.5 shows an automaton that we call **control automaton**, which is guaranteed by Theorem 1 to preserve the properties of its corresponding schedule automaton, Figure 4.4. All nodes in a control automaton are *starting-only* states: states for which all outgoing transitions, if there are any, are task starting transitions. Note that some possible states, corresponding to interleavings of different tasks (e.g., $(0, 13, 245)$ for Figure 4.4) are *not* present in the control automaton as they are *non-starting-only* states and Theorem 1 guarantees that studying *starting-only* states only is sufficient.

A control automaton can theoretically be obtained from the schedule automaton, but it is more efficient to slightly change Algorithms 3, 4 and 5 to directly produce the control automaton without constructing the schedule automaton. The pseudo-code to create a control automaton is shown in Algorithms 6, 7 and 8, with highlighted lines illustrating the difference. These differences remove all the *non-starting-only* states and all transitions corresponding to the end of a task. More precisely, when the execution of a task t starts, the following state is obtained by moving said task from the set of future task to the set of completed tasks, instead of the set of running tasks, as *starting-only* states always have an empty set of running tasks (since any task in that set would lead to an outgoing task end transition in the automaton). This can be seen for example with state $(03, -, 1245)$ being followed by the states $(03, 1, 245)$ and $(03, 2, 145)$ in Figure 4.4, but by the states $(013, -, 245)$ and $(023, -, 145)$ instead in Figure 4.5.

In the next section, step (iii) of the analysis flow of Figure 4.1 is presented for the control automaton. It can easily be adapted to the conventional schedule automaton since the search of deadlock-free states in the *conventional* and *proposed* analysis flows follow the same principle. We also propose a possible representation of the results, to be used in an execution environment.

4.5 Liveness Analysis of Automaton

In this section we informally describe the liveness analysis, based on a concrete example. Our contribution rests on Theorem 1 presented in Section 4.3 of this chapter, and fully formalized in Section 4.7. This theorem allows us to precisely detect deadlocks without the need to explore the complete state space of all schedules for

```
input
    H = (T, B);                                   // AHSDF of applications
    M;                                            // vector of memory units' sizes
output
    G = (Q, E);                                   // pruned schedule automaton of H
define
    H.predecessors(t);                            // set of immediate predecessor tasks of t in H
    H.successors(t);                              // set of immediate successor tasks of t in H
    H.occupancy(x)[i];                            // occupancy of memory i in state x
cntEvt ← emptyMap;                                // initialize map
x_start ← (∅, ∅, T);                              // initial state: all tasks to be in executed in future
Q ← {x_start};                                    // initial state set: start only
E ← ∅;                                            // initial edge set: empty
foreach t ∈ T do                                  // Initialize cntEvt
    cntEvt[start(t)] ← |H.predecessors(t)|;
end
explore(x_start, cntEvt);                         // recursively explore-build automaton graph
```

Algorithm 6: Compute control automaton from an AHSDF graph

a workload graph.

A possible representation of the results obtained from the liveness analysis are the **anti-deadlock rules**. They are obtained from an automaton by searching for transitions from a non-blocking state to a blocking state. In order to decide which states are blocking or non-blocking, a reverse traversal of the automaton is performed starting from the final state x_{end} up to the initial state x_{start}. All states reachable from x_{end} are non-blocking, and the remaining ones are blocking. These rules can be used to ensure the deadlock-freedom of a system, e.g., by compilers, CAD tools. In Figure 4.5, we can produce three rules, forbidding the execution of task T_2 if the set of future tasks TF is 12345, 2345, or 1245. More generally, anti-deadlock rules can be formalized as $r = (\mathcal{B}_r, \mathcal{T}_r)$ where $\mathcal{B}_r$ represents a set of buffers simultaneously allocated in memory (on the same or on different units), and $\mathcal{T}_r$ the set of tasks whose execution would eventually lead to a deadlock and therefore must **not** be executed when all buffers in $\mathcal{B}_r$ are allocated. Any scheduler that follows the set of rules produced by the liveness analysis is guaranteed to avoid the deadlocks caused by memory shortage.

```
1  Function explore(x, cntEvt) is
2      input
3          x = (tc, tr, tf);                                   // current state
4          cntEvt;                                             // current map
5      foreach (a, n) ∈ cntEvt do
6          if n = 0 then                                       // event can happen
7              y ← getNextStartingOnlyState(x, a);             // x successor by a
8              if ∀i ∈ {1, ..., m}, H.occupancy(y)[i] ≤ M[i] then   // no overflow
9                  newEdge ← (x, a, y);
10                 E ← E ∪ {newEdge};
11                 if y ∉ Q then                               // y not yet explored
12                     Q = Q ∪ {y};
13                     newCntEvt ← cntEvt.copy();
14                     newCntEvt.remove(a);
15                     t ← task(a);
16                     foreach t' ∈ H.successors(t) do
17                         newCntEvt[start(t')] ← newCntEvt[start(t')] − 1;
18                     end
19                     explore(y, newCntEvt);
20                 end
21             end
22         end
23     end
24 end
```

Algorithm 7: explore function for the *control* automaton

```
1  Function getNextStartingOnlyState(x, a) is
2      input
3          x = (tc, ∅, tf);                                    // source state
4          a;                                                  // action from source state
5      output
6          y;                                                  // x successor starting-only state by action a
7      t ← task(a);
8      y ← (tc ∪ {t}, ∅, tf \ {t});
9      return y
10 end
```

Algorithm 8: getNextStartingOnlyState function

4.5.1 Reduction of computational costs

Thanks to Theorem 1, we obtain a considerable gain in terms of the number of states to inspect in a control automaton, as opposed to the states to explore in a schedule automaton. This gain depends on the number of tasks and on the parallelism in a workload graph. It can be expressed in closed form only for linear workload graphs, with no parallelism. In this case, the schedule automaton of a workload with n tasks is composed of $2n + 1$ states. Only $n + 1$, out of $2n + 1$ are the states that form the corresponding control automaton. This yields a reduction factor of about 2. In general, for a *system* composed of k instances of similar linear workloads, the reduction factor amounts to approximately 2^k.

The presence of parallelism in a workload graph increases the gain (reduction factor). This significantly favors the scalability of analysis based on Theorem 1. This reduction cannot be computed in closed form, however, it can be appreciated by considering a few examples. In the workload of Figure 4.3, where the maximum degree of parallelism is 3 (3 tasks, namely tasks 1, 2 and 3, at most can execute in parallel at any time), the number of states to analyze is reduced from 27, in Figure 4.4, to 11, in Figure 4.5. For systems of applications, this reduction is even more important. In a system composed of 10 instances of the parallel workload in Figure 4.3, our theorem yields a reduction of the number of states by a factor of $(27/11)^{10} \approx 8000$. Instead, for a system with 10 *linear* workloads, our theorem yields a smaller reduction, equal to $2^{10} = 1024$.

4.6 Evaluation of Schedule and Control Automaton Analysis

In this section, we discuss first the evaluation of the effectiveness of searching for deadlocks in the control automaton as opposed to searches in a schedule automaton. Our evaluation is summarized in Table 4.1 for 3 types of workloads, where, for each type, we report the average and worst-case results of 100 experiences. Then we compare our exact approach with an existing heuristic to illustrate that exact approaches are much more permissive than heuristics with regard to the number of possible schedules.

The aim of our evaluation is to provide statistically significant results that show the limits of traditional full state-space analysis and prove that our contribution pushes these limits further away, enabling the study of more complex systems. We created a test-bench by generating random application graphs using SDF^3 [4]. In our test-bench, each actor has a variable number of successors, between 2 and 5. The maximum degree of parallelism is 4 tasks. The average size of buffers is 2,048 *du*. In order to compare our work with the standard maximally permissive liveness analysis we had to guarantee that the latter would not overflow the RAM of the PC where we ran the experiments. For instance, for a workload with 6 applications, the maximum number of tasks per application we study here is 6, as for any larger number of tasks the complete schedule automaton would overflow the PC RAM, even if our approach allows to handle much larger problems. The evaluation was conducted on a personal computer with 1 CPU core @ 3.50GHz, 32 GiB of RAM. The analysis running times are not polluted by swapping of memory pages by the Operating System (OS): we explicitly forbade swapping in our configuration.

Workloads are scheduled on the same target architecture, with 3 PEs and 3 memories, each with a capacity of 4 Gibibyte (1 GiB is $2^{30} = 1,073,741,824$ bytes) (GiB). This large capacity was selected because it is paradoxically a worst case scenario for our deadlock analysis: any memory can host all the buffers of the workloads, thus the execution of a workload never incurs into deadlocks and the complete exploration of the graphs is necessary. Tasks in a workload were randomly mapped to PEs and their input/output buffers were also randomly assigned to memories. We allowed buffers of a given task to be mapped to physically different memories.

In Table 4.1, we can see that, thanks to our theorem, we reduce by at least two orders of magnitude the number of states in the automaton to analyze, and therefore of the computation time of the liveness analysis. The complexity of our approach is still exponential. However, this reduction enables maximally permissive deadlock analysis on more complex systems for which the cost of analysis was prohibitive. With a 32 GiB RAM PC, we reach the limit of the control automaton that we can study for cases with approximately 150M states. This corresponds to 8 applications for the first workload type (top-most), 7 applications for the second and 6 applications for the last (bottom-most). It is possible, for instance, to analyze a configuration with 6 applications of 10 tasks each, with a control automaton of size 149M, while the

analysis of the complete automaton is out of reach with a size of 268×10^9 states.

	Number of states		**Analysis time**	
6 apps. of 6 tasks	Average	Maximal	Average	Maximal
Schedule automaton	63M	139M	12 min	28 min
Control automaton	458k	1.2M	5.1 s	14 s
5 apps. of 8 tasks	Average	Maximal	Average	Maximal
Schedule automaton	59M	148M	10 min	26 min
Control automaton	450k	834k	4.3 s	8.2 s
4 apps. of 10 tasks	Average	Maximal	Average	Maximal
Schedule automaton	36M	149M	5 min	24 min
Control automaton	273k	646k	2.3 s	5.6 s

Table 4.1: Evaluation results for a test-bench of 3 workload types

A comparison of the exact deadlock handling strategy derived from the liveness analysis presented in Section 4.5 of this chapter with the deadlock prevention presented in Chapter 3, and with a state-of-the-art tool is deferred to Chapter 5.

4.7 Mathematical Formalization

4.7.1 Definitions

We define here all the mathematical objects that are used in the formal proof of theorem 1. All given examples are from Figure 4.3 or its derived schedule automaton shown in Figure 4.4. We first prove theorem 1 for a system with only one memory unit. The generalization to multi-units systems is trivial.

- $v \in \mathbb{N}$ is the size of memory in a target platform.
  Example: $v = 15$.
- T the set of tasks of all applications in a workload graph.
  Example: $T = \{T_0, T_1, T_2, T_3, T_4, T_5\}$.
- B the set of input and output buffers in a workload graph.
  Example: $B = \{T_0 - T_1, T_0 - T_2, T_0 - T_3, T_1 - T_4, T_3 - T_4, T_2 - T_5, T_4 - T_5\}$.

- $\forall t \in T, I_t \subset B$ is the set of input buffers of task t.
  Example: For $t = T_4, I_t = \{T_1 - T_4, T_3 - T_4\}$.

- $\forall t \in T, O_t \subset B$ is the set of output buffers of task t.
  Example: For $t = T_4, O_t = \{T_4 - T_5\}$.

- $\forall b \in B, v_b \in \mathbb{N}$ is the size of buffer b.
  Example: For $b = T_4 - T_5, v_b = 4$.

- $\forall b \in B, p_b \in T$ is the producer task of buffer b.
  Example: For $b = T_4 - T_5, p_b = T_4$.

- $\forall b \in B, C_b \subset T$ is the set of consumer tasks of buffer b.
  Example: For $b = T_4 - T_5, C_b = \{T_5\}$.

- Q is the set of states and also the vertexes of G.

- G has one unique final state s_{end}, where all tasks are completed and all buffers are deallocated.

- Each state $x \in Q$ is uniquely identified by the (TC_x, TR_x, TF_x) tuple where TC_x, TR_x, TF_x are respectively the sets of already completed, currently running and future tasks.

- $E \subset Q^2$ is the set of transitions and also the edges of G.

- $\forall (x, y) \in E$, y is a *child* of x and x is a *parent* of y.
  Example: $(0, 13, 245)$ is a child of $(0, 1, 2345)$, and $(0, 1, 2345)$ is a parent of $(0, 13, 245)$.

- $S = \{\langle x_1, \ldots, x_n \rangle \mid \forall 1 \leq j < n, (x_j, x_{j+1}) \in E\}$ is the set of traces in G.

- $\forall \langle x, \ldots, y \rangle \in S$, y is a *descendant* of x and x is an *ancestor* of y. We denote this $x \to y$.
  Example: $(013, 4, 25)$ is a descendant of $(0, 1, 2345)$, and $(0, 1, 2345)$ is a ancestor of $(013, 4, 25)$.

- $\forall x \in Q, v_x$ is the memory occupancy in state x.
  Example: For $x = (0, 13, 245)$, $v_x = 11$.

- $\forall x \in Q, TI_x \subset T$ is the set of tasks which input buffers are available in state x: $TI_x = \{t \in T \mid \forall b \in I_t, p_b \in TC_x\}$.
  Example: For $x = (013, -, 235), TI_x = \{T_2, T_4\}$.

- $\forall x \in Q, TO_x \subset T$ is the set of tasks whose output buffers can be allocated without memory overflow in state x: $TO_x = \{t \in T, v_x + (\sum_{b \in O_t} v_b) \leq v\}$.
  Example: For $x = (013, -, 235), TO_x = \{T_4, T_5\}$.

- $\forall x \in Q, T_x \subset T$ is the set of *runnable* tasks when in state x: $T_x = TF_x \cap TI_x \cap TO_x$
  Example: For $x = (013, -, 235), TF_x = \{T_4\}$.

- $E^- \subset E$ is the set of task-*stopping* transitions.

- $E^+ \subset E$ is the set of task-*starting* transitions.

- $S^- \subset S$ is the set of traces with only stopping transitions.

- $\forall e \in E, t_e \in T$ is the task that terminates (if $e \in E^-$) or starts (if $e \in E^+$).

- $Q^| = \{x \in Q \backslash \{s_{end}\} \mid \nexists (x, y) \in E\}$ is the set of *terminal* states, that is, states which are not the final state and without any successor state.
  Example: $Q^| = \{(012, -, 345), (023, -, 145)\}$.

- $Q^\dashv = \{x \in Q \mid \nexists \sigma \in S, \sigma = \langle x, \ldots, s_{end} \rangle\}$ is the set of *blocking* states, that is, states from which there is no trace to the final state. As G is a directed acyclic graph, blocking states are also states from which all execution traces encounter a terminal state.
  Example: All yellow or orange states in Figure 4.4.

Note that **terminal and blocking states are responsible for deadlocks**. The purpose of our analysis is to reduce the computational cost of their identification.

- $Q^+ = \{x \in Q \mid \forall y \in Q, (x, y) \in E \Rightarrow (x, y) \in E+\}$ is the set of *starting-only* states, that is, states which all outgoing transitions, if there are any, are task starting transitions. The *starting-only* states are all states $s = (TC_s, \emptyset, TF_s)$ where no tasks are currently running (and cannot thus stop). Note that, by definition, terminal and final states are also considered as task *starting-only*

states because they have no outgoing transitions: $Q^{|} \subset Q^{+}, s_{end} \in Q^{+}$.
Example: All states with a black dot in Figure 4.4.

