Runtime multicore scheduling techniques for dispatching parameterized signal and vision dataflow applications on heterogeneous MPSoCs

Julien Heulot

Résumé

Une tendance importante dans le domaine de l'embarqué est l'intégration de plus en plus d'éléments de calcul dans les systèmes multiprocesseurs sur puce (MPSoC). Cette tendance est due en partie aux limitations des puissances individuelles de ces éléments causées par des considérations de consommation d'énergie. Dans le même temps, en raison de leur sophistication croissante, les applications de traitement du signal ont des besoins en puissance de calcul de plus en plus dynamique. Dans la conception et le développement d'applications de traitement de signal multicoeur, l'un des principaux défis consiste à répartir efficacement les différentes tâches sur les éléments de calcul disponibles, tout en tenant compte des changements dynamiques des fonctionnalités de l'application et des ressources disponibles. Une utilisation inefficace peut conduire à une durée de traitement plus longue et/ou une consommation d'énergie plus élevée, ce qui fait de la répartition des tâches sur un système multicoeur une tâche difficile à résoudre.

Les modèles de calcul (MoC) flux de données sont communément utilisés dans la conception de systèmes de traitement du signal. Ils décomposent la fonctionnalité de l'application en acteurs qui communiquent exclusivement par l'intermédiaire de canaux. L'interconnexion des acteurs et des canaux de communication est modélisée et manipulée comme un graphe orienté, appelé un graphique de flux de données. Il existe différents MoCs de flux de données qui offrent différents compromis entre la prédictibilité et l'expressivité. Ces modèles de calculs sont communément utilisés dans la conception de systèmes de traitement du signal en raison de leur analysabilité et leur expressivité naturelle du parallélisme de l'application.

Dans cette thèse, une nouvelle méthode de répartition de tâches est proposée afin de répondre au défi que propose la programmation multicoeur. Cette méthode de répartition de tâches prend ses décisions en temps réel afin d'optimiser le temps d'exécution global de l'application. Les applications sont décrites en utilisant le modèle paramétrée et interfacé flux de données (PiSDF). Ce modèle permet de décrire une application paramétrée en autorisant des changements dans ses besoins en ressources de calcul lors de l'exécution. A chaque exécution, le modèle de flux de données paramétré est déroulé en un modèle intermédiaire faisant apparaitre toute les tâches de l'application ainsi que leurs dépendances. Ce modèle est ensuite utilisé pour répartir efficacement les tâches de l'application. La méthode proposé a été testée et validé sur plusieurs applications des domaines de la vision par ordinateur, du traitement du signal et du multimédia.

Abstract

An important trend in embedded processing is the integration of increasingly more processing elements into Multiprocessor Systems-on-Chip (MPSoC). This trend is due in part to limitations in processing power of individual elements that are caused by power consumption considerations. At the same time, signal processing applications are becoming increasingly dynamic in terms of their hardware resource requirements due to the growing sophistication of algorithms to reach higher levels of performance. In design and implementation of multicore signal processing systems, one of the main challenges is to dispatch computational tasks efficiently onto the available processing elements while taking into account dynamic changes in application functionality and resource requirements. An inefficient use can lead to longer processing times and higher energy consumption, making multicore task scheduling a very difficult problem to solve.

Dataflow process network Models of Computation (MoCs) are widely used in design of signal processing systems. It decomposes application functionality into actors that communicate data exclusively through channels. The interconnection of actors and communication channels is modeled and manipulated as a directed graph, called a dataflow graph. There are different dataflow MoCs which offer different trade-off between predictability and expressiveness. These MoCs are widely used in design of signal processing systems due to their analyzability and their natural parallel expressivity.

In this thesis, we propose a novel scheduling method to address multicore scheduling challenge. This scheduling method determines scheduling decisions strategically at runtime to optimize the overall execution time of applications onto heterogeneous multicore processing resources. Applications are described using the Parameterized and Interfaced Synchronous DataFlow (PiSDF) MoC. The PiSDF model allows describing parameterized application, making possible changes in application's resource requirement at runtime. At each execution, the parameterized dataflow is then transformed into a locally static one used to efficiently schedule the application with an a priori knowledge of its behavior. The proposed scheduling method have been tested and benchmarked on multiple state-of-the-art applications from computer vision, signal processing and multimedia domains.

N° d'ordre : 15ISAR 27 / D15 - 27

Institut National des Sciences Appliquées de Rennes

20, Avenue des Buttes de Coësmes • CS 70839 • F-35708 Rennes Cedex 7

Tel : 02 23 23 82 00 - Fax : 02 23 23 83 96

Thèse

2015

Julien HEULOT

THESE INSA Rennes

sous le sceau de l'Université européenne de Bretagne pour obtenir le titre de

DOCTEUR DE L'INSA DE RENNES

Spécialité : Traitement du Signal et de l'Image

présentée par

Julien Heulot

ECOLE DOCTORALE : *MATISSE*

LABORATOIRE : *IETR*

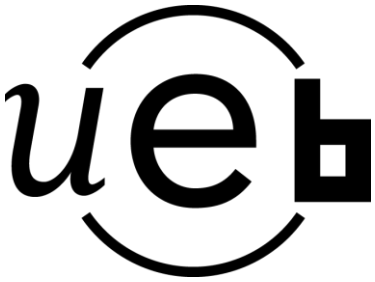
Runtime Multicore Scheduling Techniques for Dispatching Parameterized Signal and Vision Dataflow Applications on Heterogeneous MPSoCs

Thèse soutenue le 24.11.2015
devant le jury composé de :

Alix MUNIER-KORDON
Professeur au LIP6 Paris / Présidente
Gilles SASSATELLI
Directeur de Recherche au LIRMM, Montpellier / Rapporteur
Alain GIRAULT
Directeur de Recherche à l'INRIA Grenoble / Rapporteur
Johan LILIUS
Professeur à l'Abo Akademi (Finlande) / Examinateur
John McALLISTER
Maitre de Conférence au Queen's University of Belfast (UK) / Examinateur
Slaheddine ARIDHI
Docteur, Texas Instruments France / Encadrant
Maxime PELCAT
Maitre de Conférence à INSA Rennes / Encadrant
Jean-François NEZAN
Professeur à l'INSA Rennes / Directeur de thèse

Runtime Multicore Scheduling Techniques for Dispatching Parameterized Signal and Vision Dataflow Applications on Heterogeneous MPSoCs

Julien Heulot



Contents

Acknowledgements

I would like to thank my advisors Dr. Slaheddine Aridhi, Dr. Maxime Pelcat and Pr. Jean-Francois Nezan for their help and support during these three years. Slah, thank you for welcoming me at Texas Instruments in Cagnes-Sur-Mer and for giving me enlightened technical advices and discussions we had during these three years. Maxime, thank you for the very open-minded brainstorms and for all the discussions we had on technical and redactional details. Jeff, thank you for your support and all your wise advice during these three years, and thank you for your ability to take a step back on my work.

I want to thank Pr. Gilles Sassatelli and Pr. Alain Girault for reviewing this thesis. Thanks also to Pr. Alix Munier-Kordon for presiding the jury, to Pr Johan Lilius and Dr. John McAllister for being members of the jury.

It has been a pleasure to work with Karol Desnos, Yaset Oliva, Judicaël Menant and Clément Guy. Thank you for your friendship and the pleasant moments we spent working together. Thanks also to the IETR Image/Impact team for being such great co-workers, going to the lab was always a pleasure. Also, many thanks to Jocelyne Tremier, Corinne Calo and Aurore Gouin for their administrative support.

This thesis also benefited from many discussions with the High Performance Embedded Processing team from Texas Instruments France: special thanks to Eric Biscondi, Louis-Paul Cordier, Filip Moerman and Sebastien Tomas.

I am also grateful to Dr. Sébastien Lafond, Johan Ersfolk, Simon Holmbacka and their team for welcoming me at the Åbo Akademi in Finland and for the interesting talks we had.

Thanks a lot to Slah & Karina Aridhi for the numerous corrections, advices and hours spent on this entire document.

Thanks to my friends, my parents and sister for the support they gave me and most specially during the redaction of this manuscript. Thanks also to Edgar for cheering me up with your hairy hugs.

Lastly, but far from least important, I would like to thank Justine for the love she gave me since our roads have crossed. Thanks for letting me go over unharmed the painful redaction of this manuscript. Thank you for your support and I will be there for you when you will have yours!

CHAPTER 1

Introduction

1.1 General Context

The recent evolution of *embedded systems* have resulted in the most remarkable technological advances. From the highly complex, cutting edge, engine controller of an FIA Formula One car to a basic pedometer that counts each step of a pedestrian, embedded systems can be found almost everywhere in the society of the early 21th century. A human heart, the most vital organ, may now be replaced with a machine [Car]. An embedded system is an integrated electronic and computing system designed for a specific purpose. Mobile phones, tablets, contactless credit cards, internet-connected watches and even domestic drones are just some of the innumerable modern devices containing one or more embedded systems.

1.1.1 Embedded Systems Constraints

A major specificity of the embedded systems design process is that the development is constrained by one or more precise requirements. These constraints can stem from various sources.

Firstly, certain constraints result directly from the application. The reaction time of an airbag in a car, the decoding rate of a music player or even the power consumption of a space probe are examples of constraints that may apply to an embedded system.

Constraints are also introduced by the economic environment. Spending years of development, to embed the most expensive cutting edge electronic devices which require frequent software updates may not be necessary for a garage door opener.

Finally, the physical environment may introduce further constraints for an embedded device. These constraints are not related to the application but must be respected in order to execute in a sustainable and safe manner. The high pressure experienced by a deep ocean device or the faults caused by solar radiation received by space probe are examples of environmental constraints.

These diverse constraints are often contradictory, or at least hard to satisfy simultaneously. As a consequence, there are trade-offs during the embedded device design process. For example, reducing the operating frequency of a chip may reduce its performance but it will also lower its power consumption.

1.1.2 Designing an Embedded Systems

The design process for an embedded device is often divided into two parts: hardware and software design.

The primary objective of hardware design is to adapt the computing resources of the system to the application and environment. Sensors, actuators or user interfaces are some of the many features that may be necessary in an embedded system. The hardware portion also provides services required by the computing device for system operation. Clock generation, power supply and (large) external memory are examples of services that the hardware provides to the software. The major objective of the hardware development is to integrate these components and services so that they operate as a system; this is often referred to as the hardware architecture of the system.

The computing resources of a system are designated as Processing Elements (PEs). A PE is a programmable device that can perform a function for the system. PEs can belong to different categories: from low-power micro-controllers to highly computational intensive General Purpose Processors (GPPs) and/or Digital Signal Processors (DSPs). Currently, embedded systems may integrate many PEs thus creating a Multiprocessor System-on-Chip (MPSoC). If the PEs are of different types, the system is a heterogeneous MPSoC (GPPs, DSPs and application specific accelerators).

The complement of hardware is the program running over the different PEs and is referred as the *software* portion of the embedded system. A software program is a list of instructions that are executed by a PE. These instructions are stored in a dedicated memory and are processed sequentially by each PE. These instructions are written using a machine language that may differ for each PE.

Currently, software programmers generally do not program PEs using assembly language but use instead a higher level language. Java, Fortran, C/C++ and their derivatives are all examples of common higher level programming languages. These higher level languages are then converted to machine language using dedicated, PE-dependent compilers.

The development of each of the hardware and software portions are tightly coupled and a choice made in one of the two portions often imposes design choices on the other.

1.1.3 Embedded Parallel Systems

In 1965, Moore predicted that the number of transistors in an integrated circuit would double every two years [Moo65]. Despite the fact that the present evolution is slowing down, this prediction has held true since it was made, making computing devices increasingly more complex with the passage of time.

For many years, computer systems have been subject of the so-called megahertz myth [meg]. This is the idea that a faster clock rate produces faster computing task execution. This myth was maintained by chip manufacturer for decades until the power dissipation needs of chips increased dramatically. Thus, starting in the first decade of the 21st century, the integration of higher numbers of PEs into MPSoC became the new solution to continuously increase the processing power of devices.

Currently, a new research field is developing a dramatic importance: parallel programming techniques. Since increasingly more PEs are integrating in systems, it has become vital to invent methods to dispatch computation and synchronize PEs automatically. New programming techniques have emerged to allow programmers to handle multiple PE systems more easily. One of these is dataflow programming.

1.1.4 Dataflow Programming

Dataflow Models of Computation (MoCs) have been designed to represent the parallelism of applications. Applications are described using computational blocks which exchange data using directed communication channels. A major advantage of dataflow MoCs is the possibility of using legacy code to express the behavior of computational blocks. The first dataflow MoC in literature was introduced by Kahn [Kah74].

Since then, many dataflow MoCs have emerged in literature that extend, assure the behavior of certain properties or add features to the original model. Some of these models target the possibility that MoCs could include dynamic changes in the application description. These dynamic changes provide mechanisms to add or remove computational tasks at runtime. Since the MoC is capable of detecting these changes, this permit better decisions to be made for multiprocessor programming.

1.2 Contributions of the Thesis

This thesis studies programming techniques for embedded multicore devices. The starting point is a dataflow model, used to represent applications and to efficiently dispatch them onto embedded multicore devices. The dataflow MoC employed is called Parameterized and Interfaced Synchronous Dataflow (SDF) (PiSDF) [DPN+13].

This dataflow MoC provides reconfigurable features allowing the adaptation of an application to modify parameters. These parameters trigger dynamic changes in the application execution structure that are processed at runtime.

Within this framework, a novel runtime software has been developed that performs the efficient scheduling of PiSDF applications onto embedded multicore devices. The contributions of this thesis are listed below:

- The introduction of a novel multicore scheduling method to unveil parallelism of a PiSDF application and to dispatch different tasks onto the targeted platform. This method is called Just-In-Time Multicore Scheduling (JIT-MS).
- Certain improvements on the previously defined JIT-MS method have been introduced. These enhancements comprise a PiSDF MoC extension, optimizations of the intermediate graph used in several multicore scheduling methods and a novel dispatch algorithm employed for massively multicore devices.
- A novel runtime software called Synchronous Parameterized Interfaced Dataflow Embedded Runtime (Spider) is introduced. Spider performs the JIT-MS method at runtime and exploits the parallelism of PiSDF applications. Spider targets heterogeneous multicore platforms and can be easily adapted to several platforms.
- A case study of two state-of-the-art computer-vision and signal processing applications on a physical multicore heterogeneous device is presented. This case study uses Spider and also discusses an application representation in PiSDF.

All contributions have been developed as part of a scientific collaboration between the IETR, and Texas Instrument France, and within the ANR COMPA project.

1.3 Outline

This thesis is organized in two parts: Part I presents the background and concepts studied in this thesis, and Part II introduces and evaluates the contributions of the thesis.

In Part I, Chapter 2 details the embedded parallel platform landscape. Then Chapter 3 defines the concept of a dataflow MoC and presents the PiSDF dataflow MoC that is the starting point of this thesis.

In Part II, the JIT-MS method is presented in Chapter 4. Chapter 5 then explores methods to improve the JIT-MS method. In Chapter 6, a PiSDF-based runtime called Spider is introduced; this runtime embeds the JIT-MS method. Chapter 7 proposes a benchmark of JIT-MS and Spider, in addition to a case study of two state-of-the-art computer-vision and signal processing applications.

Finally, Chapter 8 concludes this thesis and proposes potential future research directions.

Part I

Background

CHAPTER 2

Embedded Parallel Platforms

2.1 Introduction

As discussed in the previous chapter, an embedded system is a computer system with a dedicated function within a larger mechanical or electrical system, often with real-time computing constraints [Emb]. The first modern embedded system is often regarded as the Apollo Guidance Computer used in the NASA Apollo missions. This system was embedded in command and lunar modules of the Apollo program. It was used for guidance, navigation and control of modules. This embedded computer was designed during the 1960's by Charles Stark Draper from the MIT Instrumentation Laboratory.

Embedded systems are now widely spread in common items used daily. And this trend will undoubtedly increase in the coming decades. According the Gartner 2014 Hype Cycle [RVdM14], the Internet of Things (IoT) will become an important technology in the next decade. The predictions for the future of IoT consists of many devices per person, each connecting to Internet. These devices will be small embedded systems included in a larger system, which may itself be embedded in a larger system.

However, due to the physical limitations of current semiconductor technology frequency improvements in processors are finite. For the rise of computational power of computer systems to continue, hardware designers must integrate more Processing Elements (PEs) into increasingly complex designs. New programming tools, compilers and frameworks are also released to design efficiently these increasingly complex systems. These new programs are necessary, but software developers then have the additional challenge to learn the new programming environments.

In this chapter, the overview of embedded systems will be described in Section 2.2. Currently available programming techniques will be subsequently in Section 2.3. The limitations of these programming techniques will be highlighted, to illustrate the need for new programming approaches.

2.2 Parallel Embedded System architecture

2.2.1 What is an parallel embedded system ?

According to Steve Heath in [Hea02], an embedded system is a microprocessor-based system designed to control one or more predefined functions. Unlike personal computers which are designed for a wide range of applications, an embedded system is designed for a very specific application. Embedded systems usually have several constraints to satisfy, such as high computing performance, low power and hard real-time deadlines.

A parallel embedded platform is generally characterized by the several computing elements runnning simultaneously and also by the way in which they are connected to each other.

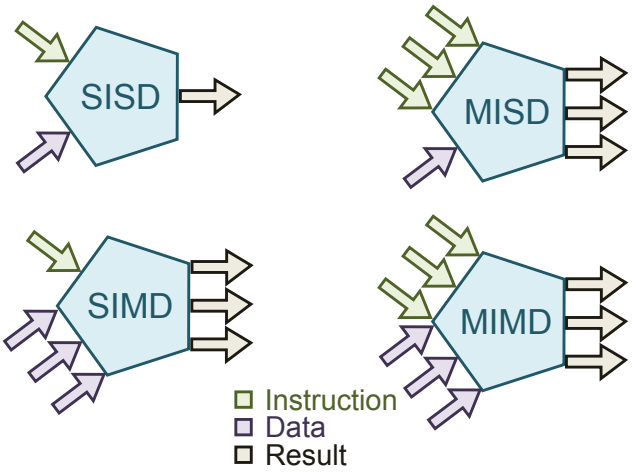


Figure 2.1 – *The four categories of Flynn's taxonomy*

These elements can be tightly coupled together or not tightly coupled. Flynn described a taxonomy to categorize computing systems in [Fly72]. He classifies systems into four categories by their instruction and data streams; these categories are represented in Figure 2.1:

- **Single Instruction, Single Data (SISD)** represents conventional sequential platforms. A single stream of instructions is applied to a single stream of data resulting in one stream of results.