- $Q^{\sim} = Q \backslash Q^{+}$, complement of Q^{+} in Q, is the set of *non-starting-only* states, that is, states with at least one *stopping* outgoing transition. The *non-starting-only* states are all states $s = (TC_s, TR_s \neq \emptyset, TF_s)$ where tasks are currently running (and can thus stop).
  Example: All states with a white dot in Figure 4.4.

- $\forall (x, y) \in Q^2, y$ is a *subordinate* of x if the only difference between x and y is that some tasks running in x are completed in y. We denote this $x \searrow y$ and we denote $\tau_{x,y}$ the set of tasks running in x that are completed in y:
  $x \searrow y \Leftrightarrow \exists \tau_{x,y} \subset T, \tau_{x,y} \neq \emptyset, TR_x = TR_y \cup \tau_{x,y} \wedge TC_y = TC_x \cup \tau_{x,y} \wedge TF_y = TF_x$
  Example: $y = (01, 3, 245)$ is a subordinate of $x = (0, 13, 245)$ with $\tau_{x,y} = \{T_1\}$.

4.7.2 Properties

- A stopping transition does not allocate any memory buffer. It can only free some memory buffers if they are not needed anymore. So, if there is a stopping transition (x, y), the memory occupancy in y is less or equal than in x:

$$\forall e = (x, y) \in E^{-}, v_y \leq v_x \tag{MD1}$$

- There is no memory overflow:

$$\forall x \in Q, 0 \leq v_x \leq v \tag{NMO}$$

- Stopping running tasks is always possible because it never increases the memory consumption. So, if tasks are running in x there is always a stopping transition to a child state y:

$$\forall x \in Q, TR_x \neq \emptyset \Rightarrow \exists e = (x, y) \in E^{-}, t_e \in TR_x \tag{STP}$$

- MD1 generalization: if a trace of stopping transitions exists from x to y then the memory occupancy in y is less or equal than in x:

$$\forall(x,y) \in Q^2, \exists\sigma = \langle x, \ldots, y\rangle \in S^- \overset{MD1}{\Rightarrow} v_y \leq v_x \tag{MD2}$$

- Equivalently, if y is a subordinate of x then the memory occupancy in y of is less or equal than in x:

$$\forall(x,y) \in Q^2, x \searrow y \overset{def}{\Rightarrow} \exists\sigma = \langle x, \ldots, y\rangle \in S^- \overset{MD2}{\Rightarrow} \quad v_y \leq v_x \tag{MD3}$$

The proof of our theorem relies on the following lemmas, concerning the relation between the states of a schedule graph and the corresponding states in a control graph.

Lemma 1. *For any* non-starting-only *state* $x \in Q$*, there exists a unique* starting-only *state* $y \in Q$ *which is also a subordinate of* x*:*

$$\forall x \in Q^{\sim}, \exists! y \in Q^{+} | x \searrow y$$

Proof. If x is a *non-starting-only* state, it has at least one running task t and by STP there is a child state x_t of x in which task t stopped and the transition between x and x_t is a stopping transition:

$$x \in Q^{\sim} \Rightarrow \exists t \in TR_x, \exists(x, x_t) \in E^-, TC_{x_t} = TC_x \cup \{t\}, TR_{x_t} = TR_x \backslash \{t\}, TF_{x_t} = TF_x$$

Recursively, from x_t, if it is not itself a *starting-only* state, there is a child state in which one more task stopped, and so on, until we reach the first *starting-only* state, y, where all running tasks of TR_x stopped. All transitions from x to y are stopping transitions, so y is a subordinate of x. y is unique because it is uniquely defined by $(TC_y, TR_y, TF_y) = (TC_x \cup TR_x, \emptyset, TF_x)$. ■

We name this unique y the *control* state of x and we denote it $y = \delta(x)$.

Lemma 2. *If y is a subordinate of x the set of runnable tasks in x is a subset of the set of runnable tasks in y:*

$$\forall (x,y) \in Q^2, x \searrow y \Rightarrow T_x \subset T_y$$

Proof.

$$\forall t \in T_x \overset{def}{\Rightarrow} t \in TF_x \overset{def}{=}{}^{\searrow} TF_y \tag{4.1}$$

$$\forall t \in T_x \overset{def}{\Rightarrow} t \in TI_x \overset{def}{\Rightarrow} \forall b \in I_t, p_b \in TC_x \overset{def}{\subset}{}^{\searrow} TC_y \overset{def}{\Rightarrow} t \in TI_y \tag{4.2}$$

$$\begin{aligned} \forall t \in T_x &\overset{def}{\Rightarrow} t \in TO_x \overset{def}{\Rightarrow} v_x + \left(\sum_{b \in O_t} v_b \right) \leq v \\ &\overset{MD3}{\Rightarrow} v_y + \left(\sum_{b \in O_t} v_b \right) \leq v \overset{def}{\Rightarrow} t \in TO_y \end{aligned} \tag{4.3}$$

$$4.1 \wedge 4.2 \wedge 4.3 \overset{def}{\Rightarrow} \forall t \in T_x, t \in T_y \Leftrightarrow Tx \subset T_y$$

■

Lemma 3. *Let $x \in Q$ be a state, let y be a subordinate of x, let x' be a child of x with transition $e = (x, x') \in E^-$ terminating a task t_e running in x but not in y. Then y is a subordinate of x' with $x' = y$ or $\tau_{x',y} = \tau_{x,y} \backslash \{t_e\}$:*

$$\begin{aligned} \forall (x, x', y) \in Q^3, &x \searrow y \wedge e = (x, x') \in E^- \wedge t_e \in \tau_{x,y} \\ &\Rightarrow (x' = y) \vee (x' \searrow y \wedge \tau_{x',y} = \tau_{x,y} \backslash \{t_e\}) \end{aligned}$$

Proof. Let $\tau_{x',y} = \tau_{x,y} \backslash \{t_e\}$. We have:

$$\begin{aligned} TR_{x'} &\overset{def}{=} TR_x \backslash \{t_e\} \overset{def}{=}{}^{\searrow} (TR_y \cup \tau_{x,y}) \backslash \{t_e\} \overset{t_e \notin TR_y}{=} TR_y \cup (\tau_{x,y} \backslash \{t_e\}) \\ &\overset{def}{=} TR_y \cup \tau_{x',y} \end{aligned} \tag{4.4}$$

$$TC_y \overset{def}{=} TC_x \cup \tau_{x,y} \overset{t_e \in \tau_{x,y}}{=} (TC_x \cup \{t_e\}) \cup (\tau_{x,y} \backslash \{t_e\}) \overset{def}{=} TC_{x'} \cup \tau_{x',y} \tag{4.5}$$

$$TF_y \overset{def}{=} TF_x \overset{def}{=} TF_{x'} \tag{4.6}$$

$$\tau_{x',y} = \emptyset \overset{4.4,4.5,4.6}{\Rightarrow} TR_{x'} = TR_y \wedge TC_{x'} = TC_y \wedge TF_{x'} = TF_y \overset{def}{\Rightarrow} x' = y$$
$$\tau_{x',y} \neq \emptyset \overset{4.4,4.5,4.6}{\Rightarrow} x' \searrow y \wedge \tau_{x',y} = \tau_{x,y} \backslash \{t_e\}$$

■

Lemma 4. *Let $x \in Q$ be a state, let y be a subordinate of x and let x' be a child of x. Then, if $e = (x, x') \in E^-$ terminates $t_e \notin \tau_{x,y}$, there is a transition $(y, y') \in E^-$ that also terminates t_e and such that y' is a subordinate of x' with $\tau_{x',y'} = \tau_{x,y}$.*

Proof.

$$e \in E^- \overset{def}{\Rightarrow} t_e \in TR_x$$
$$t_e \notin \tau_{x,y} \overset{def}{\Rightarrow} t_e \in TR_y \overset{def}{\Rightarrow} \exists e' = (y, y') \in E^-, t_{e'} = t_e$$

Let $\tau_{x',y'} = \tau_{x,y}$. Then, we have:

$$\begin{aligned} TR_{x'} &\overset{def}{=} TR_x \backslash \{t_e\} \overset{def\searrow}{=} (TR_y \cup \tau_{x,y}) \backslash \{t_e\} \\ &= (TR_y \backslash \{t_e\}) \cup \tau_{x,y} \overset{def}{=} TR_{y'} \cup \tau_{x',y'} \qquad (4.7) \\ TC_{y'} &\overset{def}{=} TC_y \cup \{t_e\} \overset{def}{=} TC_x \cup \tau_{x,y} \cup \{t_e\} \\ &= (TC_x \cup \{t_e\}) \cup \tau_{x,y} \overset{def}{=} TC_{x'} \cup \tau_{x',y'} \qquad (4.8) \\ TF_{y'} &\overset{def}{=} TF_y \overset{def}{=} TF_x \overset{def}{=} TF_{x'} \qquad (4.9) \end{aligned}$$

$$4.7 \wedge 4.8 \wedge 4.9 \overset{def}{\Rightarrow} x' \searrow y'$$

■

Lemma 5. *Let $x \in Q$ be a state, let y be a subordinate of x and let x' be a child of x. Then, if $e = (x, x') \in E^+$ starts t_e, there is transition $(y, y') \in E^+$ that also starts t_e and such that y' is a subordinate of x' with $\tau_{x',y'} = \tau_{x,y}$.*

Proof.

$$e \in E^+ \overset{def}{\Rightarrow} t_e \in T_x \overset{lemma\ 2}{\Rightarrow} t_e \in T_y \overset{def}{\Rightarrow} \exists e' = (y, y') \in E^+, t_{e'} = t_e$$

Let $\tau_{x',y'} = \tau_{x,y}$. Then, we have:

$$TR_{x'} \overset{def}{=} TR_x \cup \{t_e\} \overset{def}{=}{}^{\searrow} (TR_y \cup \tau_{x,y}) \cup \{t_e\}$$
$$= (TR_y \cup \{t_e\}) \cup \tau_{x,y} \overset{def}{=} TR_{y'} \cup \tau_{x',y'} \tag{4.10}$$
$$TC_{y'} \overset{def}{=} TC_y \overset{def}{=} TC_x \cup \tau_{x,y} = TC_{x'} \cup \tau_{x',y'} \tag{4.11}$$
$$TF_{y'} \overset{def}{=} TF_y \backslash t_e \overset{def}{=} TF_x \backslash t_e \overset{def}{=} TF_{x'} \tag{4.12}$$

$$4.10 \wedge 4.11 \wedge 4.12 \overset{def}{\Rightarrow} x' \searrow y'$$

■

Below is the formal definition and proof of theorem 1 that we informally stated in Section 4.3.

Theorem 1. *Let $G = (Q, E)$ be a schedule automaton, let x be a* non-starting-only *state and let y its control state. x is blocking if and only if y is blocking.*

Proof. Using our definitions the theorem can be re-formulated as: a *non-starting-only* state x is blocking if and only if its control state $y = \delta(x)$ is also blocking:

$$\forall x \in Q^{\sim}, x \in Q^{\dashv} \Leftrightarrow \delta(x) \in Q^{\dashv}$$

$x \in Q^{\dashv} \Rightarrow \delta(x) \in Q^{\dashv}$ is trivial: as $x \to \delta(x)$, if $\delta(x)$ was non-blocking, there would be a trace from x to s_{end} through $\delta(x)$ and x would also be non-blocking.
Let us assume that $y = \delta(x)$ is blocking but not x. There must thus be a trace $\sigma_x = \langle x = x_1, x_2, \ldots, x_n = s_{end} \rangle$ from x to s_{end}. Based on this we will prove that a trace $\sigma_y = \langle y = y_1, y_2, \ldots, s_{end} \rangle$ from y to s_{end} also exists, which contradicts $y = \delta(x) \in Q^{\dashv}$. We construct σ_y by induction. We first build a sequence (not a trace) of states $\sigma = \langle y, y_2, \ldots, y_n = s_{end} \rangle$ where two consecutive states are either the same state or a parent-child pair. σ is built such that:

- $y = \delta(x) \Rightarrow x \searrow y \wedge \tau_{x,y} = TR_x$
- $\forall 2 \leq i \leq n, x_i \searrow y_i \wedge \tau_{x_i,y_i} \subset \tau_{x,y} = TR_x$

For each transition $e = (x_i, x_{i+1})$:

- If $t_e \in \tau_{x_i,y_i}$ (e is a stopping transition, the stopped task is running in x), we add $y_{i+1} = y_i$ to σ.
- Else, we extend σ with a y_{i+1} such that transition (y_i, y_{i+1}) starts or terminates the same task t_e and allocates or deallocates (if they are still allocated) the same buffers as e.

Induction hypothesis: $H_i : x_i \searrow y_i \wedge \tau_{x_i,y_i} \subset TR_x$. H_1 obviously holds by definition of control states. Let us assume now that H_i holds for some $1 < i < n$ and let us construct y_{i+1} according the already presented construction scheme:

Case #1: $e = (x_i, x_{i+1}) \in E^- \wedge t_e \in \tau_{x_i,y_i}$.
We add $y_{i+1} = y_i$ to σ and define $\tau_{x_{i+1},y_{i+1}} = \tau_{x_i,y_i} \backslash t_e$. Let us prove H_{i+1}: y_i is a subordinate of x_i by H_i. x_{i+1} is a child of x_i such that $e = (x_i, x_{i+1})$ terminates $t_e \in \tau_{x_i,y_i}$. So lemma 3 applies:

- If $y_{i+1} \neq x_{i+1}$ then, $y_{i+1} = y_i$ is a subordinate of x_{i+1} and $\tau_{x_{i+1},y_{i+1}} \subset \tau_{x_i,y_i} \subset TR_x$. So H_{i+1} holds.
- If $y_{i+1} = x_{i+1}$ the sequence σ joins σ_x, we extend σ with the tail of σ_x and we stop the induction:

  $\sigma = \langle y, y_2, \ldots, y_{i+1} = x_{i+1}, x_{i+2}, \ldots, x_n = s_{end} \rangle$

Case #2: $e = (x_i, x_{i+1}) \in E^- \wedge t_e \notin \tau_{x_i,y_i}$.
lemma 4 applies and there is a transition $(y_i, y_{i+1}) \in E^-$ that also terminates t_e and such that y_{i+1} is a subordinate of x_{i+1}. $\tau_{x_{i+1},y_{i+1}} = \tau_{x_i,y_i} \subset TR_x$. So H_{i+1} holds.

Case #3: $e = (x_i, x_{i+1}) \in E^+$.
lemma 5 applies and there is a transition $(y_i, y_{i+1}) \in E^+$ that also starts t_e and such that y_{i+1} is a subordinate of x_{i+1}. $\tau_{x_{i+1},y_{i+1}} = \tau_{x_i,y_i} \subset TR_x$. So H_{i+1} holds.