- **Single Instruction, Multiple Data (SIMD)** represents platforms that can execute the same instruction stream on multiple data streams. SIMD instructions are used increasingly in computationally intensive and repetitive domains. The majority of modern Digital Signal Processors (DSPs) and Graphics Processing Units (GPUs) are defined by this feature.

- **Multiple Instructions, Single Data (MISD)** represents platforms that simultaneously execute different instruction streams on the same data stream. This computer architecture is not common. Systolic arrays described by Kung et al. [KL79] are among the few systems using this pattern. Matrix multiplication is one of the suitable applications of these systems cited by Kung et al.

- **Multiple Instructions, Multiple Data (MIMD)** represents the majority of the multicore platforms. Multiple instruction streams are applied on distinct data streams simultaneously. This includes superscalar cores.

By definition, the last 3 categories are classified as parallel systems. New designs called Multiprocessor Systems-on-Chips (MPSoCs) can now include several subsystems of different categories, thus increasing the architecture complexity. MPSoCs are by definition MIMD. They can embed multiple processing cores, each of which may be SIMD or MIMD. Platforms become parallel at different architecture levels making programmation of parallel systems even more complex.

Kalb has defined Autonomous SIMD (ASIMD) as an extension of SIMD platforms[Kal91]. ASIMD platforms grant a certain form of autonomy to the PEs of a platform, allowing them to support non-tightly synchronous operations. A different level of autonomy can be granted to PEs: execution, addressing, connection or I/O autonomies. PEs may be entitled to one or more of the following, depending upon which level was granted: to conditionally execute certain instructions (execution), to freely fetch input data (addressing), to allow a non-even communication pattern between them (connection) and to also unevenly access I/O ports.

2.2.2 Processing Element Types and Platform Heterogeneity

As described in the previous section, parallel systems may be characterized by multiple PEs of different types operating synchronously.

The range of possible PEs is wide: from very specialized co-processors to fully programmable processors. Programmable PEs, also called *cores*, are characterized by their Instruction Set Architecture (ISA) which lists supported instructions.

Processors are usually separated into two categories, as a function of the size of their ISA. Whereas Complex Instruction Set Computer (CISC) processors have a wider ISA leading to a smaller generated code, the ISA of Reduced Instruction Set Computer (RISC) processors is limited. The emergence of heterogeneous and hybrid processors has made categorization more complicated. Hybrid processors can be designed for a specific application, such as Application-Specific Instruction-Set Processors (ASIPs) or for an application domain such as DSPs or GPUs.

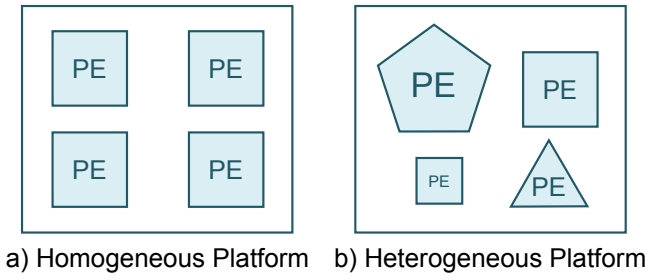


Figure 2.2 – *Heterogeneousness of Platforms*

An embedded MPSoC will often contain all these different core types. For example, the Texas Instruments Keystone II[66A13] architecture is composed of up to 8 DSP cores, 4 General Purpose Processor (GPP) ARM cores and some hardware co-processors (Network, Fourier transform, and so on). This platform is categorized as heterogeneous it contains embedded PEs of several types.

2.2.3 Memory architecture of embedded systems

Parallel embedded platforms are characterized by both memory organization and hierarchy. From these two parameters, the memory architecture can be defined.

2.2.3.1 Memory Organization: Shared or Distributed

A shared memory system is the most simple: all PEs may access all memory regions. However, other schemes of memory architecture exist. In the literature, shared and distributed memory systems are commonly separated into three memory models, as displayed in Figure 2.3.

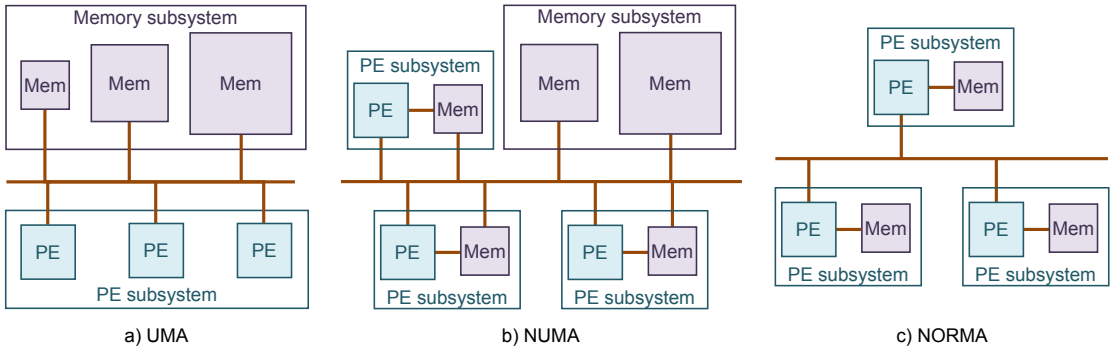


Figure 2.3 – *Memory Architecture*

Figure 2.3-a illustrates the shared memory system of an **Uniform Memory Access (UMA)** machine with multiple memory banks. Each bank can be accessed by every PEs. Both reliability and access speed are identical for each PE. UMA systems are often implemented using a bus (PEs share connection to each memory) or a crossbar network (each PE has its private connection to each memory). With increasing number of PEs, the UMA generally experiences bottlenecks (for the bus implementation) or requires complex memory subsystems (for the crossbar implementation). UMA systems are thus not suitable for massively parallel platforms.

A **Non Uniform Memory Access (NUMA)** machine provides multiple memory banks where each bank can also be accessed by every PEs, as shown in Figure 2.3-b. However, the access speed is dictated by the accessing PEs. This memory architecture is used to provide a dedicated access at a memory bank to a specific PE. When the number of PEs increases, this memory architecture restricts the memory subsystem growth.

To reduce the impact of the increasing number of PEs distributed memory systems are used. They are called **NO Remote Memory Access (NORMA)** machines. As is indicated in Figure 2.3-c, PEs memory access is limited to local memory banks. Communication for data exchange with other PEs is possible through dedicated hardware. Interconnections between cores can be implemented using Network on Chip (NoC) which improves communication performance and scalability but also increases programming complexity. NORMA memory systems have very good scalability properties but their complexity brings new challenges in MPSoC programming.

2.2.3.2 Memory Hierarchy

Equally as important as the memory organization is the memory system hierarchy. Despite the fact that small memories are designed with high speed access, big memories are still

required for applications like video or image processing. Thus, depending of the targeted application, the required memory hierarchy may differ.

In the literature, two schemes of memory hierarchy are commonly employed. These two schemes are represented in Figure 2.4.

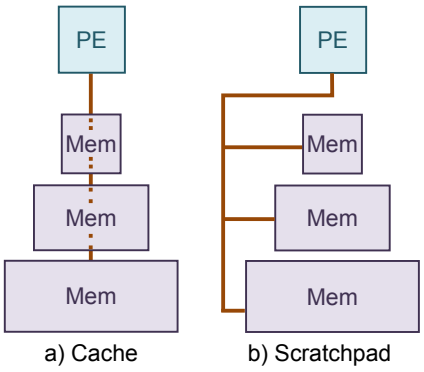


Figure 2.4 – *Memory Hierarchy*

Cache systems use the small, fast memory to store a replica of the data stored in a far memory. This memory scheme allows the programmer to obtain good performance without a complex program, as many cache hardware subsystems automatically handle this feature. However, the data integrity in multicore systems (also)called cache coherence) may pose problems in such a memory scheme. Hardware cache coherency subsystems are starting to appear in recent embedded systems. However, it is not a widely spread feature in the embedded landscape because it is resource and power consuming.

Scratchpads allow different memory banks to be accessed separately. This memory scheme allows the programmer better control of data thus optimizing its use. Another advantage is that scratchpad access times are predictable, whereas cache systems have nondeterministic access times [Ste90]. It must be noted that this memory scheme requires a greater understanding of the platform used and may introduce non-portable code over multiple platforms.

2.3 Parallel programming methods

This section focuses on general concepts in programming parallel embedded systems. Common programming styles are firstly introduced in Section 2.3.1. Then, the concept of granularity of parallelism is described in Section 2.3.2. Next, certain sources of parallelism are detailed in Section 2.3.3. Finally, methods to extract the parallelism from an application are presented in Section 2.3.4.

2.3.1 Programming Styles

There are many ways to program parallel embedded platforms. Two programming styles have been defined by Blank [BN92]: Single Program, Multiple Data (SPMD) and Multiple Programs, Multiple Data (MPMD). In SPMD, the same program is executed on each processing core of a parallel architecture. However, each processing core is fed with a distinct data stream, leading to an unique context and an unique execution. As shown in the portion of Figure 2.5, in SPMD, control units are duplicated for each processing cores

allowing them to run the same program differently, depending on the data fetched and branching conditions.

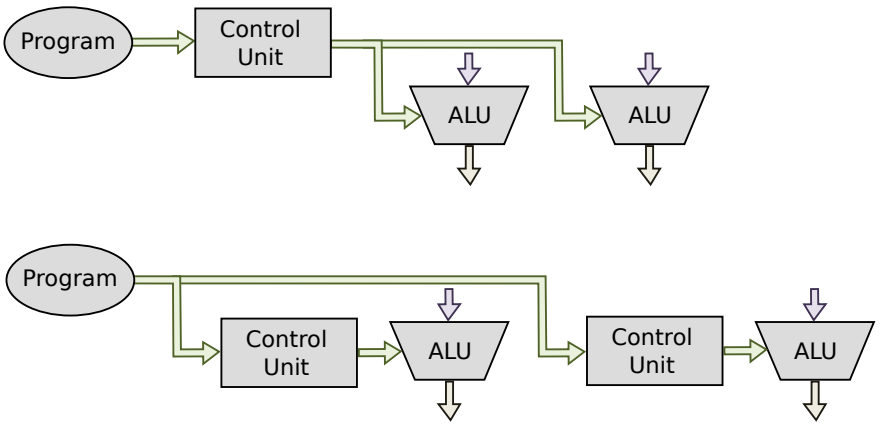


Figure 2.5 – *SIMD versus SPMD*

This can be contrasted with the MPMD style, which is illustrated in the lower portion of Figure 2.5. A MPMD programming model distributes a different program on each of the processing cores of parallel architectures. This approach allows the programmer to better optimize code and so achieve superior performance. The drawback to this style of that there is an increase of the programming complexity for architectures with numerous PEs.

An important trend in embedded processing is the integration of increasingly more PEs into embedded MPSoC devices [BFFM12]. In particular, there is an emergence of massive many-core platforms based on tiny RISC cores [MPP15, Epi15]. This trend is due, in part, to limitations of the processing power of individual PEs, resulting from power consumption considerations. Assuming that this direction continues, the SPMD programming model would be the best choice for future generations of embedded platforms.

2.3.2 Granularity of Parallelism

Parallelism in a program can be performed at different levels of granularity. In the literature, three levels of granularity of parallelism are usually described: fine grained, middle grained and coarse grained.

Fine grained parallelism is commonly defined as Instruction-Level Parallelism (ILP) and is a well known topic in literature. Currently, multiple embedded processors propose ILP features. Some DSPs, such as TI's Keystone architectures [SPR14], embed SIMD instructions that execute the same instruction on multiple data streams simultaneously. Additionally, multiple nonidentical instructions can be executed in parallel, this is called Very Long Instruction Word (VLIW). ILP can be extracted directly by compilers from a sequential code to fully exploit the parallelism offered by the DSP at the instruction level. Instruction sets focusing on ILP are becoming standard in common architectures. For the x86 architectures of conventional computers, MultiMedia eXtensions (MMX) and Streaming SIMD Extensions (SSE) are ones of the references. For embedded systems, NEON is the best known SIMD extension for ARM processors. These instruction sets are generally used with C/C++ code. ILP has now reached its limit in terms of optimization and a new parallelism source is required for further efficiency improvements of embedded MPSoCs.

Middle grained parallelism occurs when the execution of small pieces of code are tightly coupled together. In the literature, this granularity of parallelism is generally associated with the thread programming model. Threads are commonly described as sequential processes sharing some memory. It is common for a Symmetric Multi-Processor system (SMP) to have a hardware implementation of the thread model. Hardware requirements are, but are not limited to, cache coherency mechanisms and synchronization mechanism. Threads were firstly developed to execute multiple parts of a program *concurrently* on a sequential core and to thus emulate some parallelism. The thread model is the most commonly used multicore programming model. Since middle grained parallelism is at higher level than the ILP, the parallel portions of the program dominate the control and synchronization portions and so provide a good balance for parallelism extraction. However, multi-threading programming models are not suitable for the growing demand of parallel computing [Lee06]. Lee has demonstrated that the non-determinism involved in multi-threading programming models means that these models are not suitable to use the hardware at its maximum efficiency. A lack of suitability of threads for parallel computing produces new challenges in middle grained parallelism.

Coarse grained parallelism exists when multiple processes are simultaneously launched on a platform. These processes are only loosely linked to each other and do not communicate intensively. This means that this granularity of parallelism can be easily produced by software programmers using few shared resources. The different tasks can be assigned to the core manually or by a Real-Time Operating System (RTOS) which controls the synchronization and load balancing.

For the majority of this thesis, middle grained parallelism will be used to express the parallelism of an application. It is felt that this approach provides a good trade-off between potential parallelism and synchronization overhead.

2.3.3 Parallelism sources

From any given algorithm, multiples parallelism sources can be derived with a middle granularity level. In generally, there are three sources: *Task level parallelism*, *Data level parallelism* and *Pipeline level parallelism*. Figure 2.6 illustrates these three sources of parallelism.

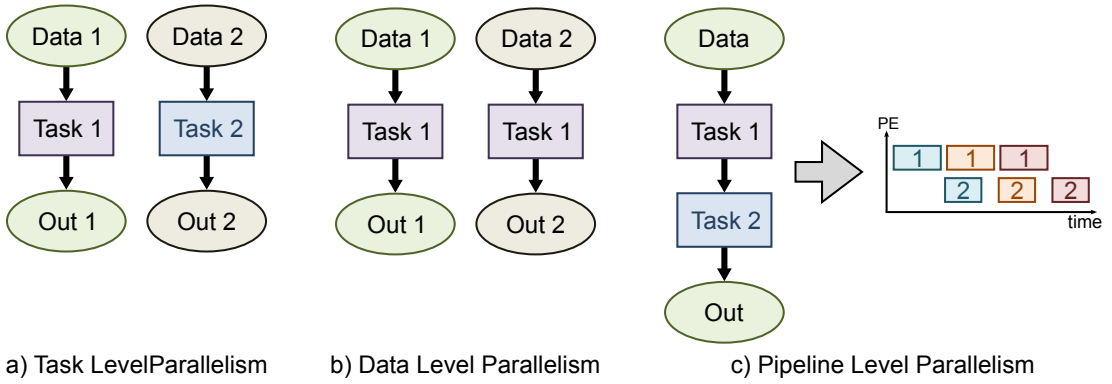


Figure 2.6 – *Middle grain parallelism sources of parallelism*

Task level parallelism occurs when two distinct tasks are applied on separate input data sets. This is usually easily extracted from applications.

Data level parallelism occurs when the same task is applied on two different input data sets. Data level parallelism is only possible when there is no dependency between the two data sets. In this way, the two tasks may be executed in parallel without problem.

Pipeline level parallelism occurs when two PEs of the same MPSoC simultaneously compute tasks on data from different iterations of an algorithm. This parallelism is often used in signal processing systems to increase the potential parallelism of the application. This then increases the PE load and the system throughput. A known drawback of this method is higher latency of the overall system; that is, longer computation time for a single algorithm iteration.

2.3.4 Parallel Programming techniques

2.3.4.1 Multicore scheduling

To extract middle grained parallelism on an application, the technique used is commonly divided into four phases: *extraction*, *mapping*, *ordering* and *timing*. These four phases of the multicore scheduling process are represented in Figure 2.7.

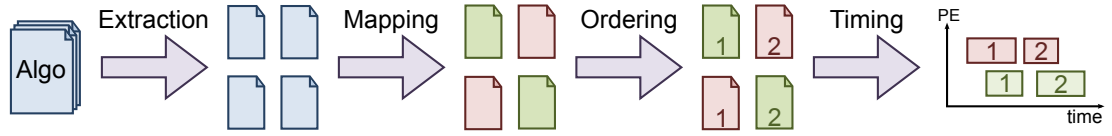


Figure 2.7 – *Middle grain parallelism multicore scheduling on a two PEs (red and green) platform*

First, parallel threads or tasks are extracted from the main program. For a deeper extraction, more threads will be created, thus unveiling more parallelism. However, when more thread are created, more time is required to handle the threads and mutual synchronization.

Next the mapping phase assigns each thread to a PE. This phase must manage various parameters such as PE utilization and communication cost. The subsequent phase of *ordering*, is sometimes grouped with the mapping phase. The ordering phase creates an execution list of threads on each PE.

Finally, the timing phase generates the choice of a start time for each thread. This task is commonly determined by a RTOS, and the choice is based on synchronization elements; that is, as soon as input data is available.

Lee et al. have defined the result of executing these phases at either compile-time or runtime by four multicore strategies[LH89], as is displayed in Table 2.1.

Table 2.1 – *Multicore Scheduling Strategies*

Strategy	Mapping	Ordering	Timing
fully dynamic	runtime	runtime	runtime
static-assignement	compile-time	runtime	runtime
self-timed	compile-time	compile-time	runtime
fully static	compile-time	compile-time	compile-time

For the case where multicore scheduling processing (mapping, ordering and timing phases) is performed at compile time, the overhead due to scheduling process at runtime is reduced. This is the fully static case. In contrast, when the multicore scheduling process

is completed at runtime, the process becomes more adaptable to current situation. This situation is the fully dynamic case and will be used in this thesis.