Conclusion: Assume $\nexists 1, \leq i \leq n, y_i = x_i$. We proved that H_1 holds and that if H_i holds H_{i+1} holds too for any $1 \leq i < n$. So, by induction, H_i holds for all $1 \leq i \leq n$. $H_n \Rightarrow x_n \searrow y_n \Rightarrow s_{end} \rightarrow y_n$ which is impossible because s_{end} has no descendant. So, $\exists 2 \leq i \leq n, y_i = x_i$ and:

$\sigma = \langle y, y_2, \ldots, y_i = x_i, x_{i+1}, \ldots, x_n = s_{end} \rangle$

By removing all duplicate states from σ we build a trace σ_y from y to s_{end}. This contradicts $y \in Q^{\dashv}$ and proves that $\delta(x) \in Q^{\dashv} \Rightarrow x \in Q^{\dashv}$. ■

This proof with one memory is trivially extended to multiple memories by replacing the single occupancy constraint by a conjunction of occupancy constraints.

4.8 Conclusion

The liveness analysis method presented in this chapter is flexible in many aspects. First, it is independent of the periodicity of applications, allowing workloads with different periods. Second, it does not require timing nor mapping information for tasks, leaving flexibility at run-time for the mapping and execution of tasks onto PEs. Third, it can be used to design exact deadlock handling strategies which do not prevent any safe schedule from being selected, be it at design time or at run time by the scheduler. All these aspects make this approach usable with a wide variety of possible implementations and objectives for the scheduling phase.

This contribution presents a significant improvement in the efficiency for exact analysis, by greatly reducing the state-space of the automaton to be analyzed in comparison with the conventional exact analysis. This enables the use of an *exact* analysis for more complex workloads than would be possible with the conventional approach.

The next chapter presents the third contribution of this thesis: how the deadlock prevention rules produced by the analysis presented in this chapter can be used at run-time in an execution environment, and the implementation of such an environment.

Chapter 5

Experimental Evaluation

5.1 Introduction

In this chapter we compare the first contribution (presented in Chapter 3) and the second contribution (presented in Chapter 4) to an existing state-of-the-art tool, namely `memDAG` [56]. Other tools presented in Chapter 2, such as SynDEx [27] and PREESM [61], were not included in this study. The main reason behind this choice is the lack of time and the simplicity of use of `memDAG`. Developing a testbed to compare `memDAG` and our contributions was not too difficult but doing the same with the other works would have required much more efforts. This is left for future works.

Note that our contributions support multi-memory systems, whereas `memDAG` currently is only working on single memory target systems. As such, our evaluation will be conducted in single-memory scenarios only.

We first compare the minimal memory size of the target platform that is supported by our heuristic, and by the `MinLevel`, `MaxSize`, `MaxMinSize` heuristics from `memDAG`. We did not use the `RespectOrder` heuristic of `memDAG` as it requires a valid schedule in input, and provides a solution greatly dependent on the quality of that schedule. In other words, we want to study the robustness of the heuristics to small memory sizes. We compare those minimal supported memory sizes with the theoretical minimum, which is computed using the exact liveness analysis. Then the permissiveness of our contributions and three different heuristics from `memDAG` is studied. Finally, the run-time of the heuristics of `memDAG` and of our implementations of Contributions 1

and 2 of this thesis is evaluated.

5.2 Implementation Details

5.2.1 First contribution: Deadlock prevention using MEGs

The first contribution of this thesis, the deadlock prevention mechanism presented in Chapter 3, has been implemented using the Python 3 language and `python-igraph`, a Python binding of the `igraph` library (implemented in C) to represent the application graphs and MEGs. It implements the two algorithms described in Chapter 3: Algorithm 1 and Algorithm 2. With regard to Algorithm 2, the order in which cliques are explored and buffers selected inside those cliques has not been configured and is down to the implementation of the `igraph` library. As explained in Chapter 3, studying alternative algorithms to find minimal oversized clique has not been studied during this thesis. The candidate dependency selected to remove a clique goes from the consuming task with the smallest due-date (i.e., having the tightest constraint) to the producer of the buffer whose consuming task has the largest due-date (i.e., the less constrained). This choice is a heuristic whose purpose is to minimize the impact of the added dependency on the makespan of the workload.

Inputs — Our implementation takes as input a file to represent the workload. This file lists the different application graphs that are part of the workload, and optionally their number of concurrent iterations. The program is also given the number and size of memories.

Outputs — During the analysis our implementation outputs to the standard output each oversized clique it finds. If a clique could not be removed, because the heuristic failed or the memory size of the target platform is too low to run the workload, the execution is aborted. At the end of the execution all artificial dependencies are saved in a file.

5.2.2 Second contribution: Liveness analysis

The optimized liveness analysis presented in Chapter 4, the second contribution of this thesis, has also been implemented using the Python 3 language and `python-igraph` to represent the application graphs and the control automaton. It has been

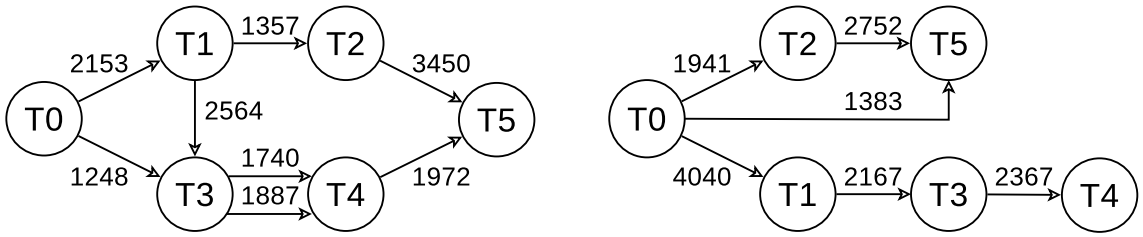


Figure 5.1: Two example applications used in our experiments. Numbers on edges represent buffer size

implemented in a program able to generate both the control automaton or the schedule automaton from a workload. The analysis of the conventional schedule automaton has been implemented to allow the evaluation of the gains in performance made when using the control automaton (cf. Section 4.6 of Chapter 4). Furthermore, analyzing the schedule automaton allows to compute the total number of possible schedules, which is useful to evaluate the permissiveness of approximate approaches.

Inputs — This program takes identical input as the implementation of the previous contribution to represent the workload. It also requires the number and size of memories to be specified in parameters. Additionally, another parameter is used to specify whether to use the control automaton or the conventional schedule automaton.

Outputs — Information on the run is outputted by the program in a file: the number of states in the analyzed automaton and how many states are live, the computation time, as well as the number of valid paths in the automaton. The number of valid paths obtained when analyzing a schedule automaton corresponds to the number of valid schedules of the input workload. Finally, our implementation also saves in another file the anti-deadlock rules presented in Section 4.5 of the Chapter 4.

5.3 Experimental Setup

To perform our evaluation of the different approaches, we use a testbench of 100 different workloads. Each workload is composed of 4 random application graphs, with each application having 6 tasks. We chose this workload size as it is the maximal size allowing for fast liveness analysis of the conventional schedule automaton, which is useful to evaluate the permissiveness of approximate approaches. The ap-

plication graphs were generated using SDF^3 [4] in a similar fashion to Section 4.6. The average buffer size is 2,048 *du*, and applications have a random topology for the dependencies between tasks. Figure 5.1 shows the topologies and buffer sizes of two example applications. The evaluation was conducted on a personal computer with 1 CPU core @ 3.50GHz, 32 GiB of RAM. The analysis running times are not polluted by swapping of memory pages by the OS: we explicitly forbade swapping in our configuration.

5.4 Minimal Supported Memory

We express by minimal supported memory the smallest memory size of the target platform for which a heuristic was able to produce a valid deadlock-free workload. This minimal supported memory is compared to the theoretical minimum, which can been computed while executing the liveness analysis presented in Chapter 4. Hereafter we call the difference between the minimal supported memory and the theoretical minimum given by our Contribution 2 the overhead. As explained in the previous section, experiments are run on 100 different workloads, each composed of 4 random application graphs made of 6 tasks each. Contribution 1 of this thesis performs better than `MinLevel` in 97 out of 100 experiments, as it is able to produce a valid deadlock-free workload graph at low memory sizes for which the `MinLevel` heuristic of `memDAG` fails. This can be seen in Table 5.1, with `MinLevel` having an average overhead of 13,790 while our heuristic average overhead is much smaller at 2,793. Contribution 1 performs better than `MaxSize` on 97 experiments, and identically on 1 experiment (for which both reach the theoretical minimum). On the other hand, Contribution 1 performs better than `MaxMinSize` in only 14 experiments. `MaxMinSize` is better than our first contribution in 69 experiments. In the remaining 17 experiments, both approaches show equal performance by reaching the theoretical minimum. Our heuristic reaches the theoretical minimum memory in 25 out of 100 experiments, while `MinLevel` never reaches it, `MaxSize` reaches it in only one experiment, and `MaxMinSize` reaches it 57 times. Overall our first contribution has a better ability to support low capacity memories than `MinLevel` and `MaxSize`, but is less efficient than `MaxMinSize`. Since our second contribution is an exact approach, it always supports the theoretical minimum memory size contrary to approximate approaches.

	Minimal supported memory			
4 apps. of 6 tasks	Minimum	Maximal	Average	Median
`MinLevel`	15,480	40,631	26,156	25,271
`MaxSize`	13,519	40,959	26,157	26,226
`MaxMinSize`	9,389	19,239	13,052	12,846
Contribution 1 (Chapter 3)	9,877	29,354	15,159	14,403
Contribution 2 (Chapter 4)[1]	8,508	18,154	12,365	12,629

Overhead of minimal supported memory (i.e., Minimal supported memory – Contribution 2)

4 apps. of 6 tasks	Minimum	Maximal	Average	Median
`MinLevel`	3,612	24,984	13,790	13,495
`MaxSize`	0	28,334	13,792	14,664
`MaxMinSize`	0	5,004	687	0
Contribution 1 (Chapter 3)	0	15,119	2,793	1,964
Contribution 2 (Chapter 4)	0	0	0	0

[1] The minimal supported memory of Contribution 2, an exact approach, is the theoretical minimum reachable.

Table 5.1: Minimal supported memory size of the approximate approaches, in comparison to theoretical minimum of Contribution 2

5.5 Permissiveness

To compare the permissiveness of our heuristic with `MinLevel`, `MaxSize`, and `MaxMinSize`, we ran 100 experiments using the same setup as the previous section. We then compare the number of remaining schedules of their respective deadlock-free output workloads. As the different heuristics are not always able to provide a solution guaranteeing to respect a given memory size, even if schedules respecting that size exist, we set the memory size at the minimum where all heuristics provide a solution. The theoretical total number of valid schedules has been computed by running the liveness analysis of Contribution 2 (Chapter 4).

As shown in Table 5.2, the median number of allowed schedules of Contribution 1 (1.00×10^{15}) is the smallest of the four heuristics, showing that it is less permissive than those provided by `memDAG`. This is reinforced by the fact that while our approach was the most permissive heuristic of the four in 27 experiments, it was also the less permissive in 52 experiments. On the other hand, `MaxMinSize` is the most permissive

	Number of allowed schedules			
4 apps. of 6 tasks	Minimum	Maximal	Average	Median
`MinLevel`	2.34E+11	6.97E+28	1.13E+27	2.54E+19
`MaxSize`	3.59E+11	1.28E+28	4.06E+26	2.09E+20
`MaxMinSize`	6.34E+15	4.43E+29	6.21E+27	1.86E+23
Contribution 1 (Chapter 3)	4.32E+03	1.10E+30	2.16E+28	1.00E+15
Contribution 2 (Chapter 4)[1]	6.40E+22	3.99E+33	8.69E+31	2.53E+28

Fraction of allowed schedules (i.e., # Allowed schedules / # Contribution 2)				
4 apps. of 6 tasks	Minimum	Maximal	Average	Median
`MinLevel`	9.11E-19	1.11E-03	1.83E-05	2.56E-09
`MaxSize`	3.27E-15	9.47E-03	1.76E-04	5.63E-09
`MaxMinSize`	5.70E-12	9.33E-03	3.70E-04	8.66E-06
Contribution 1 (Chapter 3)	1.94E-26	7.87E-03	2.91E-04	3.14E-12
Contribution 2 (Chapter 4)	1	1	1	1

[1] The number of schedules allowed by Contribution 2, an exact approach, is the total number of valid schedules.

Table 5.2: Number of allowed schedules by approximate approaches, in comparison to the theoretical maximum of Contribution 2

of the four heuristics in 59 experiments, and the less permissive in only 1 experiment. As such, `MaxMinSize` can be regarded as the heuristic providing the most flexibility for the schedule of the deadlock-free workload it produces among the four heuristics studied here. Since our second contribution is an exact approach, it always allows *all* valid schedules and is more permissive than all approximate approaches.

5.6 Computational Time

In this section we discuss the computational time taken by the different approaches. This is the time taken by the tools to perform their respective computation, with the same input memory size as in previous section. It should be noted that low memory levels are worse case situations for `memDAG` and our first contribution (presented in Chapter 3) as they require to add more edges to produce the final result. On the other side, a low memory level is a best case situation for the liveness analysis of

Contribution 2, since the total number of states in the automaton is lower, hence making its analysis faster.

	Computational time (in s)			
4 apps. of 6 tasks	Minimum	Maximal	Average	Median
`MinLevel`	0.017	0.040	0.028	0.027
`MaxSize`	0.015	0.026	0.019	0.018
`MaxMinSize`	0.016	0.039	0.020	0.019
Contribution 1 (Chapter 3)	0.054	0.715	0.276	0.264
Contribution 2 (Chapter 4)	0.143	0.416	0.242	0.224

Table 5.3: Computational time of the five approaches: three heuristics from `memDAG`, and our implementation of the first and second contributions

From Table 5.3 we can observe that the exact liveness analysis takes a higher amount of time to produce its results than either `memDAG`, and a similar one to our first contribution. This is the case even in these experiments with low target memory size, a scenario which is favorable to the liveness analysis and unfavorable to the other approaches.

Both our contributions were first implemented fully in the Python 3 language, using a python library for graph representation. This first implementation was slow and took a few minutes to hours to study the workloads of our testbench. This is why we switched from this python library to the C-implemented `igraph` library, which led to a significant increase in performance with computation times falling under a second. Whether switching the remainder of the code to C or another compiled language would lead to significant improvement in performance remains an open question.

Our implementations do not make use of multi-threading. Given the nature of the algorithms designed in the first contribution, especially the loop of MEG creation/update followed by the oversized clique finding, we think that this contribution is not well suited for multi-threading. On the other hand, the liveness analysis could benefit from multi-threading. Indeed, each state of an automaton must be analyzed after its descendants but can be analyzed in parallel to states that are not in an ancestor/descendant relation. If we take the example of Figure 4.5, states $(02, -, 1345)$ and $(0134, -, 25)$ could be analyzed in parallel, whereas $(01, -, 2345)$ and $(0134, -, 25)$ could not, as the latter is a descendant of the former. This leaves

rooms for potential improvement of the performance of the liveness analysis, especially of larger automaton that have more parallelism opportunities.

For the liveness analysis, memory usage is a factor limiting the size of analyzable workloads. Indeed, larger workloads than the ones studied in Section 4.6 led to the saturation of the 32 GiB of RAM when studying the conventional schedule automaton. For our control automaton, the largest workload supported before reaching the saturation of the 32 GiB RAM were the following: 8 applications of 6 tasks, 7 applications of 8 tasks, and 6 applications of 10 tasks. Reducing the memory footprint of our implementation is an important work in order to make it usable for more complex workloads.