2.3.4.2 Scheduling approaches

There are a number of methods to unveil parallelism of an application. In [POH09], Park et al. compare many approaches of MPSoC design. These approaches are characterized by their inputs and can be described as follows:

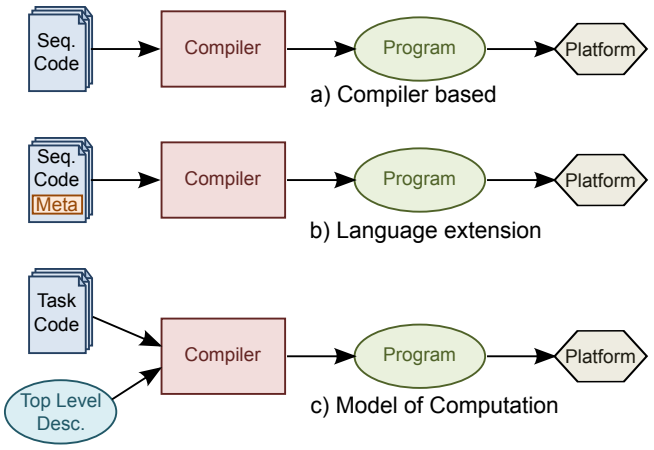


Figure 2.8 – *Middle grained parallelism multicore scheduling on a two PEs (red and green) platform*

- **The compiler based** approach which uses a sequential language such as C or C++. The compiler is designed to analyze the input program to extract its parallelism, and is illustrated at Figure 2.8-a. A broadly equivalent method is employed for ILP for finest grain extractions. The major advantage of this method is that no modification are required for the use of legacy sequential code. Code analysis tools have to separate code into multiple threads. However, this code analysis is not straightforward as created threads must be sufficiently coarse-grained to overcome the synchronization overhead. Conversely, threads must also to be sufficiently fine-grained to unveil the necessary parallelism. An example of compiler based tools is MPSoC Application Programming Studio (MAPS) [LC10] which was largely developed by RWTH Aachen University. This tool extracts parallelism from C sequential code and can generate code for embedded platforms [AJLA14].

- **The language extension** approach is similarly based on sequential code, however metadata is also used to highlight parallel regions of the code as illustrated in Figure 2.8-b. A well known example of this approach is the OpenMP (Open Multi-Processing [Ope15a]) Application Programming Interface (API). OpenMP supports shared memory architecture and generally targets homogeneous platforms. Based on C, C++ or Fortran code, this API uses compiler directives such as pragmas to define parallel regions and to provide a multicore communication API. The software developer must manually specify parallel regions in his code. OpenMP is now widely supported on embedded MPSoC platforms, such as TI's keystone architecture. Other tools or APIs based on language extensions are available, such as OpenMPI (Open

Message Passing library [Ope15b]), OpenCL (Open Computing Language [SGS10]) and CUDA (Compute Unified Device Architecture [cud14]).

- **The Model of Computation (MoC)** approach is based on a high-level description of the application abstracting out low-level details as illustrated in Figure 2.8-c. A MoC describes an application as several tasks with organized communications between them. This approach is labour-intensive for the software developer, as the top application that uses the MoC must be described, to allow the integration of the legacy code. However, the MoC approach has multiple advantages. First, the parallelism extraction phase of the multicore scheduling process is less complex and so is suitable for runtime scheduling. Moreover, certain MoCs may ensure by construction deadlock freeness of the application. Finally, the MoC based approach can permits easy hardware and software co-design. SystemC language [Sys15] is now widely spread as it is an extension of the C language. Many other tools also exist, either from academic or industrial sources, such as Ptolemy[BHLM94], Preesm[PDH+14], OpenEM [Moe14], Orcc[YLJ+13], StreamIt[TKA02] or Simulink[Sim15].

The next section presents an embedded parallel runtime available for the TI's Keystone II platform targeted in this thesis.

2.3.4.3 An embedded programing runtime: the Open Event Machine

Open Event Machine (OpenEm) is a multicore RTOS [Moe14]. This embedded parallel runtime is focused on performance, scalability and flexibility, and is event based. Events are dispatched and consumed over the different cores of the platform.

An OpenEM application may be handled by multiple **processes**. A core set is associated with each process, but processes cannot share several cores. A process is composed of many runtime objects that are shared among all cores and are multicore safe. These objects can be one of the three following types:

- **Events:** each event belongs to an event pool. An event pool possesses a free queue which stores all unused events. Buffers can be attached to an event as payload.
- **Execution objects:** object that contains the algorithm to execute once an event is received. A receive function, that executes the algorithm, is registered for each execution object.
- **Event queues:** which connect events and execution objects. Each event queue is associated with one execution object and all queued events are processed by this execution object. Each event queue contains a context allowing persistent data storage; that is, data can remains alive before and after the event processing.

Each event needs to be scheduled and dispatched to be executed by an execution object. Where there is a unique dispatcher per core, there is one scheduler shared by all dispatchers.

The dispatcher life cycle is the following:

1. Request a new event from the scheduler.
2. The scheduler selects a non-empty event queue and sends the oldest event to the dispatcher.
3. The dispatcher fetches the execution object and executes the corresponding receive function.

4. The receive function consumes the input event.
5. The receive function can create one or more events and send them to another queues (even to those of another process).
6. The dispatcher returns to state 1.

It can be seen that each process requires one scheduler. The scheduling process dynamically maps events from different queues to each requesting dispatcher. The scheduler can take decisions based on four criteria:

- Priority: Each event queue has a priority. However there is no preempting mechanism if an event arrives in a high priority queue.
- Atomicity: A queue can be specified atomic, meaning that two events from this queue cannot be handled simultaneously.
- Locality: One or multiple queues can be only assigned to a subset of the core list.
- Order: If many events are available for scheduling, the oldest event will be scheduled.

This runtime is developed and deployed in Texas Instruments multicore DSPs. These platforms can use hardware coprocessors such as Fast Fourier Transform (FFT) co-processors, and network co-processors to transparently handle events. Furthermore, the scheduler of the OpenEm runtime is deployed on the Packet RISC engine of the Multicore Navigator meaning that the scheduling operations are executing in the background. Each scheduling operation is triggered by a scheduling request submitted by a dispatcher running on each core.

2.4 Conclusion

The major contribution of this thesis is the creation of a new programming technique called Just-In-Time Multicore Scheduling (JIT-MS) based on a dataflow MoC. Employing the MoC approach enables better decisions on the multicore scheduling method since the parallelism extraction is already defined by the MoC. It may also provide features such as deadlock freeness.

The JIT-MS method targets heterogeneous architectures with shared memory systems (NUMA). These architectures are increasingly more present in the MPSoCs market. They represent a good trade-off between multicore system performance and programming complexity. Distributed memory systems allow systems to have numerous PEs. Since increasingly more PEs are integrated into MPSoCs, this architecture may become very common in the future. The JIT-MS method has been designed to be easily extended to distributed memory systems.

JIT-MS is a fully-dynamic multicore scheduling process, focused on middle-grained parallelism. At this parallelism granularity, the program is separated into several pieces of computation that are dependent of each other. The proposed method allows extracting data, task and pipeline parallelism. This multicore programming method has been implemented in a runtime called Spider, which will be detailed in later chapters.

CHAPTER 3

Dataflow Models of Computation

3.1 Introduction

Diagrams have always been a convenient method to express an idea or specify a system. The majority of software architectures are based on diagrams such as the Unified Modeling Language (UML) diagrams [MBFBJ02].

In certain cases, graphical diagrams may also be used to specify a functional system. The implementation may then be inferred automatically from the graphical description. For example, Petri nets, Grafcet and ladder programming models are all based on drawn specifications [DA92, ZT98].

As seen in the previous chapter, MoCs can be used to parallelize applications using a middle granularity over multiples PEs on a platform. The MoCs allow the software programmer to express the application using a representation. Dataflow MoCs can be represented using graphical diagrams. These dataflow graphs are composed of tasks called "actors" exchanging data using links called "channels". This representation of the application allows the extraction of natural parallelism, which can enhance the application performance on parallel systems.

In this chapter, we will see that multiple dataflow MoCs are published in the literature. An overview of dataflow MoCs is provided in Section 3.2 and the most commonly used dataflow MoCs are then described in Section 3.3. We will see in this chapter that MoCs can be classified as dynamic or static dataflow. The primary advantage of a static dataflow MoC is predictability, allowing better scheduling decisions. Covertly, dynamic dataflow MoC providing reconfigurability to the application description.

The Parameterized and Interfaced Synchronous Dataflow (SDF) (PiSDF) MoC is the choice employed in this thesis. The PiSDF semantics is then introduced in details and compared with other dataflow MoCs. The PiSDF MoC is based on static dataflow MoCs and is enhanced by adding efficient management of parameters and hierarchy. It will be shown that PiSDF allows the reconfiguration of the application using locally static regions of the graph.

3.2 Dataflow Models of Computation (MoC): an Overview

3.2.1 What is a Model of Computation (MoC) ?

In general terms, a MoC is defined as a group of operational elements that when connected, describe the behavior of an application. A MoC description is composed of rules which specify the intended execution of the described application. The purpose of a MoC can vary: it may be used for specification, simulation or programming purposes.

Classical examples of MoCs are the Turing machine [Tur36] and the Lambda calculus [Bar84] models. During the last twenty years, numerous MoCs have been studied, due to the emergence of dataflow programming. The MoCs studied in this thesis are particularly useful for modeling signal processing applications; that is, applications which process ordered streams of data.

3.2.2 Dataflow MoC Definition

A dataflow MoC focuses on the representation of the flow of data that exists between a group of tasks collaborating within a single application. The application description is represented as a graph with vertices (tasks) connected to each other through edges (data communications). This representation explicitly specifies the dependencies that exist between tasks and, for certain models, also specifies the amount of required data communication between them.

The first dataflow MoC was defined by Kahn in 1974 and was called Kahn Process Network (KPN) [Kah74]. Kahn defined a KPN as a network of concurrent tasks connected by directed unbounded First-In First-Out queue (FIFO) channels transmitting *data tokens*. In his definition, data tokens are indivisible, produced only once and consumed only once, and they cannot be shared by tasks.

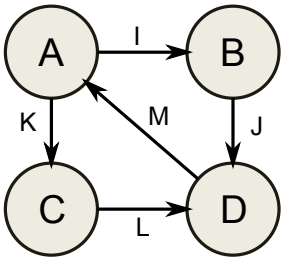


Figure 3.1 – *Kahn Process Network (KPN) example*

An example of a KPN network is proposed in Figure 3.1. In this example, the network is composed of 4 tasks (A, B, C and D) and 5 channels (I, J, K, L and M) which connect the tasks. In this network, tasks B and C can be executed in parallel since there is no dependency between them.

In 1995, Lee and Parks have specified how such tasks can behave in a dataflow network by defining the semantics of the Dataflow Process Network (DPN) MoC [LP95]. A Dataflow Process Network (DPN) is defined as follows:

Definition 3.2.2.1 (DPN Definition)
A DPN is a directed graph G noted as $G = (A, F)$ where:

- *A is the set of vertices called* actors *of G representing computational tasks. Each actor embeds:*
  - *IN: A set of n_{in} input ports represented as $IN_{0..n_{in}-1}$,*
  - *OUT: A set of n_{out} output ports represented as $OUT_{0..n_{out}-1}$,*
  - *F: A set of firing rules represented as $F_{0..n_f-1}$. A firing rule is a condition which, when satisfied, enables the execution (also called* firing*) of this actor,*
  - *R: A set of rates. A rate is the number of tokens consumed at an input port or produced at an output port corresponding to a specific firing rule. Each given firing rule has $n_{in} + n_{out}$ corresponding rates: one for each port.*
- *F is the set of edges representing* FIFO *data queues. These* FIFOs *are the only channels allowed to perform data token communications between actors. Each* FIFO *is connected to a source port (which provides tokens to the* FIFO*) and a sink port (which consumes tokens from the* FIFO*). A* FIFO *can also possess* delays*. A delay is the initial number of data tokens stored in the* FIFO *at the beginning of the application.*

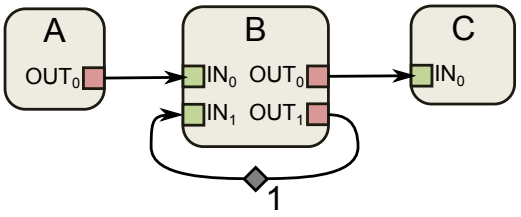


Figure 3.2 – *Dataflow Process Network (DPN) example*

The semantics of the DPN MoC only specifies the connections between actors. The graph network does not include details about firing rules or consumption and production rates. This information is usually stored in the actor definition. Actors can be described using any language, providing that the firing rules and FIFO accesses are supported. For instance, imperative languages such as C, Java or VHDL are commonly used. Specific actor description languages also exist, such as CAL Actor Language (CAL) [EJ03], CAPH [SBA13], SigmaC [dD13] or Cx [WSC+13]. These languages comprise both the coordination language (the language composing the graph definition) and the host language (the language that describes the actors).

3.2.3 Dataflow MoC Properties

Numerous dataflow MoCs exist in literature, with sometimes only slight differences between them. The majority are extensions of previously defined models, with either new features or new restrictions for improved analyzability. When comparing these models, it is necessary to identify key criteria.

In this section, the important properties for comparing MoCs employed by this thesis will be defined. These properties are not the only possibilities to characterize a dataflow MoC; however they have been chosen as they elucidate the differences in compile-time analyses.

Firstly, the two terms of schedulable and consistent need to be defined. These two properties are used to categorize applications described in dataflow. These properties do not categorize the MoC itself but rather the application description that uses the MoC.

Definition 3.2.3.1 (Dataflow Graph Schedulability)
A dataflow graph is Schedulable *if and only if it is possible to find a schedule, that is, a finite sequence of actor firing, for which it can fire without a deadlock and if the final graph state is equivalent to the initial graph state.*

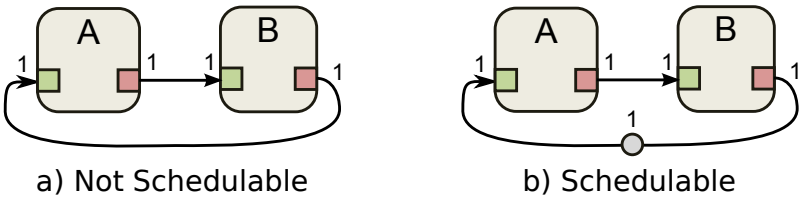


Figure 3.3 – *Schedulability example*

In Figure 3.3, an example of a non schedulable graph is shown using an SDF graph. An SDF graph specifies the number of tokens produced and consumed on each Fifo for each firing of the connected actors. There is only one firing rule per actor in an SDF graph and the rates are displayed directly in the graph. The SDF MoC will be more fully explained in Section 3.3.1.1. The graph of the Figure 3.3-a is not schedulable since to start, both actors A and B need the token produced when the other is fired, but neither has sufficient data tokens to start first. The graph of Figure 3.3-b is schedulable since the initial token stored in the delay enables actor A to be fired thus launching computation. The schedule obtained is (1xA,1xB): one execution of A followed by one execution of B which then returns the graph to its initial state. The schedulability of an application can also be subject to external factors such as the time constraints or memory limitations of a specific hardware platform.

Definition 3.2.3.2 (Dataflow Graph Consistency)
A dataflow graph is Consistent *if and only if its execution does not accumulate data tokens on any Fifo.*

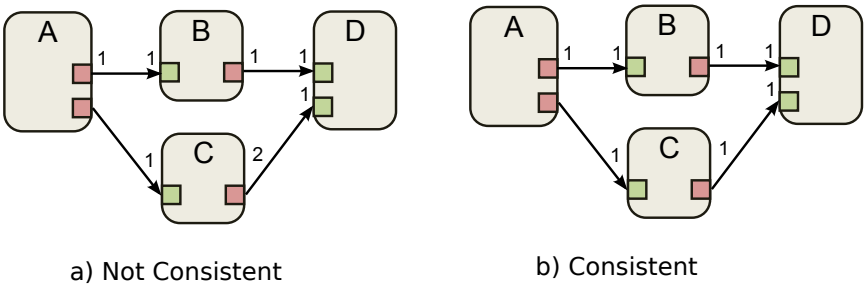


Figure 3.4 – *Consistency example*

Figure 3.4-a shows an example of a non-consistent graph using an SDF graph. A schedule which repeats (1xA,1xB,1xC,1xD) will result in indefinitely adding tokens to the CD Fifo and so will eventually lead to a memory leak of the application. Conversely, a schedule repeating (1xA,1xB,1xC,2xD) is not possible since it will starve the BD Fifo and lead to a deadlock. This may be contrasted with the second example in Figure 3.4-b, where the graph is consistent and the obtained schedule is (1xA, 1xB, 1xC, 1xD).

The identification of *schedulability* and *consistency* properties of dataflow application graph, allows the definition of key criteria of dataflow MoCs: Expressiveness, Decidability, Determinism, Reconfigurability, Predictability and Compositionality.

Definition 3.2.3.3 (Dataflow MoC Expressiveness)
As defined by [Far07], the theoretical expressivity of logic is defined as the measure of the ideas expressed in the logic without reference to the method of expression. For a dataflow MoC, expressiveness measures the ability of the MoC to describe a wide range of applications and constructs such as conditions, loops, parallelism and so on.

A particular case of dataflow MoC, that DPN MoC has maximal expressivity as has been proven to be Turing complete[BLo93], that is, it can describe all applications that may be represented with a Turing machine. The expressiveness of a MoC is almost always limited by model restrictions that are included to significantly simplify its analysis.

Definition 3.2.3.4 (Dataflow MoC Decidability)
As defined in [BL06], a dataflow MoC is decidable if and only if the schedulability and consistency of the representations derived from this MoC can be determined at compile time.

A decidable dataflow MoC is vital to ensure that each application will not result in deadlocks or constantly growing memory. However, decidable dataflow MoCs have limited expressivity because these MoC have additional rules and so restrict the designable application landscape [BL06].

Definition 3.2.3.5 (Dataflow MoC Determinism)
As defined in [LP95], a dataflow MoC is deterministic if and only if the graph behavior and execution depend solely on data values passing through it and not on external dataflow factors.