5.7 Discussion

Overall, we can see that the method presented in Chapter 3 can support lower memory sizes than the `MinLevel` and `MaxSize` heuristics, but is less efficient than `MaxMinSize` in that regard. On the other hand, the heuristics from `memDAG` allow for more flexibility in the selection of a schedule than Contribution 1 of this thesis. Since our first contribution computational time is significantly worse that those of `memDAG`, The conclusion is that for single memory systems, `memDAG` is likely to be a better option, as it is likely to produce a more permissive result. Using the approach from Chapter 3 is a suitable choice for inputs for which all the heuristics of `memDAG` fail to produce a deadlock-free workload graph, or for multi-memory systems which `memDAG` does not support at the time of writing.

Our second contribution provides an optimal solution as it is an exact approach. As such it has a greater permissiveness than our first contribution or the heuristics of `memDAG`, and is always able to provide a valid solution for the theoretical minimum memory size.

Evaluating the impact of the different heuristics on the best possible make-span is an interesting future work, given that both our first contribution and the `MinLevel` heuristic of `memDAG` aim at minimizing their impact on the make-span. Furthermore, other approaches could be integrated into this evaluation as well.

In the introductory chapter (Chapter 1), we stated our goal to develop methods to avoid the issue of deadlocks caused by memory shortages. We have proposed

two different approaches to address the issue. Since the liveness analysis offers more flexibility, both by being maximally permissive and by supporting applications with independent periods, we selected this approach to be used in a run-time environment executing dataflow applications and targeting embedded systems. In use cases where the liveness analysis of the control automaton is too computationally expensive to be run dynamically, it should instead be run at the design-time of the system, and the anti-deadlock rules provided by our implementation be used by a run-time environment managing the execution of the workload. Such a mixed static / dynamic scheduling is presented in the next chapter (Chapter 6).

Chapter 6

Run-Time Environment

6.1 Introduction

Chapters 3 and 4 each presented a method to prevent memory shortages (and therefore the deadlocks caused by the shortages) in embedded systems. Of the two methods, the second method has the advantage of doing exact prevention, i.e., all possible non-deadlocking schedules are allowed. The results from this memory usage analysis can be used in combination with a run-time environment in order to fully exploit its flexibility. This chapter is a discussion about the design of a run-time environment that can use the results of the liveness analysis presented in Chapter 4. Section 6.2 presents the functionalities that a run-time environment should provide for the integration of the results of the proposed liveness analysis, and the potential for parallelism when providing those functionalities. Section 6.3 studies whether existing environments can be used or modified to exploit the results of the liveness analysis. When it makes sense it also briefly presents the needed modifications. Then Section 6.4 presents in details an example implementation designed to maximize the parallelism of the control, including its architecture, configuration files, and internal data structures. Finally, Section 6.5 concludes the chapter.

6.2 Functionalities of a Run-Time Environment

In this section we will study the various roles of a run-time environment with a particular emphasis on the parallelization opportunities. The reason for this emphasis is twofold. Indeed, with the increase of complexity (and dynamic variability) of the intensive data processing applications, like machine learning or 4G-5G baseband digital signal processing, the control software becomes more and more demanding. On the other hand, recent high-end data processing architectures (e.g. Xilinx Zynq UltraScale+ [72], NXP S32V2 vision processors [59], etc.) embed several general purpose CPU cores that can be used to improve the degree of parallelism of the control software, and consequently its reactivity. While monolithic, purely sequential, control software architectures are still suitable for entry-level applications, high-end data processing systems will probably require a higher degree of parallelism to meet their tight performance constraints.

In order to perform the execution of the requested applications, a run-time environment has to provide a variety of functionalities, that can be split in four main categories:

1. exchanges with the external environment,
2. buffers and memory management,
3. management of execution nodes and tasks,
4. deadlock handling policy (e.g., use liveness analysis results to prevent them).

For the *exchanges with the external environment* the run-time environment has to receive the user's commands, such as starting or stopping an application, and to manage the transfers of source and sink data back and forth. These two concerns can be separated into several parallel execution threads, or be managed in a single execution thread. Since data transfers with the external environment are usually dependent on the user's command, it could be that separating the two concerns in distinct execution threads provides little added value over a unique manager, with regard to the gain in parallelism versus the overhead in synchronization cost and system complexity.

The *buffers and memory management* consists of the allocation and deallocation of the buffers used by the running applications. It can very naturally be parallelized with one execution thread per memory unit in the target platform.

The *execution management* consists in three main aspects. The first one is the overall management of tasks: their status, execution priority, etc. The task management is independent for each PE and as such can easily be parallelized with one execution thread per PE, but can also be managed in a single execution thread for the whole system, or some intermediate level of parallelism. The second aspect is the effective scheduling of tasks, i.e., selecting which task to execute next, based on buffer availability, task priority, and overall scheduling policy. This aspect is constrained by the deadlock handling policy. Finally the third aspect is the actual computation of the task, be it in software or in hardware for hardware accelerator PEs. This aspect can, and should, be parallelized since it is the precise purpose behind the development of MPSoCs and other parallel architectures. The most reasonable choice to fully exploit the potential for parallelism provided by the target platform is thus to have one execution thread per PE, separated from the task management and scheduling aspects.

The *deadlock handling policy* is a critical part of the run-time environment to ensure the safe execution of the applications. By virtue of the liveness analysis presented in Chapter 4, it is constrained in its potential for parallelism. This is explained by the system-wide view of the analysis, which requires a coherent view of the status of tasks and buffers across the whole system. Furthermore, since the liveness analysis considers *atomic* transitions, i.e., no two tasks can change status simultaneously, it necessitates the establishment of a synchronization mechanism for the scheduling of tasks, in order to avoid multiple scheduling decisions being taken simultaneously and leading to an incoherent, and potentially deadlock-inducing, system state. As such, deadlock handling cannot be parallelized, and the scheduling of tasks must either be performed in a global execution thread or, if divided across multiple execution threads –a common example would be one execution thread per PE–, a synchronization mechanism such as a lock must be used. This is the principal limitation in parallelism across the whole run-time environment and as such must be implemented in the most effective manner if one wants to avoid having an unreasonable overhead from the run-time environment.

6.3 Design Choices to Integrate the Deadlock Handling Policy

Since the liveness analysis presented in Chapter 4 does not allow for multiple concurrent scheduling decisions, any run-time environment that takes as input the results of the analysis must have a compatible architecture. In Chapter 2, multiple deadlock handling tools that provide a run-time environment were presented. due to their lack of run-time flexibility.

StarPU [8] is a task based dataflow run-time oriented toward HPC. The StarPU run-time architecture is composed of a central scheduler and a set of workers, with each worker managing the execution of jobs onto a PE (or set of homogeneous PEs). Since the scheduling and memory management are centralized, our deadlock handling strategy could be integrated into the central scheduler, with little change to the overall architecture of the run-time (i.e., a centralized control architecture).

The PRUNE run-time [15] is used to run applications that have been guaranteed at compile time to be deadlock-free. It does not implement a scheduler, instead running task inside threads provided by the GNU/Linux `pthread` library and leaving the scheduling to the underlying operating system. As such, the PRUNE run-time offers very little control of the execution of jobs. Integrating our deadlock handling strategy into the PRUNE run-time would defeat its purpose of been lightweight, since it relies on static decidability of the deadlock-freeness and memory-boundedness properties provided by the PRUNE MoC.

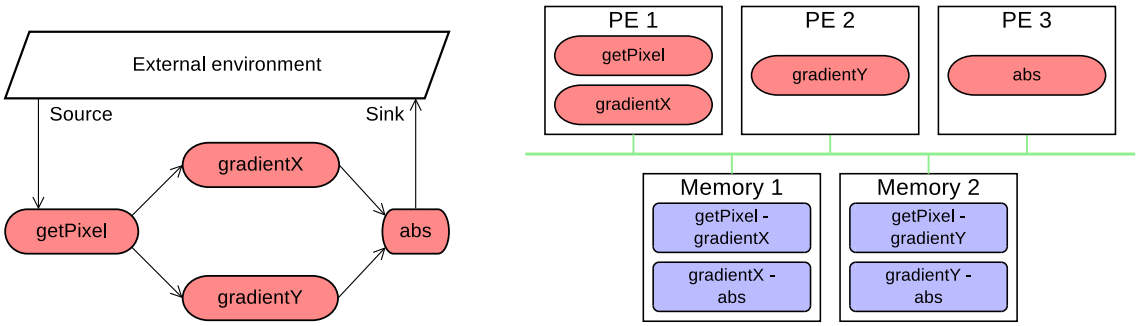
The SPIDER [39] run-time divides the control in two types of thread. The overall management of applications is done in a centralized fashion by the Global Run-Time, which makes all the mapping and scheduling decisions. Then, there is one Local Run-Time per PE, which manages the execution of jobs. Our deadlock handling strategy could be integrated into the Global Run-Time, leading to a centralized control architecture.

XKaapi [33], similarly to StarPU [8], is also a run-time environment oriented toward HPC. XKaapi runs with one thread per PE, each thread being responsible for the execution of tasks onto their associated PE. There is no central control for the execution of tasks, instead each task has the possibility to create new tasks. For example a task might create the tasks that consume its output buffers. Each thread has

a FIFO containing the tasks it will have to execute. When this task FIFO gets empty, a thread can perform work-stealing, i.e., take a task ready for execution from the task FIFO of another thread. To integrate our deadlock handling mechanism into the XKaapi run-time a new control thread dedicated to deadlock prevention should be introduced, with synchronization mechanisms to prevent execution threads from selecting tasks before the deadlock prevention thread propagated its messages. Symmetrically the execution threads shall inform the deadlock prevention thread about their scheduling decisions or, more precisely, their data buffer allocations.

HMBE [70] is a run-time designed for HPC systems. It uses a single control thread, which dynamically maps and schedules jobs. Jobs are executed in software threads called worker threads. Worker threads are configured to execute one actor, and will run one execution of said actor when requested from the control thread. There are up to n worker threads per job, with n the number of PE. When threads are not active they are in a dormant state and do not use computing resources. It is possible to configure HMBE so that more worker threads can be activated simultaneously than there are PEs, in which case context switching will occasionally occur to ensure progress on the execution of all active threads. Integrating our deadlock handling policy into HMBE is feasible, in the same way as for XKaapi [33].

The conclusion from the study of existing run-time environment is that it is possible to implement the approach from Chapter 4 into some existing HPC-oriented run-times but it is probably of limited interest: they only support homogeneous architectures with very large memories while the work of this thesis is focused on deploying applications onto heterogeneous architectures with distributed memory and size-limited memory units. The integration is also possible with less HPC-oriented run-time environments such as StarPU [8] and SPIDER [39], where it makes much more sense. However, these run-time environments have a centralized control architecture, while, as explained in Section6.2, we want to experiment with a control architecture design to maximize its parallelism, i.e., which provides its functionalities using a decentralized control architecture. With such a goal, the easiest path is to design a new run-time environment. Such an implementation and its architecture are presented in details in next section.



(a) Example application dependency graph, a Sobel filter.

(b) Mapping of the Sobel application onto a target platform with 3 PEs and 2 memories.

Figure 6.1: Example application and mapping

6.4 Architecture of our Run-Time Environment

In the previous section the functionalities that a run-time environment should provide have been presented, as well as the potential for parallelism for an implementation of those functionalities. This section describe the architecture of prototype implementation we designed, where the choice was made to maximize parallelism.

When needed we will use the example application (a Sobel filter) which dependency graph is represented on Figure 6.1a, mapped onto the target platform with 3 PEs and 2 memories represented on Figure 6.1b.

6.4.1 Global Overview

Before entering into the details let us first present the global overview. Figure 6.2 illustrates our software architecture.

The run-time environment we propose is a multi-threaded, POSIX-compliant, C program, referred to as the *server* in the following. There are 2 threads per PE, one thread per memory unit, plus two common threads, which role and interactions are detailed in the following subsections:

- Each PE has a *control thread* (`PeCth`) and an *execution thread* (`PeXth`).
- Each memory unit has a control thread (`MuCth`).
- The *general thread* (`Gth`) is responsible for the interface with the environment.

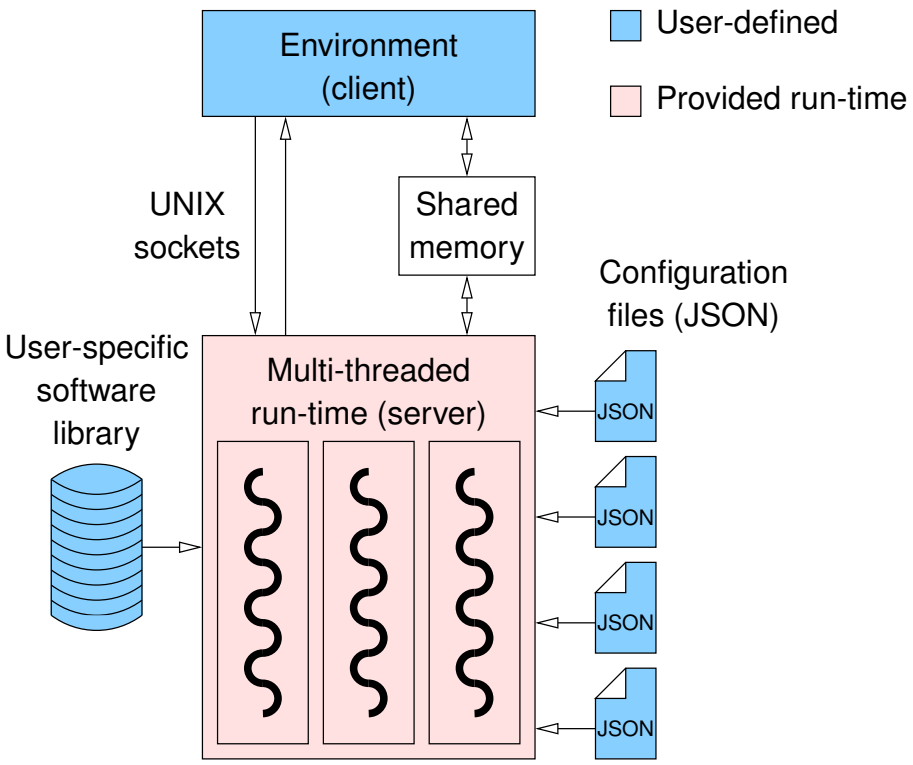


Figure 6.2: Software architecture

- The *deadlock prevention thread* (`Dth`) is responsible for the deadlock prevention.

With the example of Figure 6.1 the server would execute 10 threads in parallel: 7 for the control and 3 for the tasks execution.

The code of the server does not need any modification to run on different platforms or to execute different workloads. It is compiled once for all for a given target CPU architecture. When launched on the control CPU of the target platform, the server is passed configuration files in JavaScript Object Notation (JSON) format and a software library containing all the user-specific code. The server configures itself at initialization time based on the content of the configuration files: they define the target platform (number of execution nodes, number of memory nodes), the applications (tasks and buffers, maximum number of concurrent iterations per application), the mapping of tasks to execution nodes and of buffers to memory nodes, and the deadlock prevention rules. The number of threads spawned by the server, for in-

stance, depends on the number of execution and memory nodes specified in the configuration files. The software library containing the user-specific code is loaded dynamically by the server during its initialization. It must contain all objects referenced by their names in the configuration files: functions for the tasks and the data structures for their control parameters or returned statuses, of course, plus, optionally, custom schedulers and/or memory managers. The user-specific code must be coded according the provided API.

The running applications are considered as periodic and the JSON configuration files defining them indeed define one iteration only, as a straightforward textual representation of the application's AHSDF graph. Several iterations of the same application can run concurrently. In the following we distinguish a task, that is, a node of the AHSDF graph, and the task instances, that is, the different executions of the task in consecutive iterations. We name the latter "*jobs*". Consecutive iterations of an application are considered as independent, except that a rule of "reasonable behavior" is implemented by the server: a job of iteration N cannot start before the job of the same task of iteration $N-1$.

In order to conduct our liveness analysis a maximum number of concurrent iterations must be specified for each application, such that the control graph product can be constructed and analyzed. The "reasonable behavior" constraint, the task-to PE static mapping, if it is known, plus all known timing characteristics, can be used to reduce the size of the control graph, speedup its building and analysis, and reduce the number of resulting deadlock prevention rules. The per-application maximum number of concurrent iterations are also added to the JSON configuration files because this information is needed by the server to dimension its internal control data structures.

Buffers and jobs are uniquely identified by a combination of their application identifier (`appID`), their iteration number and their local buffer (`bufferID`) or task (`taskID`) identifier inside their application AHSDF graph. When a deadlock prevention rule refers to a data buffer or a job it uses a relative iteration number, relative to the oldest currently running iteration. This way, the same deadlock prevention rules are recycled each time iterations terminate or start. This is a bit like if the analyzed control graph product was anchored to the beginning of times and limited to the defined numbers of iterations.

One last software component is responsible for the high-level control and is referred to as the *environment* or the *client* in the following. It is provided by the user and must be coded according the server's API. The client is another independent program responsible for deciding when to launch a new iteration of the applications of the workload, for providing the input data buffers of the source tasks of each iteration and for receiving the output data buffers of the sink tasks. In a wireless communications context, for instance, where the server would execute the baseband DSP (the *physical layer*), the client could be the Medium Access Control (MAC) layer or an interface layer between the physical and MAC layers. The client can be developed in any programming language provided it communicates with the server using the specified protocol, both for data and control information.

In the current implementation the data exchanges between the server and the client use a shared memory area which layout is entirely under control of the client. The shared memory area contains data buffers and parameters consumed or produced by the source or sink tasks of the workload. The server knows absolutely nothing about their content, format or organization. As the shared memory area is actually accessed only by the client and by the tasks, and as the code of the client and the tasks is provided by the user, the only thing that the server does is receive from the client addresses falling in the shared memory area, and pass them to the source and sink tasks it launches such that they know where to find their input or output buffers and their control parameters inside the area.

The control commands, referred to as *signals* in the following, are exchanged through a pair of UNIX sockets. Their format is specified in the API.

Other communication mechanisms could very easily be implemented. The signals, for instance, are serialized before sending and thus do not depend on the communication medium.

6.4.2 Server Configuration

The server takes several types of configuration files in JSON format as its input. They describe the target platform, the applications being run, their mapping on the platform, and the deadlock prevention rules. The JSON format has been selected because it is human readable and commonly used for configuration files. Of course,

other formats could easily be used, even less human readable ones. And these files do not have to be written manually. The files describing the applications, the platform and the mapping can be automatically generated from DSE frameworks or Integrated Development Environment (IDE) tools. They are straightforward, we do not detail them further.

The deadlock prevention rules are extracted from the exact liveness analysis presented in Chapter 4. Each rule has two components: a firing condition, which is the set of buffers that must be allocated for the rule to apply, and the list of the jobs that must be blocked if the condition holds. A buffer is identified by its application identifier, (`appID`), the iteration number relative to the oldest currently running iteration of the application, and its local identifier inside its application graph (`bufferID`). The jobs to block are identified in the same way with their `taskID` instead of the `bufferID`.

An example file with two deadlock prevention rules is given below. It applies to a workload with three applications: the Sobel filter of Figure 6.1 (`appID` 0), plus a SUSAN filter (`appID` 1) and a JPEG encoder (`appID` 2). As such, the rule `0` in this example file would lead to block the job responsible for the `gradientY` task (`taskID` 2) of the second oldest running iteration. This rule would apply if buffers `getPixel-gradientY` (`bufferID` 1) and `gradientX-abs` (`bufferID` 2) from the second oldest iteration of the Sobel filter, alongside two buffers from the SUSAN filter and one from the JPEG encoder are simultaneously allocated in their respective memory units.

```
{
    "Rules":[{
        "rule_id": 0,
        "buffers": [
            {"appID": 0, "bufferID": 1, "iter_num": 1},
            {"appID": 0, "bufferID": 2, "iter_num": 1},
            {"appID": 1, "bufferID": 2, "iter_num": 0},
            {"appID": 1, "bufferID": 2, "iter_num": 1},
            {"appID": 2, "bufferID": 0, "iter_num": 0}
        ],
        "tasks_blocked": [ {"appID": 0, "taskID": 2, "iter_num": 1} ]
    },{
```

```
        "rule_id": 1,
        "buffers": [
            {"appID": 0, "bufferID": 3, "iter_num": 0},
            {"appID": 0, "bufferID": 3, "iter_num": 1},
            {"appID": 1, "bufferID": 5, "iter_num": 0},
            {"appID": 1, "bufferID": 5, "iter_num": 1},
            {"appID": 2, "bufferID": 3, "iter_num": 1}
        ],
        "tasks_blocked": [ {"appID": 2, "taskID": 1, "iter_num": 0} ]
    }]
}
```

6.4.3 Data Structures

In order to provide the functionalities, the server has to store relevant information into data structures suitable for the selected architecture and applications. The main data structures used in this implementation represent jobs and data buffers. For performance reasons, all these data structures are allocated by the server at initialization time, another reason why the maximum number of concurrent iterations of each application must be known. When an iteration terminates the corresponding data structures are recycled for the next new iteration of the same application. This mechanism is implemented as a kind of circular buffer, which is very efficient because it is not the data structures that move when iterations start or end but the pointer to the oldest iteration and the iterations counter that are updated.

Jobs

A job is one particular instance of a computing task. It consumes a set of input buffers and produces a set of output buffers. For sake of implementation simplicity, in the current version, a job runs on a statically allocated PE and cannot migrate at run-time, even though the liveness analysis presented in Chapter 4 does allows dynamic mapping and migration of jobs.

A job is represented by a data structure that carries control information like its complete identifier (`appID`, iteration number, `taskID`), the list of its input buffers, the

list of its output buffers, pointers to its `PeCth` and `PeXth`, its current state... At initialization time a job is created in the `IDLE` state, which means that its input buffers are no yet available and the job cannot be scheduled. Over its lifetime, a job can be in the following states:

- `IDLE`: the job has been created but it is not yet runnable because its input data buffers are not all available.
- `IDLE_BLOCKED`: same as `IDLE` but a deadlock prevention rule blocks the job; it shall not be scheduled until it is unblocked.
- `RUNNABLE`: the job has been created and all its input data buffers are available, it can be scheduled.
- `RUNNABLE_BLOCKED`: same as `RUNNABLE` deadlock prevention rule blocks the job; it shall not be scheduled until it is unblocked.
- `SCHEDULED`: the job has been scheduled for execution.
- `DONE`: the job execution has been completed.

Figure 6.3 shows the Finite State Machine for job states, displaying the conditions for changing the state of a job.

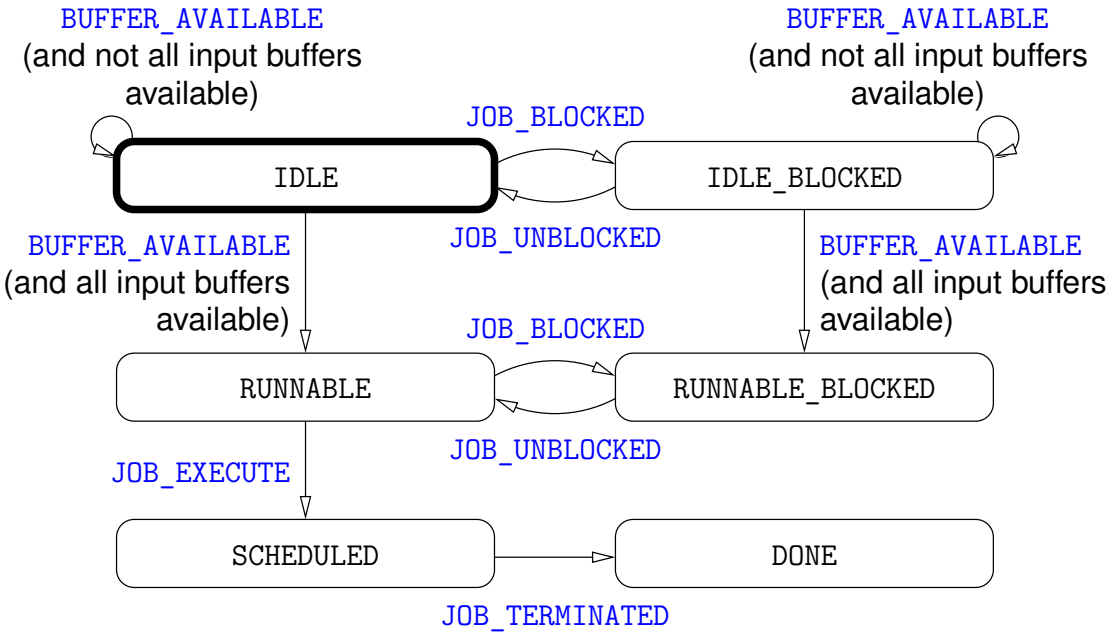


Figure 6.3: Finite State Machine of job status.

A job is put in the `IDLE_BLOCKED` (respectively `RUNNABLE_BLOCKED`) state when

it is in the IDLE (respectively RUNNABLE) state and its PeCth receives a JOB_BLOCKED notification from the Dth that this job cannot be scheduled, as a result of the deadlock prevention algorithm. It is marked as unblocked when its PeCth receives a JOB_UNBLOCKED notification that this job can be scheduled again. When a job is in IDLE or IDLE_BLOCKED state its PeCth monitors the BUFFER_AVAILABLE notifications about its input buffers until all have been received. The job is then put in RUNNABLE or RUNNABLE_BLOCKED state.

Transfers

A transfer is just a type of job. It is one particular instance of a data transfer task. It consumes one single input buffer and produces one single output buffer. A transfer runs on a statically allocated suitable PE, such as a DMA engine.

Transfers can happen in any kind of memory architecture. For example, they are useful in NUMA architectures to move the input data of a task in a different memory from the one they were produced in, since PEs might not be able to access all memories. This is commonly used to move a buffer from the output memory of one hardware accelerator to the input memory of another hardware accelerator.

Buffers

A buffer is allocated in the memory it has been statically mapped to. A buffer has one single producer job and one or several consumer jobs. Buffers are managed by the MuCth of their memory unit. This MuCth tracks the termination of the producer and consumer jobs of its buffers.

Like jobs, a buffer is represented by a data structure that carries its complete identifier (appID, iteration number, bufferID), its producer job, the list of its consumer jobs, a pointer to its MuCth, its size, its address in its memory unit, its current state... At initialization time a buffer is in the VOID state, which means that it has not been allocated yet. Over its lifetime, the buffer structure can be in the following states:

- VOID: the buffer is not yet usable, as it is not yet allocated in memory.
- ALLOCATED: the buffer has been allocated into memory and is ready to be filled by its producer job.

- `AVAILABLE`: the buffer has been filled by its producer job, it is ready to be read by its consumer jobs.
- `FREED`: all consumer jobs terminated, the buffer is no longer needed, it has been freed.

Figure 6.4 shows the Finite State Machine of buffer states, displaying the conditions for changing the state of a buffer. When in the `AVAILABLE` state the status of the buffer is re-evaluated each time its `MuCth` receives a `JOB_TERMINATED` notification about one of a consumer job.

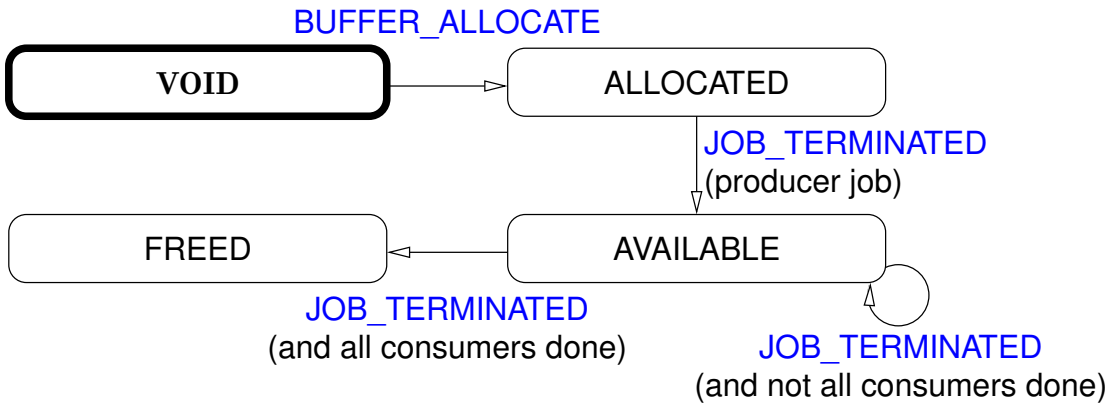


Figure 6.4: Finite State Machine of buffer status.

6.4.4 Threads

Figure 6.5 shows the architecture of the existing implementation. It is composed of a set of threads each focused on a specific part of the system. Threads communicate through events, which are described in Subsection 6.4.5.