MoC determinism is usually a desirable feature since typical programming languages are determinate. However, non-determinism can extend expressiveness and allow a MoC to describe more types of applications.

Definition 3.2.3.6 (Dataflow MoC Reconfigurability)
According to [NL04], the reconfigurability of a dataflow MoC is defined as its ability to change the firing rule rate over time, as a function of certain parameters.

Reconfigurable dataflow MoCs are distinct from static MoCs, where rates are fixed to a specific value at compile time. Reconfigurable dataflow MoCs naturally have better expressiveness than static MoCs. Parameterized and dynamic dataflow MoCs are reconfigurable, and will be discussed in the following sections.

Definition 3.2.3.7 (Dataflow MoC Predictability)
The predictability of a dataflow MoC is defined as the ability of MoC to have Non-data-dependent behavior.

The static dataflow MoCs are the most predictable models, as their firing rules are known at compile-time and do not depend on data. Conversely, dynamic dataflow MoCs are the least predictable models as the actor firing rules for the majority of models depend on the value of the received data. The predictability of parameterized dataflow MoCs is between that of the dynamic and static MoCs; in these models, the firing rules of actors depend on parameters values but not on data. Parameters differ from data as they have their own communication channels and they can modify firing rules, but cannot serve as data.

Definition 3.2.3.8 (Dataflow MoC Compositionality)
As described in [TBG+13], the compositionality of a dataflow MoC is defined as the behavioral independence of the internal specification of the actors of a dataflow graph.

Non-compositionality of dataflow MoCs can be a problem for certain graphs since it can cause unexpected deadlocks when large applications are created from the description of their subparts. In [PBL95], a theorem to ensure compositionality of SDF specifications is presented.

3.3 The Landscape of Dataflow MoCs

Dataflow MoCs may be categorized into two classes: static and dynamic. The scheduling of applications described with dynamic MoCs is dependent on the incoming data of the actors. These models are generally derived from the previously defined DPN. Conversely, static dataflow models have predictable behavior making them schedulable at compile time. The major advantage of statically schedulable dataflow MoCs is their analyzability resulting in predictability and easier optimization. However, their static properties result in reduced expressiveness and lower flexibility for software programmers, when compared with dynamic dataflow MoCs.

In order to incorporate some of the designer friendly features into the static dataflow MoC, MoC extensions have been developed, allowing integration of hierarchical features. The extensions included in these hierarchical dataflow MoCs subdivide the application into subsystems.

The parameterized dataflow MoC is another extension. By giving the developer parameters that depend on incoming data, parameterized dataflow MoC offers better expressivity than the static dataflow MoC. Moreover, since parameters are explicitly defined in parameterized dataflow MoCs, they can be analyzed and so offer a similar predictability and analyzability to static MoCs.

The remainder of this section will detail static and parameterized dataflow MoCs.

3.3.1 Static Dataflow MoCs

Static dataflow MoCs are deterministic and, by definition, not reconfigurable. The sequence of firing rules and the rates of each actor port both are known at compile time, and so are independent of the data values passing through each FIFO. Thus, the static dataflow MoCs can be checked for schedulability at compile time, resulting in high predictability but also reduced expressiveness. A major restriction resulting from fixed firing rules is that the if-then-else statements based on data are not representable with static dataflow MoCs.

3.3.1.1 The Synchronous Dataflow (SDF) MoC

One of the most commonly used static dataflow MoCs is the Synchronous Dataflow (SDF). This model was firstly introduced in [LM87]. Formally, the SDF model is defined as follows:

Definition 3.3.1.1 (SDF MoC definition)
An SDF graph is a directed graph G noted as $G = (A, F)$ that conforms to the DPN definition (3.2.2.1). However, there is only one firing rule per actor and rates are fixed scalars.

An example of an SDF graph is given in Figure 3.5. In this figure, actor A produces 2 tokens at each firing which are passed to actor B which consumes one token per firing. A

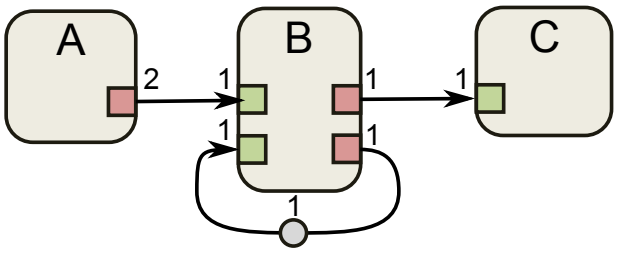


Figure 3.5 – *SDF graph example*

delay is present on the feedback edge of actor B and contains 1 token, which is required in the initial state of the graph.

Analyzability and compile time scheduling are the key features which have made SDF such a well known dataflow model. Low complexity algorithms from the literature can be used to check the consistency and schedulability of SDF graphs and to determine an efficient schedule.

3.3.1.1.1 Consistency and Schedulability of SDF

The schedulability and consistency properties of an SDF model are vital; they are necessary for the creation of a valid mono-core or multicore schedule. A valid schedule fires with no deadlocks and its final state is equal to its initial state, making it indefinitely runnable.

To allow an analytic study of an SDF graph, and to check the schedulability and consistency properties, a **topology matrix** is derived from the SDF graph. This matrix is defined as follows:

Definition 3.3.1.2 (SDF Topology Matrix)
The topology matrix Π *is a matrix of size* $\|A\| \times \|F\|$ *where each row is associated with one* Fifo *and each column is associated with one actor. Elements of the matrix are computed as:*

$$\Pi(a, f) = n$$

where n *is the amount of tokens produced (if positive) or consumed (if negative) by Actor* a *into the* Fifo f*. If the actor does not produce or consume tokens on* f*, the value* n *is set to* 0*.*

The topology matrix of the SDF graph of Figure 3.5 is as follows:

$$\Pi = \begin{array}{c} \\ AB \\ BC \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{pmatrix} 2 & -1 & 0 \\ 0 & 1 & -1 \end{pmatrix} \end{array}$$

The feedback Fifo connected to actor B is not displayed in the topology matrix as it does not influence interactions between actors of the graph. Its sole role is to sequentialize the firings of actor B.

Theorem 3.3.1.3 (SDF graph consistency)
An SDF graph is consistent if and only if:

$$rank(\Pi) = \|A\| - 1$$

where $\|A\|$ *is the number of actor in the graph* G*.*

By checking the rank of the topology matrix, Theorem 3.3.1.3 implies that there is a unique vector, herein called Basic Repetition Vector (BRV) of size $\|A\|$ which is the base of solution q of equation $\Pi.q = 0$.

The computation of the BRV can be performed by the Gauss-Jordan algorithm as described in [LM87]. The result must be the smallest solution for which all vector values are positive integers and the BRV is not null.

Theorem 3.3.1.4 (SDF graph schedulability)
An SDF graph G is schedulable if and only if:

- *G is consistent,*
- *G is deadlock-free: a schedule can be found, compliant with the BRV, that respects all actor firing rules.*

An example that fails the requirement of deadlock freeness can be found in Figure 3.3-a. It can been seen that there are insufficient initial tokens, and this causes the requirement failure. The procedure to construct a single-core schedule to demonstrate schedulability is detailed in [BELP96].

3.3.1.1.2 A pre-scheduling transformation of SDF graphs

To create a multicore schedule of an SDF graph, an intermediate representation with the maximum possible parallelism can be created. This intermediate representation is called **Single-Rate Directed Acyclic Graph (DAG) (SRDAG)**. This graph is a dataflow model that is defined as followed:

Definition 3.3.1.5 (Single-Rate DAG (SRDAG) definition)
The SRDAG MoC is a restriction of SDF MoC; these restrictions are:

- ***Acyclic**: an SRDAG does not comprise cycles.*
- ***Single-Rate**: the production rate on each Fifo is equal to the corresponding consumption rate of this same Fifo.*

The conversion from SDF graph to SRDAG consists of multiple steps. First, the previously defined BRV must be computed:

$$\Pi = \begin{array}{c} \\ AB \\ BC \end{array} \begin{array}{c} \begin{array}{ccc} A & B & C \end{array} \\ \begin{pmatrix} 2 & -1 & 0 \\ 0 & 1 & -1 \end{pmatrix} \end{array}$$

Gives:

$$BRV = \begin{array}{c} \\ A \\ B \\ C \end{array} \begin{array}{c} \begin{array}{c} \end{array} \\ \begin{pmatrix} 1 \\ 2 \\ 2 \end{pmatrix} \end{array}$$

Then, accordingly to its repetition value, each actor of the SDF graph is instantiated multiple times in the created SRDAG. Unfortunately, the graph expansion due to this step may lead to exponential growth of the SRDAG.

Finally, each instantiated actor must be linked to other instances. It may be necessary in this link step to create some new actors called special actors to handle the token flow over the newly created single-rate actors. This method was introduced in [Pia10].

Delay tokens also require special care. When initial tokens are present in a FIFO as a delay, these tokens must also be generated for the SRDAG. These tokens are created using a special actor called *Init*. An additional actor called *End* is required to discard the last token unused by the rest of the graph.

Figure 3.6 presents the SRDAG generated from the example graph of Figure 3.5.

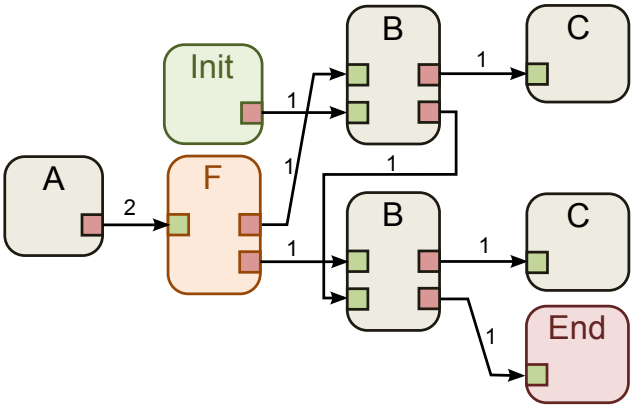


Figure 3.6 – *SRDAG resulting from SDF example*

This single-rate transformation allows the multicore scheduling to make use of task parallelism as well as data parallelism. The SRDAG can be used as a precedence graph to map, order and time all the actors present in it.

3.3.1.2 The Cyclo-Static Dataflow (CSDF) and Affine Dataflow (ADF) MoCs

The Cyclo-Static Dataflow (CSDF) MoC is an extension of the SDF MoC and is described as follows:

Definition 3.3.1.6 (CSDF MoC definition)
An CSDF representation is a graph G noted as $G = (A, F)$ where each port is associated with a sequence s of n fixed scalar rates.
Considering an actor $a \in A$ and a port p of a, the consumption or production rate of p for the i-th firing of actor a is given by $s[i \mod n]$.

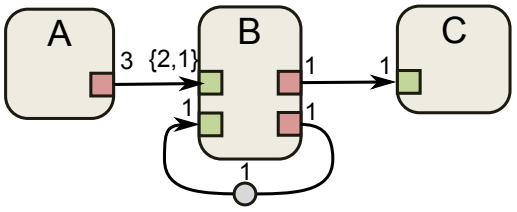


Figure 3.7 – *CSDF graph example*

An example of CSDF graph is given in Figure 3.7. In this example, the first input port of actor B has a sequence of 2 token rates.

This MoC extension may be interpreted as an increase of expressiveness when compared to SDF, however this is not the case. An application that can be described in CSDF can be

described in SDF; generally, the CSDF description requires fewer actors than in SDF (the model is said to be more compact). Moreover, CSDF can make an application schedulable with less delay tokens. A transformation has even been proposed in [PPL+95] to transform a CSDF graph into an SDF graph with same properties. All techniques used to analyze a SDF graph can be applied to CSDF after transformation. The resulting SRDAG of the example in Figure 3.7 is given in Figure 3.8.

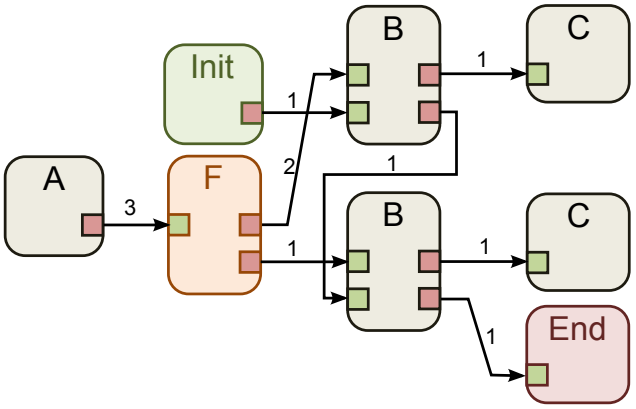


Figure 3.8 – *SRDAG resulting from CSDF example of Figure 3.7*

The Affine DataFlow (ADF) MoC is another extension of SDF MoC with the same objective: improving model compactness. The ADF model has been introduced in [BTV12]. While CSDF has an infinite sequence of rates, the ADF replaces the unique rate of SDF with an initial sequence of firing rates followed by an infinitely repeated sequence of firing rates. These two concatenated sequences are called the ultimately periodic sequence. As with CSDF, ADF does not extend expressiveness. A transformation also exists to convert an ADF graph into an equivalent SDF graph.

3.3.2 Hierarchical Dataflow MoCs

Hierarchy is an important feature of dataflow MoCs. It can be used to enhance two properties of a model: Compositionality and Reusability.

Reusability is useful for an incremental approach of system design. Reusability is the ability to define a component that can be instanciated multiple times in the design process. Certain primitive blocks such as basic signal processing algorithms (fft, dct, fir, iir) or basic image processing algorithm (Sobel filter, median filter, and so on) are reference designs that can be reused in many applications.

Hierarchy is of interest in a dataflow MoC since the internal behavior of a dataflow actor is usually hidden in the top level graph. However, hierarchy is included in DPN and SDF semantics.

Several generalizations of the SDF MoC have been proposed [PBR09, LM87, TBG+13] to include hierarchy as an explicit part of the SDF MoC.

3.3.2.1 Non Compositional Hierarchy Mechanism for SDF

A method to add hierarchy into the SDF MoC has been introduced by Lee in [LM87]. This implementation consists of associating an SDF graph with a hierarchical actor to define its internal behavior.

This is performed by flattening the graph to a single-level before executing an analysis or scheduling tools to the upper SDF graph (also called *top graph*). This flattening operation consists of the sole step of replacing the hierarchical graph by its inner graph.

An example of this operation is shown in Figure 3.9, where the hierarchical actor H is composed of two subactors H_1 and H_2.

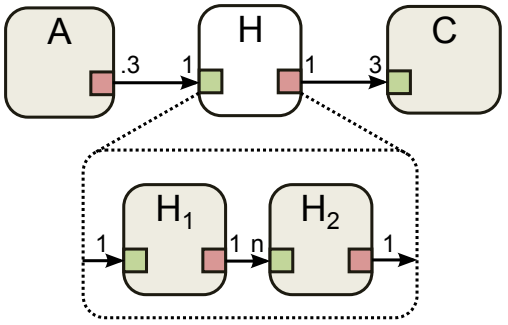


Figure 3.9 – *Hierarchical SDF graph example*

However, as seen in [Pia10, TBG+13], there is a compositionality problem due to this hierarchical implementation. When only the top graph is taken into account, the actors A and C will be fired once and the actor H will be fired three times.

It may be noted that, depending on the value of rate n, the flattened graph leads to different but equivalent SRDAGs, as presented in Figure 3.10. The upper graph in Figure 3.10 corresponds to $n = 1$, and in this case the analysis of the top graph is respected. It may be seen that a fork actor F has been introduced to distribute the tokens to different FIFOs and that a join actor J has been inserted to interleave result tokens into a single FIFO. When $n = 2$, the repetition values of actors in the top graph are changed, as displayed in the lower graph of the Figure 3.10. This graph contains two firings of actor A, which does differs from the analysis of the upper graph.

This example highlights the non-compositionability of SDF graphs. It is necessary to use a top-down approach with a compositional model to ensure that the result is a schedulable graph. The non-compositionability of SDF requires the analysis process to flatten the whole graph before commencing the schedulability analysis. Thus analysis complexity can be very high and the graph may suffer from an exponential growth due to flattening.

3.3.2.2 The Interface-Based SDF (IBSDF) MoC: a Compositional Dataflow MoC

The Interface-Based SDF (IBSDF) MoC [PBR09] is an extension of the SDF MoC that ensures compositionability. IBSDF is defined as follows:

Definition 3.3.2.1 (IBSDF MoC definition)
An IBSDF graph is a graph $G = (A, F, I)$ where:

- *The external behavior of an actor conforms to the SDF MoC and its internal behavior can be specified with an IBSDF graph, called a subgraph.*
- *I is a set of interfaces. Interfaces enable the transmission of information between levels of hierarchy. Each interface $i \in I$ corresponds to a data port of the hierarchical actor.*

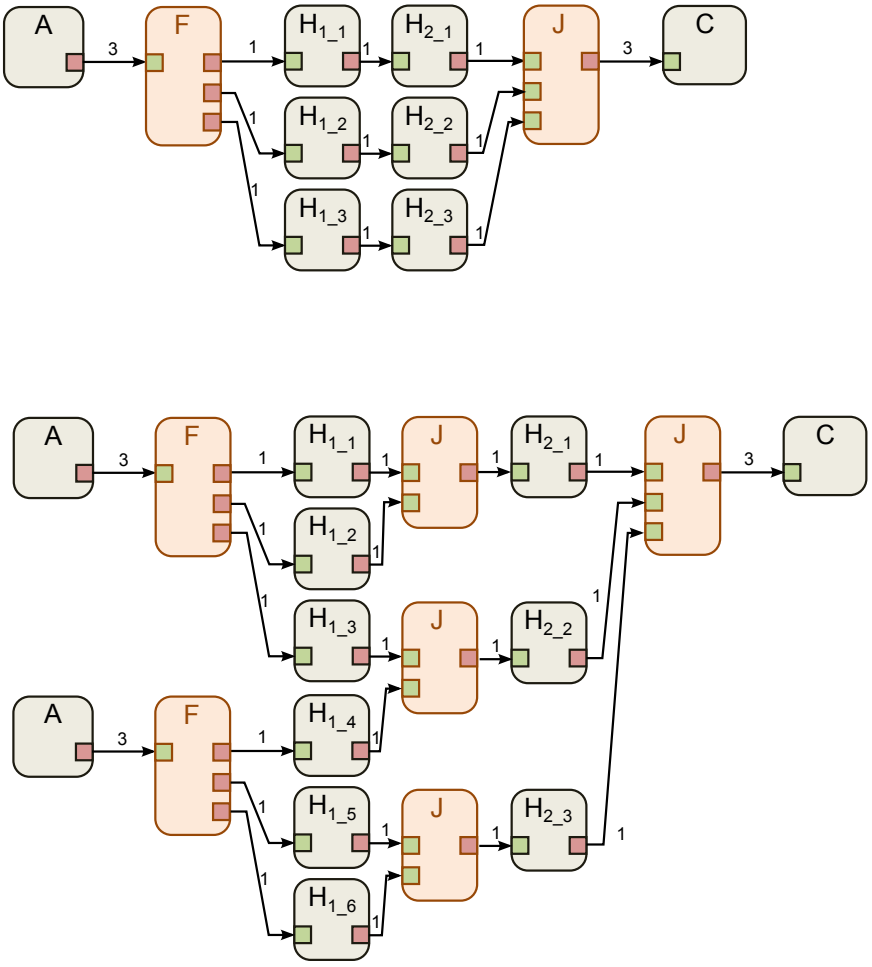


Figure 3.10 – *Resulting SRDAG from Hierarchical SDF graph example*

- *If more data tokens are consumed on a data input interface than specified by its rate, the data input interface then behaves as a round buffer, producing the same tokens several times.*

- *If a data output interface receives more tokens than specified by its rate, only the last received tokens will be forwarded to the upper level of hierarchy.*

The interfaces of the IBSDF ensure compositionality of the dataflow MoC. The example of the first SDF approach (Figure 3.9) is represented with the IBSDF MoC in Figure 3.11.