The general thread (`Gth`) is the overall controller of the system. It receives instructions from the client about application iterations starts and stops, about the organization of the shared memory used for data exchanges with the client, and the availability of source jobs' input data in the shared memory. In return it signals to the client the termination of application iterations and the availability of sink jobs' output data in the shared memory. Once an application is running the `Gth` manages its execution: when the client requests the start of a new iteration, it recycles the data structures (jobs, buffers...) of a terminated iteration for the new one and it injects

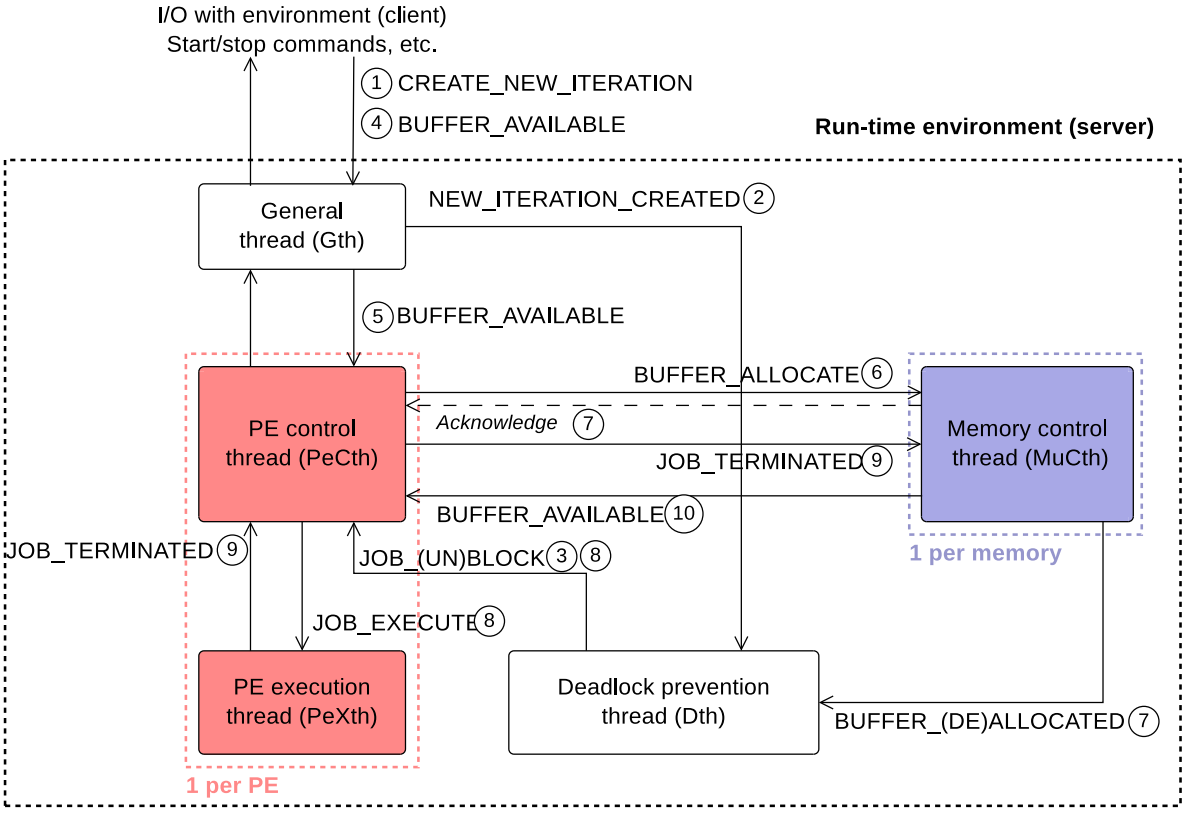


Figure 6.5: Architecture of the server. Circled numbers correspond to the step in which the event is sent in the example run.

these new jobs and buffers into the pool of existing objects managed by the other threads.

The deadlock prevention thread (`Dth`) aims to determine which deadlock prevention rules to apply at a given instant. It receives information on buffers allocations and deallocations from all memory unit control threads (`MuCths`). Based on this information it checks which rules start or stop applying. When the conditions of a rule are met, and a job j that was not blocked must be blocked, it sends a blocking event `JOB_BLOCK`(j) to the `PeCth` of job j (where j is the job's unique identifier). Conversely, once the conditions of a rule are no longer met, and a job j that was blocked can again be scheduled, it sends an unblocking event `JOB_UNBLOCK`(j) to the `PeCth` of job j. The `Dth` also receive notifications from the `Gth` about the creation of new iterations because a newly introduced job, even if it is in `IDLE` state, can be blocked at initialization time if the current state of the memory units forbids the allocation of its output

buffers.

Memories each have a control thread (MuCth) that manages the underlying memory, allocating and deallocating buffers as needed. The MuCth receives from the PE control threads (PeCths) allocation request events BUFFER_ALLOCATE(b) requesting it to allocate a given buffer b. It also receives JOB_TERMINATED(j) events indicating that the execution of a job j is completed. This allows to indicate which buffers have been filled with data and to send that information, via buffer availability events BUFFER_AVAILABLE(b), to the PeCths of the jobs having that buffer b as input. This also allows, once all consumer jobs of a buffer are completed, the immediate deallocation of the buffer that is no longer needed. As already mentioned, the MuCths notify the Dth about buffers allocations and deallocations.

When processing BUFFER_ALLOCATE requests the MuCths must allocate a region in their memory unit. By construction of the deadlock prevention it is guaranteed to be always possible on a pure memory occupancy point of view. Finer grain aspects like memory fragmentation are not covered in this prototype implementation. A default memory management algorithm is provided, that selects the first suitable memory region to accommodate the requested data buffer, and assumes that memory fragmentation will never be an obstacle. This is sufficient, for instance, when the data buffers in the considered memory unit all have the same size. In more complex situations a solution to this problem could be to oversize the actual memory unit or to expose a size less than the actual one to our liveness analysis. What the margins should be and how to verify that they suffice is also not covered in this thesis and left for future works. However, in order to address this fragmentation issue, software hooks allow users to define their own memory management policies and use them instead of the default one. They must be designed according the provided API and are provided to the server in the form of a dynamically loaded library, like the user-defined scheduling policies and all tasks implementations.

PEs each have a control thread (PeCth) and an execution thread (PeXth). It is important to remember that these threads run on general purpose CPUs that can be different from the PE itself. If the PE is a hardware accelerator, for instance, its control and execution threads run on one or two CPUs. The execution thread only implements the low-level communication and control of the PE; the data processing is performed by the hardware accelerator. If the PE is itself a CPU, it executes

its execution thread but its control thread can run on the same or a different CPU, depending on the mapping.
The role of the PeCth is to manage the jobs to be executed, as requested from the Gth. It selects which job the PE should execute next among jobs that are ready (all input buffers are available), while following the directives from the Dth, i.e., blocked job are not selected until an unblocking event JOB_UNBLOCK is received. Once a job has been selected, the PeCth sends the buffer allocation request events BUFFER_ALLOCATE necessary to perform that job, thus requesting the MuCths to allocate the output (and optional working) buffers. Buffer allocation requests are guaranteed to be granted by the MuCths thanks to the deadlock prevention rules. The PeCth then waits for the acknowledge back from the MuCths indicating that the requested output buffers are ready before ordering the corresponding PE execution thread to perform the job via the execute event JOB_EXECUTE. A default scheduling algorithm is provided, but users can define their own scheduling policy (or set of policies). Just like for memory management policies users provide their custom schedulers as part of the dynamically loaded library.
The PeXth role is to control the underling hardware (e.g., hardware accelerator) to perform the requested job, or to perform the necessary computations itself when it is implemented in software (e.g., tasks running on generic CPUs). It waits for execution events JOB_EXECUTE from its PeCth in order to launch job executions. Once a job execution has finished, it sends a job termination event JOB_TERMINATED back to its corresponding PeCth. The PeCth then forwards this event to the MuCths of the input and output buffers of the terminated job. A task's implementation, be it a pure software one or only the software driver controlling a hardware accelerator, is again provided by the user as part of the dynamically loaded library.

Let's illustrate how the threads interact with the example application and target platform of Figure 6.1. The following steps occur for the communication between threads. Events occurring within a step are annotated by the circled number of that step in Figure 6.5.

1. The client sends to the server a request to run one new iteration of the Sobel application. This request is received and processed by the Gth.
2. The Gth initializes the corresponding data structures (jobs and buffers) and makes them available to the other threads. It also informs the Dth because

an incoming job can be blocked at initialization time if the current state of the memory units forbids the allocation of its output buffers.

3. The Dth updates its internal state and informs the PeCths of blocked/unblocked jobs, if there are any.
4. The client populates the input buffer of getPixel in the shared memory area used to communicate with the server. It signals the availability of this input buffer to the Gth.
5. The Gth informs the PeCth1 that the input buffer of getPixel is available. As can be seen with this example the Gth also somehow acts as the equivalent of a MuCth for the shared memory area.
6. The PeCth1 selects job getPixel to execute. It sends two buffer allocation request events (one for the output buffer to gradientX, the other to gradientY) to their respective MuCth.
7. The MuCth1 receives the allocation request event for buffer getPixel-gradientX. It allocates the buffer, informs the Dth, and acknowledges to the PeCth1. The same happens with the MuCth2 for the getPixel-gradientY buffer.
8. Again, the Dth updates its internal state and informs the PeCths of blocked/unblocked jobs, if there are any. Note that this cannot impact the getPixel job that has already been selected for execution. The PeCth1 does not need to process these events immediately if it receives some. Simultaneously, the PeCth1 sends the execution request to the PeXth1.
9. The PeXth1 receives the execution events and starts the execution of job getPixel. Once completed it sends the termination event back to the PeCth1, which forwards it to the MuCths in charge of the input and output buffers of getPixel. As the input buffer of getPixel is in the shared memory area it is handled differently from the others but in other circumstances some input buffers could be freed because of the job completion and the responsible MuCths would deallocate them and inform the Dth.
10. The MuCth1 marks buffer getPixel-gradientX as available and send this information to the PeCth1, in charge of the consumer job gradientX. The same happens in parallel for buffer getPixel-gradientY with the MuCth2 and the PeCth2.

This procedure then continues in a similar fashion to steps 6 to 10 in order to perform the execution of the remaining jobs.

The execution of jobs by their respective PeXth can occur concurrently but the respective PeCths of jobs gradientX and gradientY must not take their scheduling decisions simultaneously, else memory shortages and deadlocks could occur. This exclusion is implemented via synchronization primitives, as will be explained in Section 6.4.6.

6.4.5 Events

An event is a data structure used for inter-threads signaling. There are different event types, depending on the type of sender and receiver threads and depending on the kind of carried information. In practice, events carry either a job or a buffer identifier, plus optional parameters.

Job-related events

This type of event is used in one of the following situations:

- When a PeCth schedules a job j, it sends the event JOB_EXECUTE(j) to its PeXth.
- Symmetrically when the execution of a job j terminates, the PeXth sends the event JOB_TERMINATED(j) to its PeCth.
- When the state of the Dth changes and, as a consequence of the deadlock handling policy, a job j cannot be scheduled any more (it is blocked), the Dth sends the event JOB_BLOCK(j) to the PeCth responsible for j.
- Symmetrically, when the state of the Dth changes and, as a consequence of the deadlock handling policy, a job j can again be scheduled (it is unblocked), the Dth sends the event JOB_UNBLOCK(j) to the PeCth responsible for j.
- When a job is terminated, the PeCth forwards the event JOB_TERMINATED(j) to the MuCths of memories holding one or several input or output buffers of the terminated job j. This has two purposes. It is used by a receiving MuCth to check if all consumers of an input buffer have terminated, and deallocate the buffer if it is the case, as it is no longer needed. It is also used by a receiving MuCth to update the status of one or several output buffers, since the buffers have been populated with data and are now available to consumers.
- When a job j that has at least one output buffer connecting to the client finishes, the PeCth sends the JOB_TERMINATED(j) event to the Gth. The Gth will then

signal to the client that the data is available in the shared memory.

Buffer-related events

This type of event is used in one of the following situations:

- When a PeCth schedules a job, for each output buffer b, it sends a BUFFER_ALLOCATE(b) event to the MuCth responsible for b. These events are acknowledged by the receiving MuCths to guarantee that the selected job starts its execution only after all its output buffers have been allocated.
- When a buffer becomes available, the PeCths managing the consumer jobs of that buffer should be notified. This is done via the event BUFFER_AVAILABLE(b), which is sent by the MuCth responsible for b. It is used by the receiving PeCths to update the status of input buffers of a job, and if all are available to update that job status.
- From the Gth to a PeCth that runs a consumer job of a source buffer b (between the client and the server) to signal that this buffer is available, with the event BUFFER_AVAILABLE(b).
- When a MuCth allocates or deallocates a buffer b it sends a BUFFER_ALLOCATED(b) or BUFFER_DEALLOCATED(b) event to the Dth such that the status of the deadlock prevention rules can be updated and some jobs can be blocked or unblocked if needed.

Iteration-related events

This type of event is used by the Gth to communicate with the Dth:

- When a new iteration is launched upon request by the client the ITERATION_CREATED(a) event is sent by the Gth to the Dth (a is the corresponding application identifier). The Dth updates its internal data structures and marks the corresponding jobs and data buffers as existing. If needed JOB_BLOCKED events are emitted for the new jobs.
- When a running iteration terminates and the Gth received all JOB_TERMINATED events for all sink jobs, it sends the ITERATION_TERMINATED(a) event to the Dth. The Dth shifts its internal data structures by one iteration for application a and

recomputes the related rules' and jobs' state. If needed JOB_(UN) BLOCKED events are emitted.

6.4.6 Synchronization

To guarantee that the deadlock prevention is properly implemented and that memory shortage is avoided, inter-thread synchronization is needed. In the current prototype implementation mutexes, semaphores and condition variables from the pthread POSIX library are used. Of course, other choices are possible, reason why we present here the principles but not the detailed implementation.

The heart of the run-time execution is a loop that repeats until the end of all applications. The synchronization is based on a unique scheduling lock (SchedLock). In practice other primitives are used to protect the shared data structures like event FIFOs, for instance, against concurrent access and to suspend threads until they have something new to do. The loop can be described as follows:

- None of the PeCths holds the SchedLock. When a PeCth does not hold the SchedLock, it waits for incoming events and resumes when it receives one. If it has runnable jobs it also waits for the SchedLock and also resumes if it acquires it. When receiving events, like BUFFER_AVAILABLE notifications, it updates its jobs' statuses and, if it has runnable jobs, it executes its scheduling algorithm to keep its list of runnable jobs up to date and ordered. If, due to the deadlock prevention, it has no runnable job any more, it stops waiting for the SchedLock.

- When a PeCth finally acquires the SchedLock (let us name it the *winner* PeCth) it finishes consuming the pending incoming events of JOB_BLOCK and JOB_UNBLOCK types. The synchronization guarantees that no new such events can arrive until it releases the SchedLock. The other PeCths continue their operations.

- If the winner PeCth, after processing the last JOB_BLOCK events, does not have any runnable job left, it releases the SchedLock such that another PeCth with runnable jobs can acquire it; the loop restarts from the beginning. Else it selects the one with highest priority and requests the allocation of all its output buffers from their respective MuCths.

- By construction of the deadlock prevention, it is guaranteed that all these output buffers can be allocated. Each requested MuCth allocates the buffers, notifies the Dth, and acknowledges the winner PeCth. The MuCths operate in parallel (except, of course, for the access to shared resources like event queues).

- The winner PeCth waits until all acknowledgments are received. It then sends a request to its companion PeXth to execute the job, and transfers the SchedLock to the Dth. Simultaneously the Dth receives the buffer allocations notifications from the MuCths, updates its internal state and sends JOB_BLOCK events if needed. Finally, it releases the SchedLock and the loop restarts from the beginning.

The real loop is slightly more complicated because of iterations start: when the Gth receives from the client a request to launch a new iteration it starts competing for the SchedLock, with maximum priority. As soon as it holds the SchedLock it notifies the Dth about the new iteration and transfers the SchedLock to the Dth.

In summary, the Dth must execute between two scheduling decisions (taken by the same or by two different PeCths). With the presented mechanism the Dth executes after each scheduling decision to update the system status, and no PeCth can take a scheduling decision before the update of blocked job from the previous scheduling decision is performed by the Dth.

6.4.7 Performance

To ensure suitable performance it is critical that the selection of the job to schedule and the update to the blocking status of jobs is done as fast as possible since they are in the critical section of the loop.

The high level of parallelism favors fast scheduling decisions because the PeCths, when they do not hold the SchedLock, can execute their scheduling policies each time they receive an event from other threads, to update their job lists and keep them properly ordered. This way, as soon as they acquire the SchedLock they can instantaneously pick the highest priority job.

The implementation is independent of the scheduling policies: a default scheduling policy is provided that selects the first candidate job in a list, but users can provide

their own, more sophisticated, scheduling policy. A scheduling policy is defined by a function that takes an unordered list of jobs and returns it ordered in decreasing order of priority. This API has been retained because of its simplicity but other, potentially more efficient, APIs are also possible. One could for instance use a stateful API where the scheduling policy maintains an ordered list of jobs and reacts to job status changes.

In order to speedup the update of the blocking status of jobs, the `Dth` uses a hierarchical data structure of linked objects. Each time the status of a buffer changes, thanks to the buffer-to-rule links, it identifies the potentially impacted rules, recomputes their status using bit-masks and detects rules' state changes. Using rule-to-job links it then finds the jobs which blocking state changes. Of course, the response time of the `Dth` depends on the size of the problem, that is, the number of rules, buffers, jobs and the number of links between them.

In the current implementation blocking and unblocking events are handled differently: `JOB_BLOCK` events must be sent to all destination `PeCths` before the latter can take their next scheduling decision. This is why the `Dth` holds the `SchedLock` and releases it only after it sent all `JOB_BLOCK` events. Unblocking events are less critical: if they are sometimes delayed the consequence is not deadlocks but only potentially sub-optimal scheduling decisions. Another design choice could thus be to propagate unblocking events is the same critical way as blocking events. The resulting system would probably be a bit less reactive but it would never take sub-optimal scheduling decisions because unblocking events were delayed.

6.5 Conclusion

This chapter studied how to use the result obtained from the liveness analysis presented in Chapter 4 in a run-time environment. Existing run-time environment have been examined in order to determine whether it would be possible or not to integrate the use of the liveness analysis results. The conclusion from this survey is that there were run-time presenting the suitable characteristics for such an integration. Some are less relevant than others because they target HPC systems. Others, more relevant, are centralized with non parallel control. In order to experiment with a run-time with multi-threaded control we developed a new solution, our **third contribution**,

and integrated our dynamic deadlock prevention strategy. This implementation is thoroughly described in the previous section of this chapter. Its key characteristic is a minimized critical section in which a `PeCth` acquires a scheduling lock, thus preventing other threads to take scheduling decisions, selects a job to schedule, requests the allocation of its output buffers, waits for the acknowledgments from the corresponding `MuCths`, launches the job on its companion `PeXth`, and finally releases the lock to the `Dth`. The `Dth` is informed of newly allocated buffers and signals blocked jobs, if there are any, before releasing the lock and any other `PeCth` can acquire it. This critical section is optimized thanks to a separation of the scheduling in a priority-based job ordering step, performed in parallel by all `PeCths` out of the critical section, and an instantaneous job selection, performed by one single `PeCth` at a time, inside the critical section. The `Dth` is optimized thanks to bit-masks operations and a dedicated data structure that links data buffers to rules and rules to target jobs. This prototype implementation has been tested on various example workloads.

Chapter 7

Conclusion

7.1 Contributions of the Thesis

The contributions of this thesis are focused on addressing the issue of deadlocks caused by memory shortages in embedded systems. Two approaches have been presented: a deadlock prevention mechanism based on the study of Memory Exclusion Graph (MEG), and a liveness analysis based on the analysis of a subset of the full state-space which have been shown to be of equivalent precision with regard to memory shortages. The latter contribution is shown to be of practical use by taking its results to develop a deadlock avoidance mechanism that is integrated into a run-time environment. This run-time environment is itself a prototype designed to by highly parallelized.

7.1.1 Deadlock Prevention Using the Memory Exclusion Graph

In Chapter 3 a deadlock prevention mechanism is presented. It is based on the addition of artificial dependencies into the workflow graph (representing all running applications) in order to make it impossible to select a schedule leading to a memory shortage. This approach is based on the analysis of a graph, the MEG, which represents buffers to be allocated in memory, and their potential for simultaneous allocation. By virtue of the properties of MEGs, this approach can only be used with applications sharing a common period (or extended to a common period) running on

single-memory platforms. These constraints limit the practical use of this approach. It is also further limited for its high computational cost, especially for an approximate (i.e., not fully permissive for scheduling) method. As such, the use of the *second contribution* of this thesis is preferred.

7.1.2 Liveness Analysis of Dataflow Applications on Multi-Memory Platforms

Chapter 4 introduces a liveness analysis for dataflow applications. This liveness analysis is derived from the conventional the study of the schedule automaton, which represents all possible states the system can reach and the transitions between those states. Instead of studying this full schedule automaton, a smaller graph is studied, the control automaton. The control automaton represents a subset of all possible states, as it excludes states followed by at least one deallocation, i.e., states in which at least one task is being executed. Theorem 1 proves that analyzing this smaller automaton is equivalent to analyzing the larger schedule schedule with regard to the issue of memory shortages. The practical use of this method has been assessed by using its results to design a deadlock avoidance mechanism which has been integrated into a prototype run-time environment.

7.1.3 Run-Time Environment with Deadlock Avoidance

In Chapter 6, the integration into a run-time environment of the deadlock avoidance mechanism derived from the second contribution is discussed. The software architecture of a run-time environment designed to by highly parallelizable and integrating the deadlock avoidance mechanism is presented.

7.2 Limitations and Improvements

As stated before the first contribution, presented in Chapter 3, is of limited practical use as it significantly constraints both the supported applications and the architecture of the target platforms. A possible way to circumvent those limitations is to divide the system to analyze in multiple sub-systems, which lifts the shared period con-

straint across different subsystems and, as the analysis is of exponential complexity, reduces the overall execution time to produce a solution. This approach however incurs an under use of the available memory, as each subsystem gets its own dedicated part of the memory, without leaving the possibility to let one subsystem use some of the memory allocated to second subsystem if it is not currently using its full memory space.

The second contribution (presented in Chapter 4) is more practical as it does not constraint the memory architecture of the target platform, nor requires timing information and shared periodicity of applications. Results of the liveness analysis have been shown to be implementable into existing run-time environments, but have not being tested with real-world systems such as 4G/5G base antenna.

The run-time environment (third contribution of the thesis) presented in Chapter 6 is a prototype and, as such, its performance has not yet been evaluated in comparison to other existing run-time environments nor with realistic applications.

7.3 Future Work and Perspective

There are multiple research prospects to further reduce the size of the automaton analyzed by liveness analysis of Chapter 4. A first option is to see how temporal information such as tasks' WCET, and earliest start and latest end times could be taken into account to reduce the state-space.

Both the deadlock prevention technique presented in Chapter 3 and the liveness analysis of Chapter 4 uses workload graphs. In the workload graph different configurations of a given application are represented as different applications. In this thesis we have not studied how to take into account that these different configurations of an application would not run in parallel, and by how much it would reduce the complexity of the analysis of systems with reconfigurable applications. Another impact of the workload graph lies with the representation of concurrent iterations of an application as different applications. If precedence constraints can be applied to tasks of different iterations, for example a task t of an iteration n should never start before the same task t of the previous iteration $n-1$ has started, adding them to the workload graph would reduce the complexity of its analysis.

A second common improvement for contributions 1 and 2 is the study of memory fragmentation. Both method have been designed by focusing on the memory size and making sure the total amount of free space is always sufficient. But when executing applications it is possible to reach a situation where this amount of free space is large enough, but split into multiple small-sized blocks such that no block is large enough for a buffer allocation. This is an issue as many hardware accelerators require their I/O buffers to occupy a contiguous space. There are at least two possible ways around this problem. The first is to provide some memory virtualization, similar to memory management units, in order to abstract away the physical memory space. The second, easier to implement, would be downsize the memory given to the analysis in comparison to the actual memory. This would require to study how large this downsizing should be so that the larger amount of memory guarantees that no allocation could be impossible at run-time due to fragmentation.

A third common improvement to our first and second contributions would be to continue on improving our implementations, by making use of multi-threading (for contribution 2), and by implementing them in C or another compiled language (for both contributions). The objective is to study the performance gain which could be made. Furthermore, reducing the memory footprint of our implementation is an important future work to allow the study of larger workloads.

The evaluation conducted in Chapter 5 can be extended by studying the impact the various approximate approaches have on the optimal make-span, and by studying larger workloads if improvements on our implementation of contribution 1 and 2 allow it. Also, other approaches such as SynDEx [27], PREESM [61], etc. could be added to the comparison.

The performance of the highly parallelized run-time environment presented in Chapter 6 should be evaluated in comparison to other run-time environments with different architectures. Furthermore, the case study should be extended to real-life applications sets, such as dataflow applications representing 4G and 5G processing in baseband receiving units.

There are also multiple implementation choices of the run-time environment whose impact on performance should be studied.
The first point blocking and unblocking events are not treated symmetrically by our implementation. Blocking events are taken into account before any scheduling de-

cision, as not doing so could lead the system into a deadlock of the decision is incompatible with a rule newly applied. On the other hand, unblocking event are not as important, as taking a decision assuming a job is still blocked can reduce performance, but not create a deadlock. This is why our implementation does not require to take them into account before taking them into account. Whether processing unblocking events before scheduling would lead to a significant improvement in performance remains to be studied.

The second point lies with the representation of deadlock prevention rules. Since their number increases exponentially on par with the exponential rise in complexity with the size of the system analyzed, their representation can be an important matter for large systems. Indeed these rules must be stored in memory (thus reducing the memory left for the execution of applications), and the deadlock prevention thread must be able to efficiently determine which rules apply or not at a given instant. Both the memory space taken up by rules, and the efficiency of the deadlock prevention thread are highly dependent on the representation of rules. As such different data structures should be studied, in order to determine their efficiency and select the most suitable for the target systems.

Part III

Back matter

Acronyms

πSDF Parametrized & Interfaced Synchronous Data Flow. 8, 27, 28, 33, 41, 45, 47

AHSDF Acyclic Homogeneous Synchronous Data Flow. 8, 25, 26, 33, 37, 39, 42, 45, 48, 56, 60–65, 69, 75, 77, 78, 114

API Application Programming Interface. 47, 114, 115, 122, 129

CAD Computer-Aided Design. 41, 83

CPU Central Processing Unit. 13, 15, 37, 46, 47, 55, 72, 86, 100, 108, 113, 122, 123

CSDF Cycle-Static Data Flow. 26, 45

DAG Directed Acyclic Graph. 40, 56

DMA Direct Memory Access. 16, 72, 119

DRAM Dynamic Random-Access Memory. 73

DSE Design Space Exploration. 41, 116

DSP Digital Signal Processing. 13, 15, 21, 41, 115

DSP Digital Signal Processor. 13, 46

FIFO First In, First Out. 24–27, 60, 63, 111, 127

FPGA Field Programmable Gate Array. 13, 15

GiB Gibibyte (1 GiB is $2^{30} = 1,073,741,824$ bytes). 86, 100, 104

GPU Graphics Processing Unit. 13, 15, 46, 47, 55, 72

GS^{3}PR Generalized Systems of Simple Sequential Processes with Resources. 31–33

HPC High-Performance Computing. 18, 42, 45, 46, 48, 59, 110, 111, 129

I/O Input/Output. 55, 56, 58, 134

IDE Integrated Development Environment. 116

JSON JavaScript Object Notation. 113–115

LCM Least Common Multiple. 26, 57

MAC Medium Access Control. 115

MEG Memory Exclusion Graph. 8, 20, 60–68, 98, 103, 131

MoC Model of Computation. 23, 24, 26–28, 33, 39, 48, 110

MPSoC Multi-Processor System on a Chip. 15–18, 24, 41, 109

NUMA Non-Uniform Memory Access. 13, 15, 16, 23, 37, 46, 48, 55, 71, 119

OS Operating System. 86, 100

PC Personal Computer. 59, 86

PC^{2}R Processes Competing for Conservative Resources. 33

PE Processing Element. 15, 16, 18–20, 34, 37, 39, 46, 47, 55, 58, 59, 63, 65, 71–75, 86, 96, 109–112, 114, 117, 119, 122, 123

RAM Random-Access Memory. 43, 86, 100, 104

RAS Resource Allocation System. 8, 10, 18, 19, 31–33, 36, 43, 45, 75, 77

RCPSP Resource Constrained Project Scheduling Problem. 34, 35

S*PR S*PR. 33

S^{3}PR Systems of Simple Sequential Processes with Resources. 31–33

S^{4}PR S^{4}PR. 31–33

SDF Synchronous Data Flow. 8, 24–28, 33, 37, 45, 47, 48, 60, 63

SDK Software Design Kit. 16, 17

SoC System on a Chip. 15

SRAM Static Random-Access Memory. 73

UMA Uniform Memory Access. 15, 16, 55, 71

WCET Worst-Case Execution Time. 58, 59, 63, 64, 133

Bibliography

[1] Data Plane Development Kit. https://www.dpdk.org/.

[2] Marvell OCTEON Fusion CNF95xx family - product brief. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-infrastructure-processors-octeon-fusion-cnf95xx-product-brief-2020-02.pdf.

[3] Open Radio Access Network Alliance. https://www.o-ran.org/software.

[4] SDF^3: SDF For Free. http://www.es.ele.tue.nl/sdf3.

[5] Intel SDK for OpenCL Applications. https://software.intel.com/en-us/intel-opencl, 2017.

[6] S. Adyanthaya, M. Geilen, T. Basten, R. Schiffelers, B. Theelen, and J. Voeten. Fast multiprocessor scheduling with fixed task binding of large scale industrial cyber physical systems. In *EUROMICRO DSD*, pages 979–988, 2013.

[7] Paul-Antoine Arras, Didier Fuin, Emmanuel Jeannot, and Samuel Thibault. DKPN: A Composite Dataflow/Kahn Process Networks Execution Model. In *24th Euromicro International Conference on Parallel, Distributed and Network-based processing*, Heraklion Crete, Greece, 2016.

[8] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[9] K. Barkaoui and I. Ben Abdallah. A deadlock prevention method for a class of FMS. In *1995 IEEE International Conference on Systems, Man and Cybernetics. Intelligent Systems for the 21st Century*, volume 5, pages 4119–4124 vol.5, 1995.

[10] Kamel Barkaoui and Laure Petrucci. Structural analysis of workflow nets with shared resources. In *Workflow Management: Net-based Concepts, Models, Techniques and Tools (WFM'98)*, pages 82–95, 1998.

[11] Gabriel Bathie, Loris Marchal, Yves Robert, and Samuel Thibault. Revisiting dynamic DAG scheduling under memory constraints for shared-memory platforms. In *IPDPS - 2020 - IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 1–10, New Orleans / Virtual, United States, 2020.

[12] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete. Cycle-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397–408, 1996.

[13] T. Blattner, W. Keyrouz, M. Halem, M. Brady, and S. S. Bhattacharyya. A hybrid task graph scheduler for high performance image processing workflows. In *2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*, pages 634–637, 2015.

[14] Timothy Blattner, Walid Keyrouz, Shuvra S. Bhattacharyya, Milton Halem, and Mary Brady. A hybrid task graph scheduler for high performance image processing workflows. *Journal of Signal Processing Systems*, 89(3):457–467, Dec 2017.