However, the SRDAG hat results from Figure 3.11 is different to Figure 3.6 due to interface rules. This SRDAG is displayed in Figure 3.12. The BRV of the top graph is respected in both generated SRDAGs. The single-rate transformation now leads to the creation of a new special actor called *RoundBuffer*. This actor has the role of duplicating tokens (for input interfaces) and of discarding all but the last tokens (for output interfaces).

3.3.3 Parameterized Dataflow MoCs

As discussed previously, parameterized dataflow MoC has an advantage over a static dataflow MoC: it has more expressiveness. By allowing the reconfigurability of rates, pa-

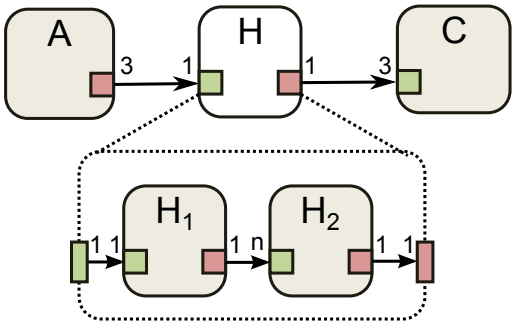


Figure 3.11 – *IBSDF graph example*

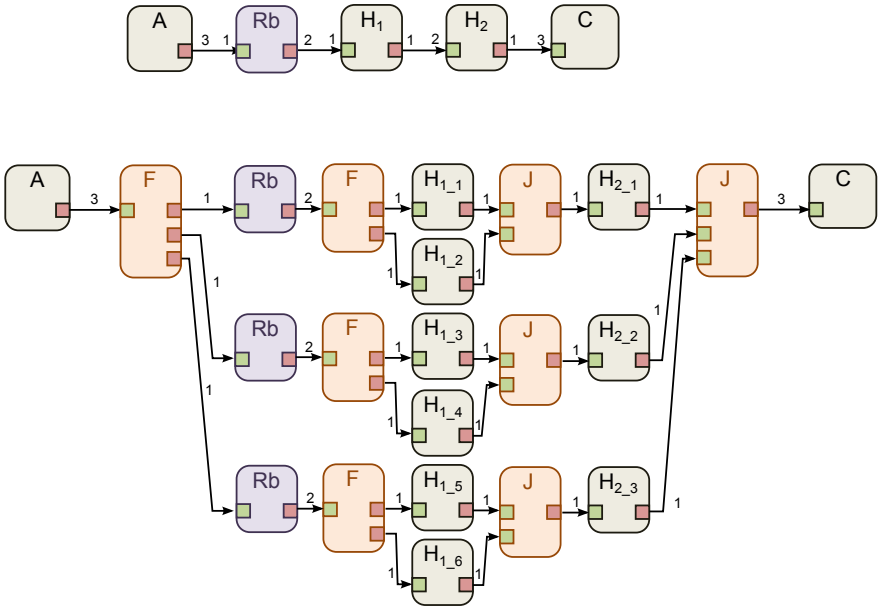


Figure 3.12 – *Resulting SRDAG from IBSDF graph example*

rameterized dataflow MoCs can describe more applications. However, this feature generally reduces the predictability of these MoCs.

In [NL04], Neuendorffer and Lee show that the predictability of a MoC can be enhanced by allowing reconfiguration only at certain instants. These instants are called *quiescent points* and exist multiple times during the execution of an application.

There are several parameterized dataflow MoCs that extend SDF with reconfigurability features such as Scenario-Aware Dataflow (SADF) [TGB+06] or Integer-controlled DataFlow (IDF) [Buc94]. Four parameterized models will be detailed in this section to cover the key aspects of parameterization: Boolean DataFlow (BDF), Schedulable Parametric Dataflow (SPDF), Parameterized SDF (PSDF) and PiSDF.

3.3.3.1 The Boolean DataFlow (BDF) MoC

The BDF MoC is an SDF graph with additional special actors called *switch* and *select* as displayed in Figure 3.13. The BDF MoC has been defined in [BLo93].

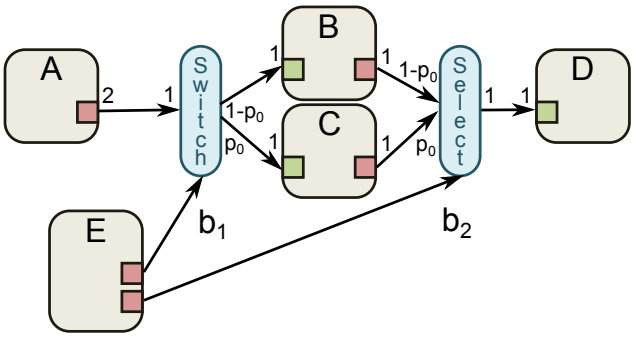


Figure 3.13 – *Boolean DataFlow (BDF) graph example*

Switch and select actors are equivalent to demultiplexers and multiplexers respectively, and are used in this model to implement if-then-else patterns by forwarding information to different FIFOs. As a result of a boolean value, the switch actor either forwards its input tokens to its first output port or its second output port, thus acting like a demultiplexer logic cell. The select actor behaves as a multiplexer: the boolean value selects the input port whose tokens are then sent to the unique output port.

The BDF model adds control flow to the SDF dataflow and makes the model Turing-complete [BLo93].

3.3.3.2 Schedulable Parametric Dataflow (SPDF) MoC

The Schedulable Parametric Dataflow (SPDF) MoC [FGP12] is an extension of the SDF MoC with expressivity equivalent to that of the DPN, but with a better predictability. The Schedulable Parametric Dataflow (SPDF) is defined as follows:

Definition 3.3.3.1 (SPDF MoC definition)
A SPDF graph $G = (A, F, P)$ is a graph where:

- *P is a set of parameters. Parameters can change dynamically and are set by an actor called* modifier*. A* period *is set for each parameter, displayed within square brackets, that specifies the smallest number of modifier firings between two parameters modifications.*
- *Rates may be either scalars or functions of parameters.*

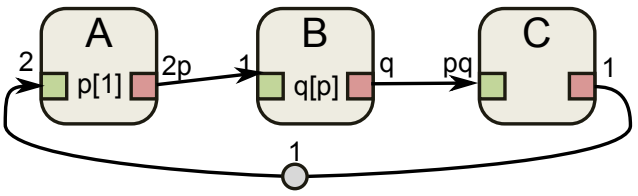


Figure 3.14 – *Hierarchical SDF graph example*

Theorem 3.3.3.2 (SPDF safety criterion)
As defined in [FGP12], the modifier actor must not change the value of a parameter during

the iteration of the part of a graph where this parameter is used. This criterion ensures that the consistency and the schedulability of SPDF graphs can be checked at compile time.

An example of a SPDF graph is shown in Figure 3.14. In this example, there are two parameters, p and q. Both parameters influence consumption and production rates of actors in the graphs.

The safety criterion is respected in this example. The parameter q is used by actor C and since its modifier is actor B, the graph description must ensure that the parameter q will not change until actor C is fired. This condition is respected, as the parameter q is defined as constant for p firings of actor B thus satisfying the $p * q$ consumption of actor C.

This parameter extension allows a modifier actor to change its output rates at each firing using a parameter. This feature, added to SDF, is sufficient to ensure the Turing-equivalent property of a SPDF MoC as expressed in [BLo93].

3.3.3.3 Parameterized SDF (PSDF) MoC

Parameterized dataflow is a meta-modeling framework introduced by Bhattacharya and Bhattacharyya in [BB01]. When this meta-model is applied, it extends the targeted MoC semantics by adding reconfigurable hierarchical actors. Notably, it can be applied to SDF [BB01] or to CSDF [KSB+12]. This section will focus on its application to PSDF.

Definition 3.3.3.3 (PSDF MoC definition)
A PSDF graph $G = (A, F, P)$ is a graph where:

- *P is a set of parameters. A parameter $p \in P$ is an integer value that can be used as a production or consumption rate for an actor A and/or influence its internal behavior. The value of parameters is not defined at compile time but rather at run time by another actor.*
- *One hierarchy level is specified with 3 subgraphs, namely the init ϕ_i, the subinit ϕ_s, and the body ϕ_b subgraphs.*
  - *the init ϕ_i subgraph sets parameter values that can influence the rate of the hierarchical actor itself as well as ϕ_s and ϕ_b subgraphs. The ϕ_i subgraph is executed only once per iteration of the graph to which this hierarchical actor belongs.*
  - *the subinit ϕ_s subgraph sets values of all remaining parameters of the body ϕ_b subgraph. The ϕ_s subgraph is executed prior to each firing of the hierarchical actor. It can consume data tokens on input ports of the hierarchical actor but cannot output data tokens.*
  - *the body ϕ_b subgraph is executed when its configuration is complete, immediately after the completion of ϕ_s. The body subgraph behaves as a graph implemented with the MoC to which the parameterized dataflow meta-model was applied.*

In the execution of a PSDF graph, reconfiguration occurs when values are given to the parameters of a hierarchical actor. Actor computation and actor production and consumption rates depend on these parameter values. An example of a PSDF graph is given in Figure 3.15.

This graph contains two parameters, L and N. These two parameters influence the rate of body actors. Parameter L is set in the init subgraph. The subinit graph is executed and then the parameter N is set. This allows the body actor to be executed; its execution and the BRV depend on the values of the parameters of L and N.

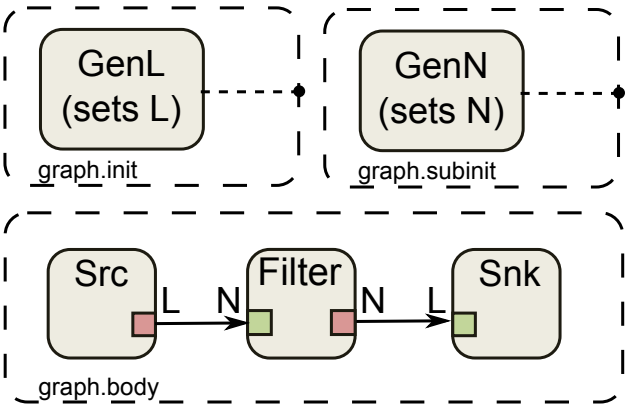


Figure 3.15 – *Hierarchical SDF graph example*

3.3.3.3.1 PSDF Runtime Operational Semantics

The runtime operational semantics of a dataflow MoC defines the successive steps of the graph execution. The operational semantics of PSDF is presented in [BB01]. The top graph is considered as a top hierarchical actor. Its execution requires the following steps:

1. Instantiate all actors in the init subgraph ϕ_i.
2. Fire all actors in the init subgraph ϕ_i.
3. Compute the BRV of the subinit subgraph ϕ_s and pre-compute BRV body subgraph ϕ_b. Parameter values used at this point are those values set in step 2 and default values required for parameters whose values will be set in step 6. The pre-computed body BRV is needed to obtain the interface rates of the current actor for the top graph.
4. Wait for the next firing of the current actor.
5. Instantiate all actors in the subinit subgraph ϕ_s.
6. Fire all actors in the subinit subgraph ϕ_s.
7. Compute the body BRV with the parameter values set in steps 2 and 6.
8. Instantiate all actors in the body subgraph ϕ_b.
9. Fire all actors in ϕ_b.
10. Return to step 4.

This loop restarts from step 1 only if a new iteration of the graph containing the current actor is fired again.

As presented in [SGTB11], the operational semantics make the PSDF MoC more predictable than the DPN but less predictable than the SDF MoC. Indeed, for all parameterized dataflow MoCs, the topology of a PSDF graph is undefined at compile time, as it depends on dynamically set parameter values.

However, this topology is defined when actors of the subinit subgraph are executed, which corresponds to a quiescent point. Consequently, this means that firing rates of

actors are known further in advance than with the DPN MoC where the firing rates can non-deterministically change at each actor firing.

3.3.3.3.2 PSDF Analyzability

When using the PSDF MoC, the schedulability of a graph may be proven at compile time, by checking its local synchrony for certain application descriptions [BB01].

A PSDF graph is locally synchronous if it is schedulable for all possible configurations. Furthermore, all its hierarchical children must be locally synchronous. To obtain this property, the three following conditions must be satisfied:

1. The init, subinit and body graphs must be schedulable for all possible configurations.
2. Each invocation of the init and the subinit graphs must give a unique value to each parameter set by the subgraph.
3. Consumption rates of subinit and body graphs on interfaces must depend only on parameters set by the init graph.

The first condition may lead to a very large number of configurations to check, resulting in an exponential number of reachable configurations. These checks can be moved to run-time execution if needed, that is, at the steps 3 and 7 of the operational semantic.

3.3.3.4 Parameterized and Interfaced SDF (PiSDF)

The Parameterized and Interfaced dataflow Meta-Model (PiMM) has been introduced by Desnos et al. in [DPN+13]. It can be used in an equivalent way to the parameterized dataflow to add parameterization to a dataflow MoC. However, the PiMM also adds an interfaced hierarchy feature which is similar to IBSDF.

In this section, the PiMM is presented as it is applied to the SDF MoC resulting in a MoC called PiSDF. However PiMM can be applied to other dataflow MoCs to introduce hierarchy with compositionality and parameterization.

3.3.3.4.1 PiSDF Semantics

The PiSDF semantics is formally defined as follows:

Definition 3.3.3.4 (PiSDF MoC definition)
A PiSDF graph $G = (A, F, I, P, D)$ extends an SDF graph $G = (A, F)$ with the following additions:

- *I is a set of hierarchical interfaces. An interface is a vertex of the graph that passes data tokens or parameter values between levels of hierarchy.*
- *P is a set of parameters. A parameter is a vertex of the graph and is used to configure the application and to modify its behavior.*
- *D is a set of parameter dependencies. A parameter dependency is a directed edge of the graph that propagates parameter configurations to other elements of the graph.*

An example of a hierarchical PiSDF graph is given in Figure 3.16.

A parameter such as N or L is a vertex of the graph associated with a parameter value and is used to configure elements of the graph, usually integer values.

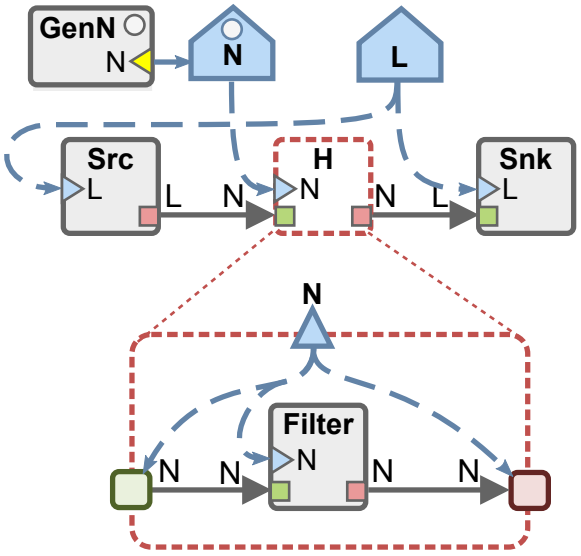


Figure 3.16 – *PiSDF Graph Example*

In a PiSDF graph, an actor can embed 4 different types of ports. The commonly used input and output data ports are present but since parameters are explicitly described in the graph, an actor can also embed input and output configuration ports that connect to parameter dependencies. This is achieved by using a dependency is to connect parameters to actor configuration ports (represented by dashed blue arrows in graph of Figure 3.16).

An output configuration port can be only connected to a parameter. This parameter will then be dispatched to other graph elements, creating a parameter dependency tree. A parameter can depend on other parameters, combining them in an expression to obtain a new value. A parameter can influence one or more of the following properties: the computation of an actor, the production/consumption rates on the ports of an actor, the value of another parameter, and the delay of a Fifo.

The hierarchy semantics used in PiSDF is inherited from the IBSDF model introduced in [PBR09]. In a PiSDF graph, a hierarchical actor is associated with a unique PiSDF subgraph. PiMM adds two additional interface types: the input and output configuration interfaces. From a subgraph perspective, an input configuration interface is equivalent to a locally static parameter since it does not change for this graph iteration.

There are two types of parameters in PiSDF: configurable parameters and locally static parameters. However, both these values must be fixed when scheduling the current graph.

A configurable parameter is one whose value is dynamically set once at the beginning of each graph iteration. Configurable parameters cannot influence the rates of input and output data interfaces, and are represented with a white circle in Figure 3.16.

Configurable parameters can be set either by the result of a dependency upon another configurable parameter or from an output configuration port of an actor. In the example of Figure 3.16, N is a configurable parameter set by an actor; this actor is thus called a *configuration actor*.

The firing of a configuration actor produces a parameter value. A parameter dependency uses this value to dynamically set the value of the configurable parameter. The configuration actor in Figure 3.16 is the *GenN* actor, and is represented with a white circle.

Since the execution of a configuration actor results in a reconfiguration of the graph, it is only permitted at quiescent points during a graph execution. This restriction leads to the following rules:

- A configuration actor must be fired only once per iteration of its parent graph. This unique firing must happen before the firing of a non-configuration actor, called body actor.
- A configuration actor can only be connected to input data interfaces of its parent graph and must have the same rate as the interfaces.
- The rate of a configuration actor can depend only on locally static parameters.
- Output data ports of a configuration actor are seen as an input data interface by other actors of the same graph.

3.3.3.4.2 Runtime Operational Semantics

Based on PiSDF semantics, the execution of a graph G contains the following steps:

1. Wait for all configuration input interfaces to receive a parameter value.
2. Compute the rates on the input and output data interfaces using the partial configuration.
3. Wait until the hierarchical actor is fired by its parent graph.
4. Fire the configuration actors of the current graph which will set the configurable parameters. The graph is now fully configured.
5. Compute the BRV of the graph to ensure its schedulability and find a schedule.
6. Fire the body actors following the computed schedule.
7. Produce, the data tokens and parameter values computed by the actors on the data ports and output configuration ports.
8. If necessary, return to step 3 and initiate a new firing of the graph.

The operational semantics of the PiSDF MoC is quite similar to that of the PSDF MoC. Steps 1 and 2 correspond to the init subgraph, steps 3 to 5 correspond to the execution of the subinit subgraph, and steps 6 to 8 correspond to the execution of the body subgraph. When compared to PSDF, the advantages of the PiSDF over the PSDF are the definition of parameter dependencies and the simplification of the operational semantics.

3.3.4 PREESM: a Framework supporting the PiSDF MoC

The Parallel and Real-time Embedded Executives Scheduling Method (Preesm) is an Eclipse-based framework that provides dataflow-based methods to study and program embedded multicore platforms [PDH+14]. This framework is open-source and many tutorials can be found on the website [PRE15] for the easy initiation of C/C++ programmers to multicore programming.

The Preesm framework focuses on providing high level rapid prototyping information on algorithm parallelism and latency. It also proposes detailed analyses on system memory

requirements. Moreover, a platform adaptable C/C++ code generation is provided to transform the dataflow representation into a runnable code.

This framework is based on the PiSDF MoC. This dataflow model describes the input algorithm and actor code is not required by the framework for simulation purpose. The executable program resulting from jointly compiling the generated and the manual code and constitutes a multicore system prototype that is guaranteed to be deadlock-free and can be retargeted to a different number of cores within minutes.

However, since this framework is a compile-time analysis tool, all code generation of this framework is restricted to static PiSDF graphs. A static PiSDF graph only embeds parameter values that are fixed and known at compile time.

The Preesm framework is detailed in Figure 3.17. It can be seen that the Preesm framework requires three inputs:

- An Architecture model: this model is a System-Level Architecture Model (S-LAM) graph that represents the architecture [PNP+09]. It lists the available cores of the platform as well as the logical communication media between them.
- A PiSDF representation of the application
- A Scenario: a database providing all necessary information to link an algorithm and an architecture.

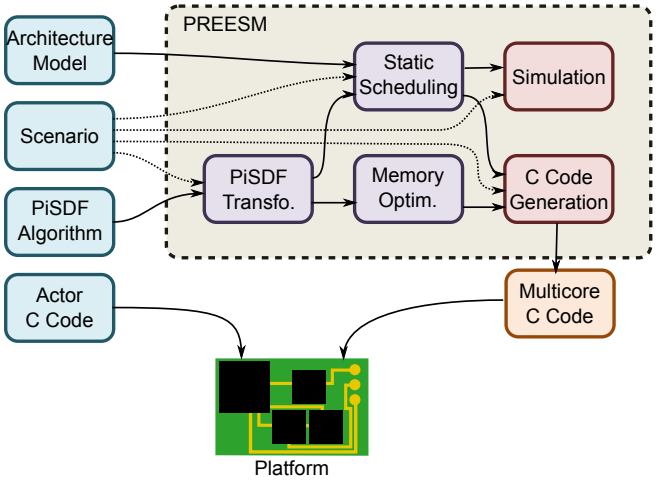


Figure 3.17 – *Preesm Framework*

Firstly, the Preesm framework performs dataflow transformations on the PiSDF algorithm to expose the parallelism. The single-rate transformation is part of this process. The intermediate transformed representation is then used to schedule the application for the target platform.

Next, the static scheduling transformation generates a periodic self-timed multicore schedule to process the input data stream that will be repeated indefinitely. The embedded multicore scheduler implements the List and Fast scheduling methods described by Kwok [Kwo97] The current Preesm plug-ins are only oriented towards latency minimization.

A memory optimization task computes optimizations to reduce execution memory requirements. As described in [DPNA15b], a memory exclusion graph is created in order to authorize memory reuse between Fifos.

Finally, the simulation task provides a rapid feedback to the designer and certain metrics for system design. Notably, a simulated Gantt chart of the code execution on the parallel architecture is generated. The code generation tasks generate a self-timed code for each core that handles actor firing, and data fetching in addition to synchronizing the cores.

3.4 Conclusion

PiSDF is one of numerous existing dataflow MoCs.

Its has advantages over models that includes actor states, such as Dynamic DataFlow (DDF). The DDF forbids multiple simultaneous executions of an actor. This constraint is due to internal states within each actor that must be progressively updated after they have been sequentially executed. In contrast, the SDF stateless MoC allows multiple repetitions of the same actor immediately after the self-feedback contribution from a Fifo (that is, no data dependency) ceases between executions. Self-feedback Fifos can be used to model the state of an actor visible to the dataflow model. However, the range of constructs representable with DDF is bigger than that representable with SDF.

A distinguishing feature of PiSDF is the integration of a tree of parameters to asynchronously transmit control values to actors [DPN+13]. The PiSDF model has been proposed to exploit the parallelism offered by locally static periods of execution, when all parameters that influence scheduling remain stable.

This model combines properties of expressivity, compositionality and predictability in such a way that a wide range of applications may be represented and exploited efficiently. Reconfiguration using parameters allows good predictability of the application execution that immediately follows, permitting informed scheduling decisions.

The execution of the PiSDF dataflow MoC on a multicore heterogeneous platform is the focus of this thesis. The current work uses some of the features of the Preesm framework and explores reconfigurable applications by defining a dataflow-based embedded runtime.

Part II

Contributions

CHAPTER 4

JIT-MS: a PiSDF-based Multicore Scheduling Method

4.1 Introduction

One of the major challenges of designing and implementing multicore signal processing systems is to dispatch computational tasks efficiently onto the available PEs. It is possible additionally to require the dispatch task to include dynamic modifications in the application functionalities or in the resource requirements. This process of assigning, ordering and timing actors on PEs in this context is called *multicore scheduling*, and was presented in Section 2.3.4.1.

Inefficient use of the PEs leads to longer processing times. The consequent idle periods of the processors result in unnecessary static power consumption, requiring higher energy. Data dependencies and dynamic signal processing algorithms make multicore scheduling a complex problem as stated in [MTK+11].

A major contribution of this thesis is the elaboration of a novel multicore scheduling method to address these design challenges. This method is based on the PiSDF MoC, and is called Just-In-Time Multicore Scheduling (JIT-MS). JIT-MS is a flexible scheduling method that takes strategic scheduling decisions at runtime. It focuses on optimizing the mapping of application functionalities onto multicore processing resources. In relation to the scheduling taxonomy defined by Lee and Ha [LH89], JIT-MS is a *fully dynamic* scheduling strategy. It handles heterogeneous platforms and focuses on middle grained parallelism. JIT-MS can exploit data, task and pipeline parallelism as presented in Section 2.3.3. This method is embedded into a runtime system called Synchronous Parameterized Interfaced Dataflow Embedded Runtime (Spider) that is described in Chapter 6.

With the range of MoCs available in the literature, the choice of input MoC impacts the scheduling method in terms of performance and implementation. Selecting an adequate MoC in terms of predictability may potentially provide more parallelism to the scheduler. Conversely, lack of expressiveness is a typical limitation of a MoC with good predictability. Lower expressiveness limits the range of constructs that can be modeled.

In this thesis, the PiSDF model has been selected to exploit the parallelism offered by locally static periods of execution, when all parameters that influence scheduling remain stable. The fundamental idea is that the runtime detection and exploitation of local SDF properties can produce scheduling solutions that are more efficient and more predictable. The PiSDF model allows the designer to model conditions through parametric dataflow

graphs with variable parameter values. JIT-MS is proposed to exploit the features of the PiSDF model. JIT-MS efficiently extracts the potential parallelism of the application and reduces the overall latency of the dataflow graph execution.

An iterative method is used by JIT-MS to handle runtime reconfiguration and hierarchical execution. The multicore scheduling method is then divided into multiple steps and is focused on progressively building an intermediate representation called a *single-rate* graph. This intermediate representation is used to unveil parallelism of the application and acts as a task dependency graph, which is then used for the Mapping/Ordering task.

This chapter is organized as follows: Section 4.2 gives an overview of the JIT-MS method. Then, the PiSDF-specific BRV and the single-rate transformation are described in Sections 4.3 and 4.4 respectively. Finally, the global iterative scheduling method is given in Section 4.5.

4.2 Overview of the JIT-MS method

An overview of the JIT-MS is presented at Figure 4.1. The JIT-MS method is based on an intermediate graph called Single-Rate DAG (SRDAG).

An SRDAG is a dataflow graph where each actor is instantiated the number of times that it must be fired. The SRDAG corresponds to the current schedule of a PiSDF graph, and it is *reset* at the end of each graph iteration. The graph transformation from PiSDF to SRDAG is highly dependent on the values of PiSDF parameters. The SRDAG can be updated after each parameter resolution, immediately after a reconfiguration point is reached.

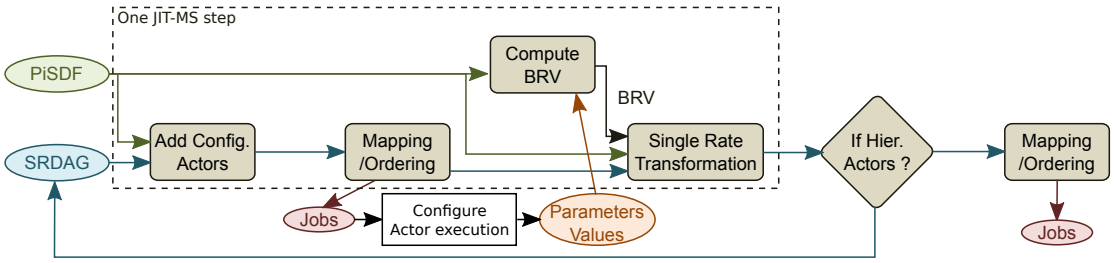


Figure 4.1 – *JIT-MS Transformation Flow*

The JIT-MS method is composed of multiple tasks (represented as boxes in Figure 4.1).

The JIT-MS method is iterative. The iterations occur over all hierarchical levels of a PiSDF graph to allow entire graph scheduling. At initialization, the SRDAG is populated with a unique top actor containing the top PiSDF graph. Then, each hierarchical actor present in the SRDAG is refined using its PiSDF description. The *refinement* of a hierarchical actor (that is, the execution of its internal graph) is called a *step*. A step is represented as the large dashed box in the Figure 4.1.

The *Mapping/Ordering* task boxes of Figure 4.1 correspond to the multicore Mapping/Ordering tasks from the SRDAG onto all PEs present in the platform. Multiple methods can be applied: numerous SDF scheduling heuristics exist that are relevant for multicore architectures [SB12]. A new Mapping/Ordering method focusing on many-core systems is introduced in Section 5.4. The Mapping/Ordering approach does not impact the JIT-MS method.

Next, the *BRV computation* and the *Single-Rate Transformation* are computed to reveal application parallelism. At the end of each step, the single-rate graph is populated with

new actors requiring execution. These two tasks are described in Sections 4.3 and 4.4 respectively.

4.3 PiSDF BRV Computation and Round Buffers FIFO behavior

On each step and for a given hierarchical actor, parameters can take different values. Once the parameter value have been determined by then configuration actors, the transformation of the PiSDF actor internal graph into an SRDAG subgraph is triggered. This subgraph is then merged into the global graph in which actors are then mapped and ordered.

This transformation has been explored in the literature for SDF graphs as described in Section 3.3.1.1.2. The first necessary operation is the BRV computation.

The BRV, also known as the SDF *repetitions vector*, represents the number of firings of each actor in a minimal periodic scheduling iteration for the graph. The BRV is a non-negative-integer vector that is indexed by the actors in the associated SDF graph.

However, some specificities of PiSDF require adaptations to the conventional repetitions vector computation process (from [LM87]).

In a PiSDF graph, certain data Fifos behave as Round Buffers (RBs) [DPN+13] to ensure hierarchical composability. Special RBs actors must be inserted in the SRDAG to implement this behavior. A RB is an actor with one input port and one output port. It transfers the input tokens to its output port when its production and consumption rates are equal. If its consumption rate is greater than its production rate, only the last tokens are transferred to the output. Conversely, if the production rate is greater than the consumption rate, input tokens are transferred several times to the output port until sufficient tokens are produced.

Such round buffer behavior is needed in the PiSDF model to enforce specific firing rules of the configuration actors and to provide hierarchical composability [DPN+13].

Fifos connected to the input or output interfaces of a hierarchical actor require RBs that ensure composability in the hierarchical specification. Fifos connecting configuration actors to other actors also require RBs to ensure that configuration actors fire only once per subgraph. Application designers using the PiSDF model of computation need to take such RB behavior into account during the development process.

This RB behavior and the special semantics of configuration actors require modifications to the conventional BRV computation process inherited from SDF graphs. RBs have no effect on repetition vectors, as they are instantiated based on the behavior of their parent graph , and not on the number of tokens they exchange. The BRV computation procedure for our JIT-MS scheduling method is specified in Algorithm 4.1.

To compute the BRV, the PiSDF graph is separated into two subgraphs. The first subgraph is composed of configuration actors only and is named CA. The second is the complement of the first, is composed of solely body actors, and is named BO. The BRV computation procedure is used at each JIT-MS step, and is thus executed several times within one iteration of the PiSDF top graph.

The BRV computation process specified in Algorithm 4.1 begins by computing rates (associated with the dataflow graphs Fifos) using previously computed parameter values. Then, the conventional BRV computation is applied across the subgraph BO.

The initial BRV is obtained by only considering this hierarchy level and this configuration flow. This BRV corresponds to the minimal execution scheme of this hierarchy level, thus ensuring that internal Fifos are neither starve nor accumulate tokens.

Algorithm 4.1: BRV computation procedure for PiSDF.

```
1  Procedure computeBRV()
2      Resolve rates of all Fifos using parameter values;
3      Compute BRV of BO ;
4      k ← 1;
5      for each {f ∈ F, f.src ∈ CA ∪ I, f.snk ∈ BO } do
6          k ← max (k, ceil (f.prod ÷ (f.cons * BRV[f.snk])));
7      endfor
8      for each {f ∈ F, e.src ∈ BO, f.snk ∈ I } do
9          k ← max (k, ceil (f.cons/(f.prod * BRV[f.src]));
10     endfor
11     BRV ← k * BRV;
```

However, the PiSDF MoC adds certain restrictions to ensure configuration flow and hierarchical consistency. Thus, this initial vector is multiplied by a scalar k to obtain an updated BRV. The factor k ensures that all tokens produced by configuration actors and input interfaces are consumed. To obtain this factor k, the algorithm iterates over all Fifos f that are connected to an interface or a configuration actor. Moreover, the factor k ensures that all tokens needed by output interfaces are produced. The resulting updated BRV is then compliant with the PiSDF MoC.

4.4 Single-Rate Transformation

Once the updated BRV has been computed, the single-rate graph of the current execution can be updated. This is performed by multiple instantiations of each actor in the PiSDF graph. The number of repetitions of each actor has already been determined in the previous BRV computation task.

Then, the actors of the single-rate graph need to be linked together using Fifos. Linking actors is equivalent to distributing the token output from each PiSDF Fifo to all targeted SRDAG actors.

However, each actor corresponds to a task and has a fixed number of inputs and outputs. So, the number of actor ports must remain stable for each single-rate actor.

In order to link the actors, new special actors that handle operations on tokens are instantiated in the graph. These actors are able to split a single Fifo into several Fifos, to group Fifos, duplicate tokens, create tokens or even delete tokens.

These *special actors* are described in Subsection 4.4.1, and the linking process is described in Subsection 4.4.2.

4.4.1 Special actors

In literature, special actors are generally employed for single-rate transformations [Pel10]. Their major task is to express non-direct token flows between actors. Examples of their usage include dividing one Fifo into several which are then connected to different sink actors and inversely, grouping a number of Fifos for a single actor. In certain cases, special actors are used to reduce the memory footprint of the overall Fifo bank. Special actors can also be used for efficient memory allocation of Fifos [DPNA15b, DPNA15a].

Numerous definitions of special actors can be found in literature. To avoid ambiguity, the five special actors used in this thesis are defined as follows:

- **Fork**: The *Fork* actor dispatches the tokens of a FIFO to several actors. The *Fork* special actor may have multiple output FIFOs. However, it always has a unique input FIFO. This actor does not create or ignore tokens, so the number of incoming tokens must match the sum of outputted tokens. The order of output FIFOs is important; it determines the destination of each token. As the behavior of a Fork actor does not modify token values, its execution may be implemented by a memory management mechanism, making tokens available to its output FIFOs.

- **Join**: The *Join* actor performs the reverse operation to the *Fork* special actor. It redirects tokens from multiple FIFOs to a unique FIFO. The *Join* actor may have multiple input FIFOs but only one output FIFO. Like the *Fork* actor, the *Join* actor does not create or discard tokens. The total number of tokens received from its input port must match the number of outputted tokens. The input FIFO order is important. It determines the order in which the tokens will appear on the output FIFO. As the behavior of a *Fork* actor does not modify token values, its execution may also be implemented by a memory management mechanism, forwarding input tokens in the right order to the output port.

- **Broadcast**: The *Broadcast* special actor is used to duplicate tokens from a single FIFO to several FIFOs. The token rate of the input FIFO and the sum of the token rates of all output FIFOs are equal. For this actor, the order of output FIFOs is unimportant. The behavior does not modify token values and as input tokens of an actor are considered read-only. The execution of this actor is also memory managed, forwarding the same data to several FIFOs.