[15] J. Boutellier, J. Wu, H. Huttunen, and S. S. Bhattacharyya. PRUNE: Dynamic and decidable dataflow for signal processing on heterogeneous platforms. *IEEE Trans. on Sig. Proc.*, 66(3):654–665, 2018.

[16] 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*, Proceedings ACSD 2014, Hammamet, Tunisia, 2014.

[17] Xiaoliang Chen, Zhiwu Li, Naiqi Wu, Abdulrahman M. Al-Ahmari, Abdulaziz Mohammed El-Tamimi, and Emad S. Abouel Nasr. Confusion avoidance for discrete event systems by P/E constraints and supervisory control. *IMA Journal of Mathematical Control and Information*, 33(2):309–332, 10 2014.

[18] Yufeng Chen, Zhiwu Li, Abdulrahman Al-Ahmari, NaiQi Wu, and Ting Qu. Deadlock recovery for flexible manufacturing systems modeled with petri nets. *Information Sciences*, 381:290 – 303, 2017.

[19] E. G. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Comput. Surv.*, 3(2):67–78, 1971.

[20] Jack B. Dennis. First version of a data flow procedure language. In B. Robinet, editor, *Programming Symposium*, pages 362–376, Berlin, Heidelberg, 1974. Springer Berlin Heidelberg.

[21] K. Desnos, M. Pelcat, J.F. Nezan, and S. Aridhi. Memory analysis and optimized allocation of dataflow applications on shared-memory MPSoCs. *Jour. of Sig. Proc. Sys.*, pages 1–19, 2014.

[22] K. Desnos, M. Pelcat, J.F. Nezan, and S. Aridhi. On memory reuse between inputs and outputs of dataflow actors. *ACM Transactions on Embedded Computing Systems*, pages 30:1–30:25, 2016.

[23] K. Desnos, M. Pelcat, J.F. Nezan, S. Bhattacharyya, and S. Aridh. PiMM: Parameterized and interfaced dataflow meta-model for MPSoCs runtime reconfiguration. In *SAMOS*, pages 41–48, 2013.

[24] Gregory F. Diamos and Sudhakar Yalamanchili. Harmony: An execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th International Symposium on High Performance Distributed Computing*, HPDC '08, page 197–200. Association for Computing Machinery, 2008.

[25] Edsger W. Dijkstra. *Cooperating Sequential Processes*, pages 65–138. Springer New York, 2002.

[26] Andi Drebes, Antoniu Pop, Karine Heydemann, Nathalie Drach, and Albert Cohen. NUMA-aware scheduling and memory allocation for data-flow task-parallel applications. New York, NY, USA, 2016. Association for Computing Machinery.

[27] Y. Sorel et al. SynDEx. `http://www.syndex.org`. last visited on September 2018.

[28] J. Ezpeleta, J. M. Colom, and J. Martinez. A petri net based deadlock prevention policy for flexible manufacturing systems. *IEEE Transactions on Robotics and Automation*, 11(2):173–184, 1995.

[29] J. Ezpeleta, F. Tricas, F. Garcia-Valles, and J. M. Colom. A banker's solution for deadlock avoidance in FMS with flexible routing and multiresource states. *IEEE Trans. Robot. Autom.*, pages 621–625, 2002.

[30] A. Fanni, Alessandro Giua, and N. Sanna. Control and error recovery of petri net models with event observers. 01 1997.

[31] Fu-Shiung Hsieh and Shi-Chung Chang. Dispatching-driven deadlock avoidance controller synthesis for flexible manufacturing systems. *IEEE Transactions on Robotics and Automation*, 10(2):196–209, 1994.

[32] Thierry Gautier, Xavier Besseron, and Laurent Pigeon. KAAPI: A thread scheduling runtime system for data flow computations on cluster of multiprocessors. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, PASCO '07, page 15–23, New York, NY, USA, 2007. Association for Computing Machinery.

[33] Thierry Gautier, Joao Vicente Ferreira Lima, Nicolas Maillard, and Bruno Raffin. XKaapi: A runtime system for data-flow task programming on heterogeneous architectures. In *IPDPS*, 2013.

[34] Farhad Habibi, Farnaz Barzinpour, and Seyed Sadjadi. Resource-constrained project scheduling problem: review of past and recent developments. *Journal of Project Management*, 3:55–88, 01 2018.

[35] K. Hadj Salem, Y. Kieffer, and S. Mancini. Efficient algorithms for memory management in embedded vision systems. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–6, 2016.

[36] K. Hadj Salem, Y. Kieffer, and S. Mancini. Formulation and practical solution for the optimization of memory accesses in embedded vision systems. In *2016 Federated Conference on Computer Science and Information Systems (FedCSIS)*, pages 609–617, 2016.

[37] K. Hadj Salem, Y. Kieffer, and S. Mancini. Memory management in embedded vision systems: Optimization problems and solution methods. In *2016 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pages 200–207, 2016.

[38] Khadija Hadj Salem, Yann Kieffer, and Stéphane Mancini. *Meeting the Challenges of Optimized Memory Management in Embedded Vision Systems Using Operations Research*, pages 177–205. Springer International Publishing, Cham, 2018.

[39] J. Heulot, M. Pelcat, K. Desnos, J. Nezan, and S. Aridhi. SPIDER: A synchronous parameterized and interfaced dataflow-based RTOS for multicore DSPs. In *EDERC*, pages 167–171, 2014.

[40] Julien Heulot. *Runtime multicore scheduling techniques for dispatching parameterized signal and vision dataflow applications on heterogeneous MPSoCs*. PhD thesis, INSA de Rennes, 2015.

[41] Simon Holmbacka, Erwan Nogues, Maxime Pelcat, Sébastien Lafond, Daniel Menard, and Johan Lilius. Energy-awareness and performance management with parallel dataflow applications. *Journal of Signal Processing Systems*, 87(1):33–48, Apr 2017.

[42] Yifan Hou and Kamel Barkaoui. Deadlock analysis and control based on petri nets: A siphon approach review. *Advances in Mechanical Engineering*, 9(5):1687814017693542, 2017.

[43] M. V. Iordache, J. Moody, and P. J. Antsaklis. Synthesis of deadlock prevention supervisors using petri nets. *IEEE Transactions on Robotics and Automation*, 18(1):59–68, 2002.

[44] Laxmikant V. Kale and Sanjeev Krishnan. CHARM++: A portable concurrent object oriented system based on C++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, OOPSLA '93, page 91–108. Association for Computing Machinery, 1993.

[45] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, 1972.

[46] E. Kofman and R. de Simone. A formal approach to the mapping of tasks on an heterogenous multicore, energy-aware architecture. In *2016 ACM/IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE)*, pages 153–162, 2016.

[47] E. A. Lee and S. Ha. Scheduling strategies for multiprocessor real-time DSP. In *1989 IEEE Global Telecommunications Conference and Exhibition 'Communications Technology for the 1990s and Beyond'*, pages 1279–1283 vol.2, 1989.

[48] Edward Ashford Lee and David G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Trans. Comput.*, 36(1):24–35, 1987.

[49] John Lee. *Hardware/Software Deadlock Avoidance for Multiprocessor Multiresource System-on-a-Chip*. PhD thesis, 12 2004.

[50] J. V. F. Lima, T. Gautier, N. Maillard, and V. Danjean. Exploiting concurrent GPU operations for efficient work stealing on multi-GPUs. In *2012 IEEE 24th International Symposium on Computer Architecture and High Performance Computing*, pages 75–82, 2012.

[51] G. Liu, C. Jiang, and M. Zhou. Two simple deadlock prevention policies for S3PR based on key-resource/operation-place pairs. *IEEE Transactions on Automation Science and Engineering*, 7(4):945–957, 2010.

[52] Gaiyun Liu and Kamel Barkaoui. Necessary and sufficient liveness condition of GS 3 PR petri nets. *International Journal of Systems Science*, 46(7):1147–1160, 2015.

[53] GaiYun Liu and Kamel Barkaoui. A survey of siphons in petri nets. *Information Sciences*, 363:198 – 220, 2016.

[54] Juan-Pablo López-Grao and José-Manuel Colom. *Structural Methods for the Control of Discrete Event Dynamic Systems – The Case of the Resource Allocation Problem*, pages 257–278. Springer London, 2013.

[55] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, page 45–55. Association for Computing Machinery, 2009.

[56] Loris Marchal, Hanna Nagy, Bertrand Simon, and Frédéric Vivien. Parallel scheduling of DAGs under memory constraints. In *PDPS*, pages 204–213, 2018.

[57] Jean-Vivien Millo., Amine Oueslati., Emilien Kaufman., Julien DeAntoni., Frederic Mallet., and Robert de Simone. Explicit control of dataflow graphs with MARTE/CCSL. In *Proceedings of the 5th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD,*, pages 542–549. INSTICC, SciTePress, 2017.

[58] Hugo Miomandre, Julien Hascoët, Karol Desnos, Kevin Martin, Benoît Dupont De Dinechin, and Jean-François Nezan. Demonstrating the SPIDER runtime for reconfigurable dataflow graphs execution onto a DMA-based manycore processor. IEEE International Workshop on Signal Processing Systems, 2017. Poster.

[59] NXP. S32v2 processors for vision, machine learning, and sensor fusion. `https://www.nxp.com/docs/en/data-sheet/S32V234.pdf`, 2020.

[60] T. M. Parks, J. L. Pino, and E. A. Lee. A comparison of synchronous and cyclo-static dataflow. In *Conference Record of The Twenty-Ninth Asilomar Conference on Signals, Systems and Computers*, volume 1, pages 204–210 vol.1, 1995.

[61] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. F. Nezan, and S. Aridhi. PREESM: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming. In *EDERC*, pages 36–40, 2014.

[62] Jonathan Piat, Shuvra S. Bhattacharyya, and Mickaël Raulet. Interface-based hierarchy for synchronous data-flow graphs. In *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, pages 145–150, Tampere, Finland, 2009.

[63] S. A. Reveliotis, M. A. Lawley, and P. M. Ferreira. Polynomial-complexity deadlock avoidance policies for sequential resource allocation systems. *IEEE Trans. on Aut. Cont.*, 42(10):1344–1357, 1997.

[64] K. Rosvall and I. Sander. A constraint-based design space exploration framework for real-time applications on MPSoCs. In *DATE*, pages 1–6, 2014.

[65] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley Publishing, 9th edition, 2012.

[66] William Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall Press, USA, 6th edition, 2008.

[67] S. Stuijk, M. Geilen, and T. Basten. Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345, 2008.

[68] Fernando Tricas, F. García-Vallés, José Colom, and Joaquin Ezpeleta. *An Iterative Method for Deadlock Prevention in FMS*, pages 139–148. 08 2000.

[69] Yang Wang and Paul Lu. Dds: A deadlock detection-based scheduling algorithm for workflow computations in HPC systems with storage constraints. *Parallel Computing*, 39(8):291 – 305, 2013.

[70] J. Wu, T. Blattner, W. Keyrouz, and S. S. Bhattacharyya. Model-based dynamic scheduling for multicore implementation of image processing systems. In *2017 IEEE International Workshop on Signal Processing Systems (SiPS)*, pages 1–6, 2017.

[71] Xilinx. SDx development environment. https://www.xilinx.com/products/design-tools/\all-programmable-abstractions.html, 2017.

[72] Xilinx. Zynq UltraScale+ MPSoC. https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html, 2020.

[73] K. Xing, M. Zhou, H. Liu, and F. Tian. Optimal petri-net-based polynomial-complexity deadlock-avoidance policies for automated manufacturing systems. *IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans*, 39(1):188–199, 2009.

[74] Wei-Chang Yeh. Real-time deadlock detection and recovery for automated manufacturing systems. *International Journal of Advanced Manufacturing Technology*, 20:780–786, 01 2002.

[75] C. Zhong, Z. Li, Y. Chen, and A. Al-Ahmari. On nonexistence of a maximally permissive liveness-enforcing pure net supervisor. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 43(1):29–37, 2013.

ECOLE
DOCTORALE

Titre : Analyses de vivacité et environnement logiciel intelligent pour la gestion mémoire d'applications flux de données

Mots clés : co-conception materielle/logicielle, simulation, exploration d'achitecture, ingénierie des modèles, partitionnement materiel/logiciel

Résumé : Cette thèse a été effectuée à Télécom Paris et a été financé par Nokia Bell Labs France. Dans cette thèse sont étudiées différentes techniques visant à la gestion des interblocages et de la saturation des capacités mémoires dans les systèmes embarqués. Ce travail trouve sa motivation dans la complexification de l'architecture des systèmes informatiques au cours des dernières décennies, notamment avec la généralisation des architectures hétérogènes et Non-Uniform Memory Access (NUMA). Cette évolution se constate dans tous types de systèmes informatiques, de l'embarqué sur Multi-Processor System on a Chip (MPSoC) aux systèmes distribués pour le calcul haute performance (High-Performance Computing). Nous nous intéressons en particulier au problème de la saturation des capacités mémoires dans les systèmes embarqués utilisés pour le traitement numérique du signal (Digital Signal Processing). Nos contributions peuvent toutefois être utilisées pour d'autres types d'applications et de plateformes. Cette thèse apporte trois contributions : (1) Nous présentons une technique de prévention des interblocages se basant sur l'étude des cliques dans un type de graphes, les Memory Exclusion Graphs. Ces graphes représentent les buffers alloués en mémoire et leur possibilité d'allocation simultanée. (2) Nous présentons une optimisation de l'analyse de vivacité conventionnellement utilisée pour l'étude de la saturation mémoire, permettant d'analyser des systèmes plus complexes en un temps réduit. (3) Nous avons développé une technique d'évitement des interblocages utilisant les résultats de l'analyse de vivacité. Cette technique d'évitement a été intégrée à un environnement d'exécution expérimental. Nous évaluons la première et la deuxième contribution en les comparant à un outil issu de l'état de l'art. Pour conclure, nous proposons plusieurs pistes de travaux futurs sur la base des contributions de la thèse.

Title : A run-time environment for efficient management of dynamic application workloads of 5G network services

Keywords : Model Driven Engineering, Hardware/Software Partitioning, Hardware/Software Co-design, Simulation, Design Space Exploration

Abstract : This thesis has been realized at Télécom Paris and it has been financed by Nokia Bell Labs France. It studies different techniques to handle the issue of deadlocks and memory shortages in computing systems. Its work is motivated by the rise over the past decades of heterogeneous and Non-Uniform Memory Access (NUMA) architectures in all varieties computing systems, from embedded systems running on Multi-Processor System on a Chip (MPSoC) to distributed High Performance Computing (HPC) systems. We focus more specifically on the issue of memory shortages in embedded systems used for Digital Signal Processing, but our contributions could be applied to different applications and platforms. The contributions of this thesis are threefold: (1) we present a deadlock prevention technique based on the analysis of cliques in Memory Exclusion Graphs, which are graphs representing buffers allocated in memory and whether they might get simultaneously allocated; (2) we present an optimization on the conventional liveness analysis for memory shortages, allowing to execute the liveness analysis in reasonable time for larger systems than previously supported; (3) we developed a deadlock avoidance strategy using results from the liveness analysis, and integrated it into an experimental run-time environment. We evaluate our first and second contributions in comparison to an existing state-of-the-art tool. Finally we propose multiple leads to improve on the contributions of the thesis.

Institut Polytechnique de Paris
91120 Palaiseau, France