- **Init**: The *Init* special actor is used to create default tokens corresponding to delays on the dataflow graph. The value of these tokens is set to zero. The *Init* special actor is used to initialize delay tokens during the single-rate transformation. SRDAG optimizations are possible when this special actor is present as the actor can be removed without consequence when its output tokens (that is, the delay tokens) are discarded.

- **End**: The *End* special actor is used to discard unused tokens. In certain cases, this special actor is mandatory to eliminate the situation where FIFOs contain tokens at the end of the execution.

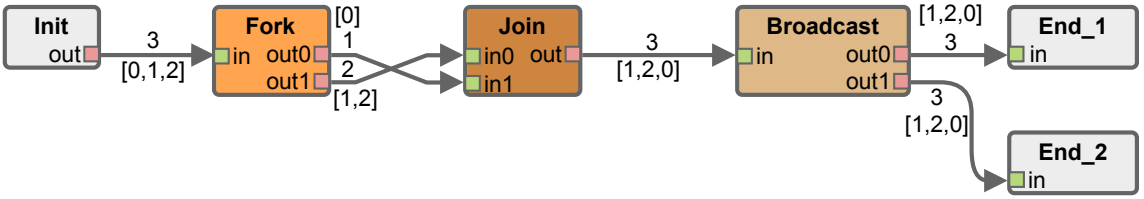


Figure 4.2 – *Example of Special Actors*

Figure 4.2 describes the behavior of the special actors. The token order into FIFOs is represented by the numbers in brackets; this allows the changes in token order in FIFOs to rapidly noted. These five special actors are sufficient to implement all single-rate transformation patterns of a PiSDF FIFO.

4.4.2 Single-Rate Transformation Patterns

Given the final computed BRV, the single-rate transformation of each subgraph can be processed. For this task, each Fifo in the PiSDF graph is managed separately.

The first step is to integrate each actor as many times as it may fire. This is a straightforward processing step. The next step is to link these actors together though the use of special actors and single rate Fifos.

This subsection will introduce different patterns that can be applied to this single-rate transformation of a Fifo. The problem has already been widely explored in literature [PPW+09, ZPBF12, LHGQ11]. The processing method presented in this thesis is derived from [PBL95], which was based on the SDF MoC.

This new method handles the different cases introduced by the PiSDF MoC. It also focuses on conserving the number of ports for each single-rate actor. The method then uses special actors to handle token management.

The PiSDF MoC requires four different patterns of single-rate transformations: the default pattern, the delayed pattern, the round buffered input and the round buffered output. These four patterns are detailed in the following subsections.

4.4.2.1 Generic Pattern and Single-Rate Transformation Algorithm

In literature, the single-rate transformation is generally considered straightforward. The entire single-rate transformation is based on the SDF Fifo transformation. Each SDF Fifo is split and then connected to several inputs (called sources) and output ports (called sinks). The pattern used for this transformation is illustrated in Figure 4.3. This pattern is defined to elucidate the single-rate transformation problem, and to allow the derivation of a generic algorithm for the single-rate transformation of a single Fifo.

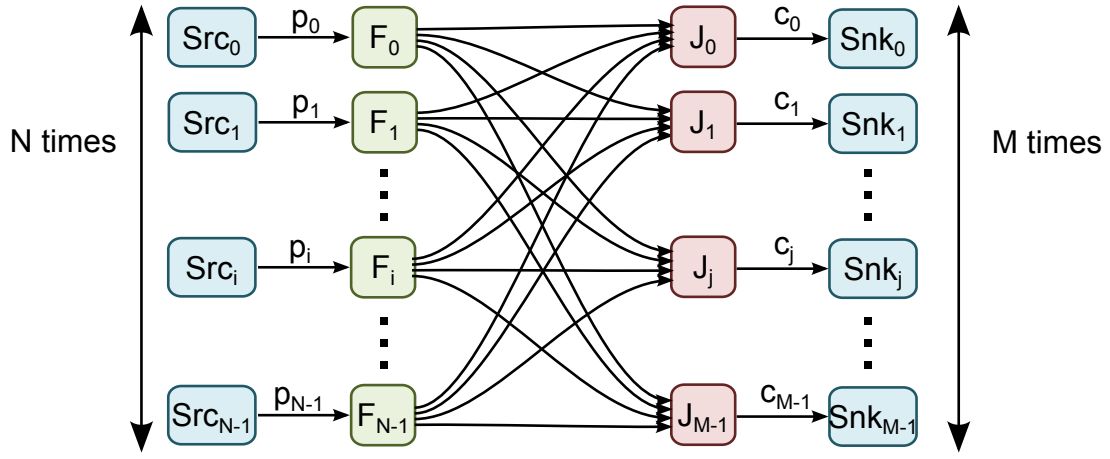


Figure 4.3 – *Fork/Join general pattern*

In Figure 4.3, N and M are defined as the number of source and sink actors respectively which require connection. N and M are closely related to R and K respectively, which are the corresponding BRV values of actors.

The p_i and c_i values correspond to the amount of tokens produced by the source actors and consumed by the sink actors, respectively for one firing. These values may not be equal.

F_i and J_i actors are *Fork* and *Join* actors respectively. These actors have the role of distributing data among the appropriate sink actors.

The problem posed by single-rate transformation is the determination o the amount of tokens outputted by *Fork* actors and received by *Join* actors. The production of the k-th output port of the i-th *Fork* actor is named $F_{i,k}$. Respectively, the consumption k-th input port of the j-th *Join* actor is named $J_{j,k}$.

Theorem 4.4.2.1 (Fork/Join Value)

$$\forall i \in [0, N-1], \forall j \in [0, M-1], F_{i,j} := J_{j,i} := FJ_{i,j}$$

Proof. Since the graph is a single-rate graph, the production and consumption rates for a given FIFO must be equal. This leads to the previous proposition where $FJ_{i,j}$ is a notation reflecting this equality. □

Given theorem 4.4.2.1, the problem can be reduced to the resolution of $FJ_{i,j}$ for $i \in [0, N-1]$ and $j \in [0, M-1]$. This corresponds to the determination of the following Fork/Join matrix:

Definition 4.4.2.2 (Fork/Join Matrix)
Fork production rates and Join consumption rates can be represented as a 2D matrix called Fork/Join matrix:

$$FJ_{N,M} = \begin{array}{c} \\ F_0 \\ F_1 \\ \vdots \\ F_i \\ \vdots \\ F_{N-1} \end{array} \begin{array}{c} \begin{array}{cccccc} J_0 & J_1 & \cdots & J_j & \cdots & J_{M-1} \end{array} \\ \begin{pmatrix} FJ_{0,0} & FJ_{0,1} & \cdots & FJ_{0,j} & \cdots & FJ_{0,M-1} \\ FJ_{1,0} & FJ_{1,1} & \cdots & FJ_{1,j} & \cdots & FJ_{1,M-1} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ FJ_{i,0} & FJ_{i,1} & \cdots & FJ_{i,j} & \cdots & FJ_{i,M-1} \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ FJ_{N-1,0} & FJ_{N-1,1} & \cdots & FJ_{N-1,j} & \cdots & FJ_{N-1,M-1} \end{pmatrix} \end{array}$$

The single-rate transformation is then equivalent to the Fork/Join matrix resolution. Algorithm 4.2 then describes the computation of this matrix. The algorithm input is derived from the source and sink count as well as the production and consumption rates associated with each actor.

The algorithm begins with the initialization of the variables. The resulting Fork/Join matrix is filled with zeros and indexes i and j are set to zero. Then, *tokenSrc* and *tokenSnk* are set with the initial production and consumption rates. *tokenSrc* and *tokenSnk* variables are used to keep track of the remaining tokens at each step.

Next, the while loop iterates until the whole Fork/Join matrix is completed. The loop fills each element of the matrix with the token rates to which it corresponds by the use of the *rest* variable. At each step, the current *Fork* or *Join* actor or both will starve of tokens, leading to the creation of a new *Fork* or *Join* actor. The *rest* variable then stores the amount of tokens transmitted for use in the current index couple (i, j).

If necessary, the *tokenSrc* and *tokenSnk* are subsequently updated with the rates corresponding to the next source or sink actor. The i and j are then also updated depending on whether the *Fork* or *Join* actors are starved. Finally, the whole Fork/Join matrix is completed. Any unchanged elements will be still equal to zero.

This algorithm is applied on the generic pattern described in Figure 4.3. The remainder of this subsection will review the four possible cases in PiSDF MoC and their expression using this generic pattern.

4.4.2.2 Default Pattern

A default pattern is composed of a non-delayed Fifo with no connections, which is subject to a round buffer effect. Such a Fifo is neither connected to an interface nor to a configuration actor. The default pattern of the transformation can be found at Figure 4.4.

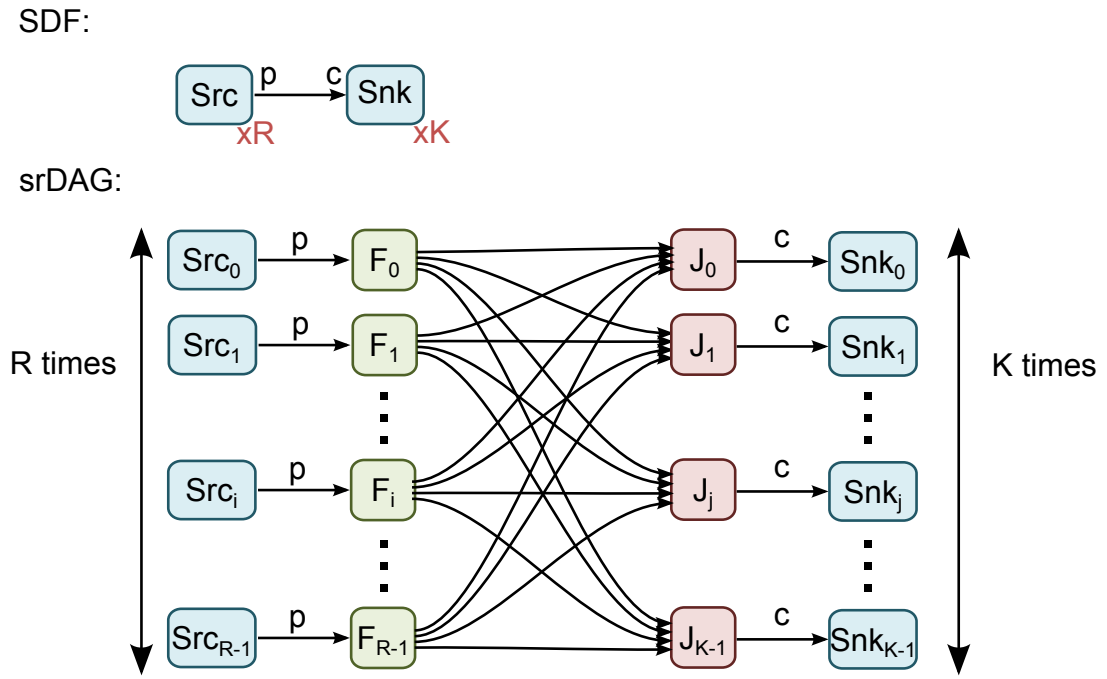


Figure 4.4 – *Fork/Join default pattern for single-rate transformation*

Property 4.4.2.3 (Default Pattern Properties)
By definition of the default pattern, we have:

$$\begin{aligned} N &= R \\ M &= K \\ \forall i \in [0, N-1], p_i &= p \\ \forall j \in [0, M-1], c_j &= c \end{aligned}$$

With Property 4.4.2.3, the generic algorithm 4.2 can be applied. The indexes N and M are directly derived from the BRV computation, and both the production and consumption rates are equal to the production and consumption respectively of the corresponding PiSDF Fifo.

4.4.2.3 Delayed FIFO Pattern

When a Fifo has a non-zero delay, this delay must be taken into account for the single-rate transformation. This is an important pattern for the single-rate transformation.

In PiSDF, delays can be used for different purposes, such as feedback loops. If Fifos of a graph make a loop of actors, the SDF MoC schedulability is only respected when the tokens necessary to launch one execution loop are set into the delay of one of the Fifos. Since there is no flattening between hierarchical levels of a full PiSDF graph, data remaining

Algorithm 4.2: Fill the Fork/Join Matrix

Input: N Number of sources actors
Input: M Number of sinks actors
Input: $p[N]$ Production of sources actors
Input: $c[M]$ Consumption of sinks actors
Output: $FJ[N][M]$ The computed Fork/Join Matrix

```
1  Variable: integer tokenSrc, integer tokenSnk;
2  Variable: integer i, integer j, integer rest;
3  /* Initialization */
4  FJ[N][M] = 0;
5  tokenSrc = p[0];
6  tokenSnk = c[0];
7  i = j = 0;
8  /* Fill the Fork/Join Matrix */
9  while i = N && j = M do
10     rest = min (tokenSrc, tokenSnk);
11     FJ[i][j] = rest;
12     tokenSrc = tokenSrc − rest;
13     tokenSnk = tokenSnk − rest;
14     if tokenSrc = 0 then
15         i = i + 1;
16         tokenSrc = p[i];
17     end
18     if tokenSnk = 0 then
19         j = j + 1;
20         tokenSnk = c[j];
21     end
22 end
```

at the end of the PiSDF graph execution in a delayed FIFO is discarded at the end of the graph hierarchy level execution. To retain tokens of a FIFO for the next iteration, these data token must be outputted through an interface. These tokens must then be re-injected into an input interface in the upper hierarchy. These actions consequently move the delay to the upper hierarchy level.

For the single-rate transformation, a delay on a FIFO will then be initialized using an *Init* actor. At the end, remaining tokens will be discarded using an *End* actor. Figure 4.5 illustrates the pattern of this transformation.

Since there are two new single-rate actors, the Fork/Join matrix size is different from that of the default case:

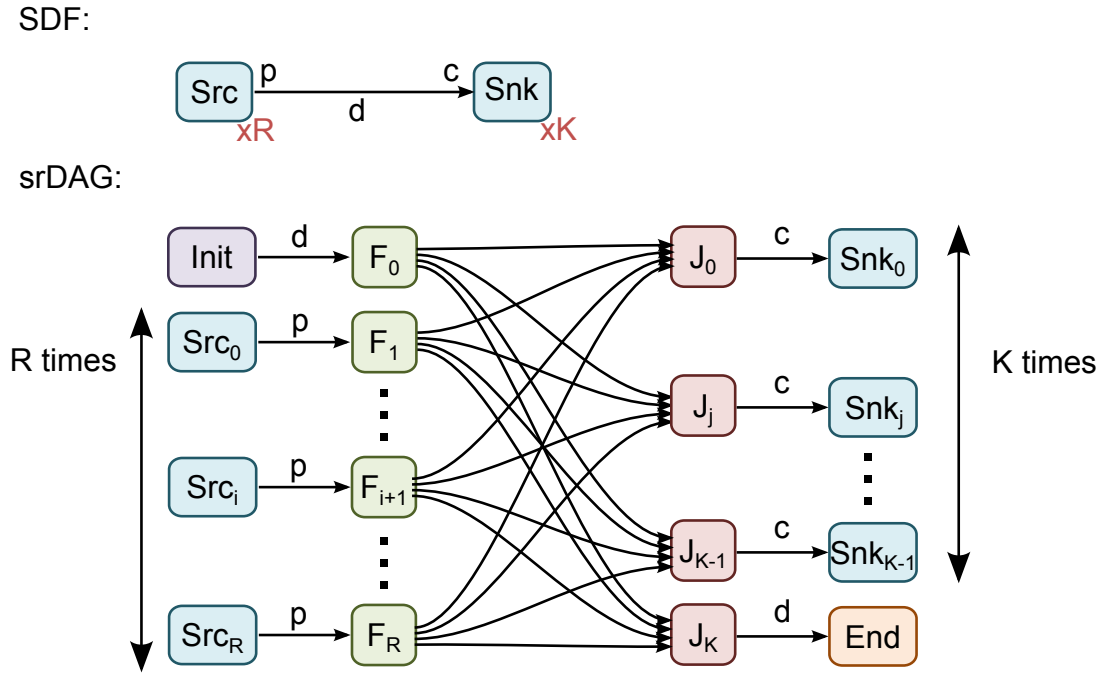


Figure 4.5 – *Delayed FIFO pattern for single-rate transformation*

Property 4.4.2.4 (Delayed Pattern Properties)
From the definition of the delayed pattern, we have:

$$N = R + 1$$
$$M = K + 1$$
$$\forall i \in [0, N-1], p_i = \begin{cases} d \text{ if } i = 0 \\ p \text{ otherwise} \end{cases}$$
$$\forall j \in [0, M-1], c_j = \begin{cases} d \text{ if } j = M-1 \\ c \text{ otherwise} \end{cases}$$

Then, though use of the Algorithm 4.2, the Fork/Join Matrix can be computed.

4.4.2.4 Round Buffered Source Pattern

Another distinct pattern occurs when the FIFO source is a round buffered source. A round buffered source exists when a FIFO is connected to an input interface or a configuration actor. The round buffered source pattern is illustrated in Figure 4.6

In this case, the amount of tokens may or may not be increased depending on the token consumption of the sink actor. In case of token duplication, a *Broadcast* actor is used as shown in Figure 4.6. However, the number of output FIFOs of the *Broadcast* actor need to be determined. From the definition of the Broadcast actor, each output FIFO has a token rate of p.

Theorem 4.4.2.5 (*Broadcast* output in Round buffered source pattern)
In the round buffered source pattern, the Broadcast actor generated possesses b output

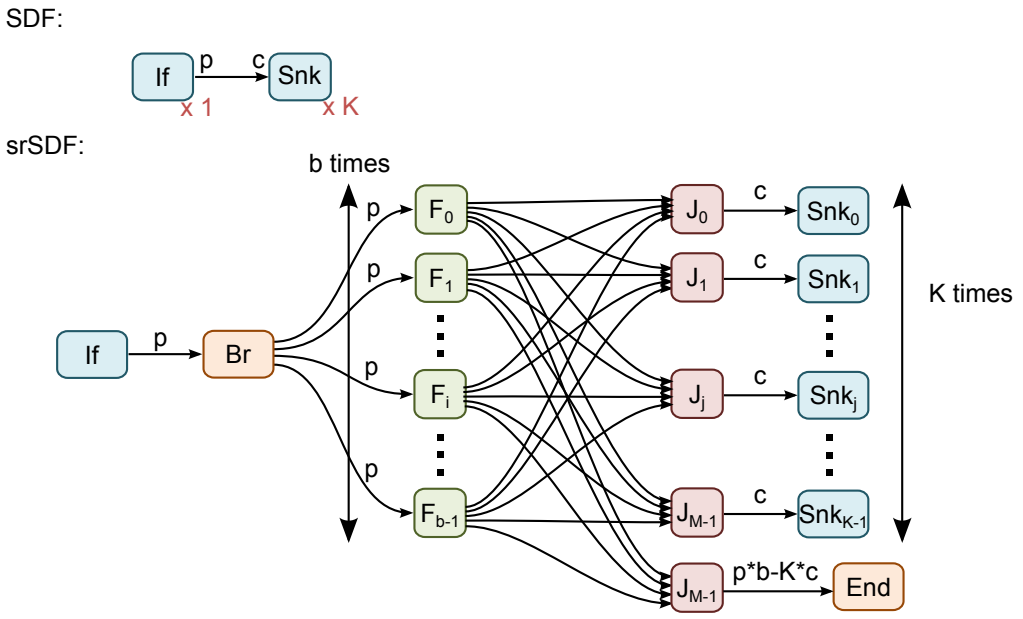


Figure 4.6 – *Round buffered source pattern for single-rate transformation*

FIFOs with:

$$b = \left\lceil \frac{K * c}{p} \right\rceil$$

Proof. Since the sink actor will consume $K * c$ tokens, where K is the repetition number of the sink, the minimum number of tokens needed to satisfy this consumption is the smallest b integer that respects:

$$\begin{aligned} b * p &\geq K * c \\ b &\geq \frac{K * c}{p} \\ b &= \left\lceil \frac{K * c}{p} \right\rceil \end{aligned}$$

□

The following properties can then be defined:

Property 4.4.2.6 (Round Buffered Source Pattern Properties)
Though the definition of the delayed pattern, we have:

$$\begin{aligned} N &= b \\ M &= K + 1 \\ \forall i \in [0, N-1], p_i &= p \\ \forall j \in [0, M-1], c_j &= \begin{cases} p * b - K * c \text{ if } j = M-1 \\ c \text{ otherwise} \end{cases} \end{aligned}$$

Finally, the Fork/Join Matrix can be computed using generic Algorithm 4.2, enabling the single rate transformation of this FIFO.

4.4.2.5 Round Buffered Sink Pattern

The last pattern occurs when the Fifo sink is a round buffered sink. A round buffered sink is only induced by an output interface in PiSDF MoC. The transformation pattern is illustrated in Figure 4.7.

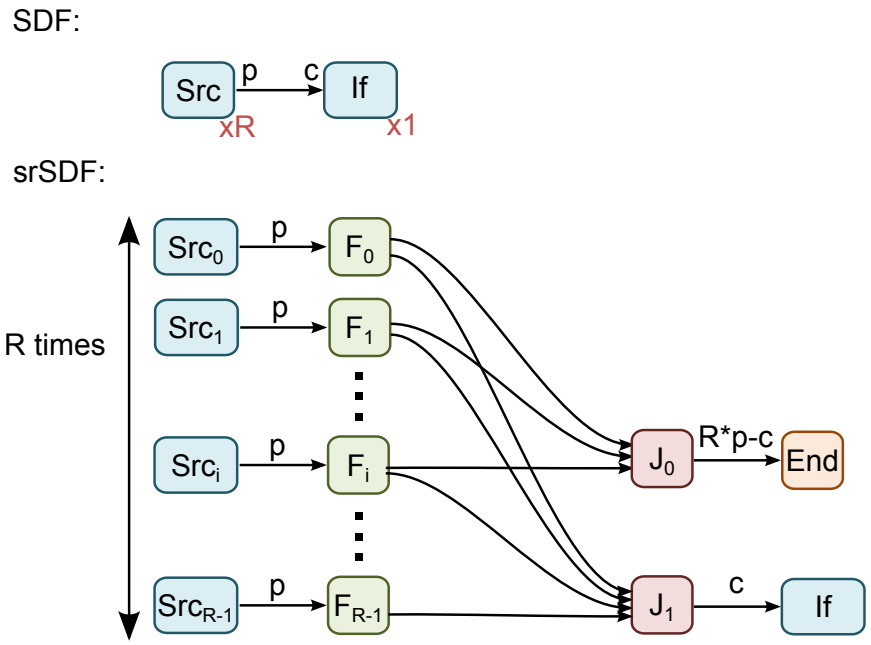


Figure 4.7 – *Round buffered Sink Pattern for Single-Rate Transformation*

In this particular case, the PiSDF MoC specifies that the final tokens are transmitted to the upper hierarchy. In this case, only two *Join* actors are created regardless of the BRV values. One *Join* actor retrieves the useful tokens and inversely, the other *Join* actor collects tokens to discard. Tokens produced are then either passed to an *End* actor or the output interface.

The resolution of the pattern may then fit the generic pattern when:

Property 4.4.2.7 (Round Buffered Sink Pattern Properties)
Through the definition of the delayed pattern, we have:

$$N = R$$
$$M = 2$$
$$\forall i \in [0, N-1], p_i = p$$
$$c_j = \begin{cases} R * p - c \ \textit{if} \ j = 0 \\ c \ \textit{otherwise} \end{cases}$$

Finally, the Fork/Join Matrix can be computed with the generic Algorithm 4.2 and the single rate transformation of this Fifo can be performed.

In conclusion, all single-rate transformation patterns used by the PiSDF can be computed using the given algorithm. This iterative method allows any possible PiSDF MoC case to be managed and the Fork/Join matrix of the given Fifo to be derived. After the generation of this matrix, the source and sink single-rate actors may be easily connected.

4.5 Multi-Step scheduling of a PiSDF graph

As presented in Figure 4.1, the transformation flow of a single PiSDF graph is iterative. The transformation iterates as required to resolve all hierarchy levels.

The JIT-MS method is based on the operational semantics of PiSDF that is described in [DPN+13]. Each hierarchical actor firing must be processed in two steps. The first stage fires the configuration actors of the subgraph for each hierarchical actor. These configuration actors produce the necessary parameters to resolve the remainder of the subgraph. When all parameters are solved at a hierarchical level, a schedule can be computed for the rest of the current hierarchy level. This operational semantic can be implemented in single-core systems.

In a multicore system, the scheduling method has to extract the parallelism of the application and then send jobs to multiple PEs. With a static MoC such as SDF, the full single-rate graph can be entirely derived at compile-time. The difficulty of the PiSDF scheduling process lies in efficient actor scheduling without knowledge of the entire resolved graph. The complete SRDAG is known only when all configuration actors have been executed.

The top element of the hierarchical graph is a unique actor named *top actor* with an infinite lifespan, and is never connected to either input Fifo or output Fifo. In the remainder of this thesis, we distinguish between *hierarchical actors* that contain a subgraph and *atomic actors* that contain a code executable on a single PE.

The JIT-MS scheduling method proceeds in multiple steps, with each using scheduling step thus unveiling a new portion of SRDAG. Every hierarchical actor firing can result in up to two scheduling steps: one to schedule configuration actors and another to schedule the body actors of the graph.

Once an SRDAG has been generated, it is then analyzed to extract the maximum parallelism while still respecting actor granularity. The JIT-MS scheduling can then map and order both actors and communications before firing them onto the platform. Newly instantiated hierarchical actors are added to the SRDAG. This process can be repeated until all hierarchy levels have been explored.

Multi-step scheduling leads to an SRDAG containing a mix of actors which have already been fired and those that have not yet been fired. To keep track of actors already executed in the SRDAG, each actor is tagged with a flag representing its execution state. This flag can take three values:

- *Run (R)*: the actor has already been fired in a previous step.
- *Not Executable (N)*: the actor cannot be executed because its parameters are not yet solved.
- *Executable (E)*: the actor can be executed in the next firing step, as all actor parameters are resolved and its predecessors are tagged as executable or run.

A flag update procedure is processed before each static scheduling step. It updates all flags as a function of SRDAG topology and actor types. The flag update procedure is described in Algorithm 4.3.

The algorithm assumes that if a configuration actor is present in the SRDAG and is not already in the *Run* state, this actor has been newly added in the current scheduling step, so it must be executable (line 4). The SRDAG body actors are constrained to be flagged as executable if and only if all predecessor actors are run or executable (line 6).

Algorithm 4.3: updateSrDAG

```
1  Procedure updateSrDAG(inputDag)
2      for each {a1 ∈ inputDag, a1.flag ≠ R} do
3          if a1 ∈ CA then
4              a1.flag ← E;
5          else if ∃a2 ∈ a1.predecessors, a2.flag = N then
6              a1.flag ← N;
7          else
8              a1.flag ← E;
9          end
10     endfor
```

The complete procedure of JIT-MS scheduling is shown in Algorithm 4.4. The global SRDAG used as an intermediate representation is called the *execution graph*. This graph describes the entire execution of one application iteration. First, the unique *top actor* of the PiSDF algorithm is placed in the empty *execution graph* (line 2). This top actor represents one application iteration.

A PiSDF graph software Fifo is used to asynchronously transfer the subgraphs that are ready for the parameter resolution to the next scheduling phase. This Fifo is named *pisdfStack* in the algorithm and must be cleared at the beginning of the graph iteration (line 3).

After initialization, the algorithm enters a primary while loop (line 4) which computes scheduling steps until there are no longer hierarchical actors in the execution graph. A single scheduling step is divided in 3 parts: *graph configuration*, *actor execution* and *graph resolution*.

The first part (line 5 to 18) replaces each executable hierarchical actor of the *execution graph* by its configuration actors. For the case of a hierarchical actor which does not contain a configuration actor the whole subgraph can be fired and the result replaces the hierarchical actor.

The second part (lines 19 to 20) maps, orders and fires executable configuration actors. Parameter value are subsequently received from the configuration actor executions. The following scheduling stages can have different policies:

1. map, order and fire all executable actors with or without a priority on configuration actors.

2. map, order and fire only actors which lead to configuration actors.

3. map, order and fire actors which lead to configuration actors in priority and add the result of the map, order and fire of executable actors to any remaining space. Non executed actors are retained for the next scheduling step.

The focus of this thesis is the map, order and firing of actors exclusively which lead to configuration actors . Exploration of the first and third options is envisaged for future work.

The third part (lines 21 to 26) corresponds to graph resolution. It is here that, the parameters resolved in the previous stage are used to resolve the graph parameters of each hierarchical step. The BRV of each subgraph is computed and the execution can be completed for these new actors.

Algorithm 4.4: Multi-Step Algorithm

```
1  Procedure MultiStep()
2      Add topActor in execGraph;
3      Clear pisdfStack;
4      while {∃h ∈ execGraph, h.subgraph ≠ ∅} do
5          while {∃h ∈ execGraph, h.subgraph ≠ ∅} do
6              curPisdf ← h.subgraph;
7              if CA ≠ {∅} then
8                  Replace h with RBs in execGraph;
9                  Put CA in execGraph;
10                 Add RBs between CA and BA;
11                 push curPisdf → pisdfStack;
12             end
13             else
14                 computeBRV(curPisdf);
15                 Add single rate BA in execGraph;
16             end
17             updateSrDAG(execGraph);
18         end
19         Schedule & fire {a ∈ execGraph, a.flag = E};
20         Wait All parameter values;
21         while pisdfStack is not empty do
22             pop pisdfStack → curPisdf;
23             computeBRV(curPisdf);
24             Add single rate BA in execGraph;
25         end
26         updateSrDAG(execGraph);
27     end
28     Schedule & fire {a ∈ execGraph, a.flag = E};
```

At the end of the algorithm (line 28), no more hierarchical actors are present in the graph. It is at this point that a final phase of mapping, ordering and firing of executable actors must be performed to execute all remaining actors.

These steps describe the entire JIT-MS scheduling method employed in this thesis. This method will be benchmarked and applied to real applications in Chapter 7.

4.6 Conclusion

The JIT-MS scheduling method allows multicore scheduling of PiSDF graphs to be performed. This scheduling method is processed at runtime and permit full exploitation of the locally static regions of the PiSDF MoC to improve scheduling efficiency.

For efficient scheduling based on the PiSDF MoC, the JIT-MS method employs an intermediate graph called SRDAG. This graph can be considered as an precedence which reveals interactions between actor firings of the entire execution. By the use of this SRDAG, the JIT-MS method can exploit maximal parallelism between actors through the use of a Mapping/Ordering heuristic.

One of the most crucial scheduling tasks in the JIT-MS is the single-rate transformation. This task is processed immediately after parameters of a hierarchy level are evaluated. It generates the SRDAG from the PiSDF graph. It is the same task used in SDF multicore scheduling but it has been extended in this thesis to handle PiSDF graphs. In fact, the configuration actor firing rules and the interface hierarchy require special attention, leading the single-rate transformation to create new actors in the graph called "special actors" that handle token distribution.

CHAPTER 5

Improving the PiSDF Multicore Scheduling

5.1 Introduction

Chapter 4 introduced the fundamentals of the PiSDF multicore scheduling technique. There are a number of possibilities for optimizing this technique for better performance at runtime. Each optimization targets a different subpart of the JIT-MS method, and several enhancements will be detailed in this chapter.

First, an extension of the PiSDF MoC is proposed to ameliorate scheduling performance and to add a new feature for PiSDF developers. This extension is defined in Section 5.2

Post-single-rate transformation optimizations on the SRDAG are also proposed to reduce the graph complexity. This lower complexity is achieved by reducing either the memory footprint or the Mapping/Ordering processing time or the potential parallelism of the application. These optimizations are detailed in Section 5.3.

The final optimization is the use of a new Mapping/Ordering algorithm. As discussed previously, the JIT-MS method is independent of the algorithm used for the Mapping/Ordering task. Any one of the algorithms proposed in [SSKH13] may be employed. One of the most famous Mapping/Ordering algorithm is the list scheduling algorithm. The list scheduling algorithm has a number of variants: for experiments presented in this thesis, the Modified Critical Path (MCP) version is used [WG87]. However, for an architecture with large PE count list scheduling is time consuming. For the situation of a many-core platform with hundreds of PEs, exploration of a new Mapping/Ordering algorithm is proposed as future work in Section 5.4.

5.2 Proposed PiSDF extension

This section presents an extension of the original PiSDF MoC which focuses on initial delay values on FIFOs. In Figure 5.1, the feedback loop delay around actor B allows the transmission of data tokens generated by the previous firing of actor B to the current firing. Clearly, this behavior is not possible for the first firing.

As described in Section 4.4, the usual practice is to add a new actor to the SRDAG. This new actor is a special actor called *Init* (as defined in Section 4.4.1). This actor generates d tokens as specified in the initial token value of the delay. However, the initial value of

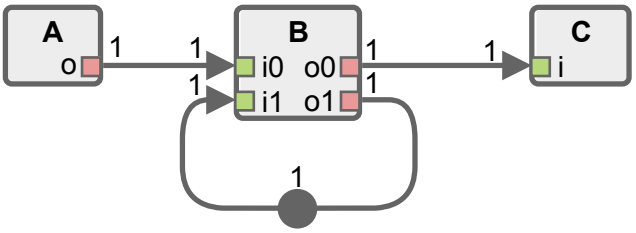


Figure 5.1 – *PiSDF delay question*

the token is not specified in the PiSDF MoC definition [DPN+13], meaning that a token may be zero, one or even an uninitialized value.

It is thus important to understand whether the values of the initial tokens are used for the first firing of actor B. If this is the case, this may mean that resulting execution depends on these randomly generated tokens. Moreover, if the actor B is hierarchical and the values generated from the initial token then influence a parameter, the consequence may be a non-schedulable graph. Thus, it is to control the initial value of the delays that the PiSDF extension is proposed.

5.2.1 Problem encountered with original PiSDF MoC

The extension proposed with PiSDF MoC is introduced using an example. This example is a common pattern in application design. It contains a *for* loop where an action is repeated a parameterized number of times on the same data.

This example is presented using a C sample code in Listing 5.1. The use of a ping pong buffer to reduce memory footprint may be noted.

```
1 int main(){
2         char start, end;
3         start = A();
4         for(int i=0; i<N; i++){
5                 end = B(start);
6                 start = end;
7         }
8         C(start);
9 }
```

Listing 5.1 – *example C code of parameterized iteration*

The single-rate dataflow graph corresponding to this example is shown in Figure 5.2. From the dataflow graph, the actor B is repeated N times. That is, B applies the computation N times on the data.

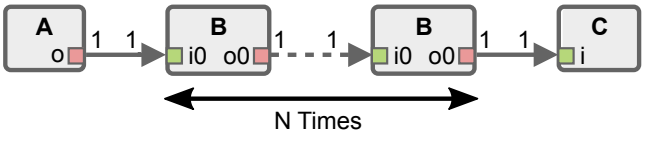


Figure 5.2 – *Single-rate dataflow graph of the application*

There are multiple ways to represent this pattern in PiSDF MoC. The first is displayed in Figure 5.3.

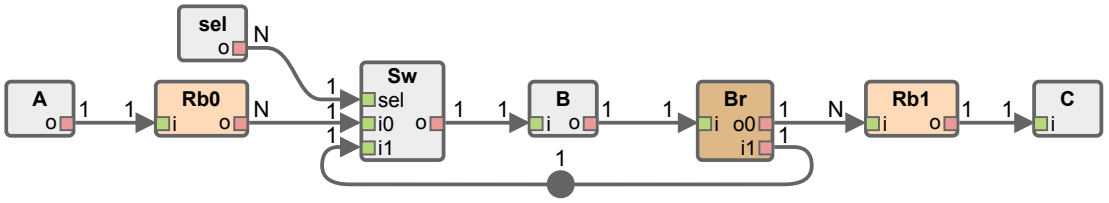


Figure 5.3 – *Flattened PiSDF representation of the example*

This representation uses multiple special actors to describe the application. The actor B is repeated N times, as implied by the $\frac{N}{1}$ ratio FIFO from the *sel* (select) actor to the Sw (Switch) actor. The Sw actor selects the input data to feed the actor B. This actor simply uses the input token value from its *sel* port to determine which token to copy to its output. In this PiSDF graph, tokens are chosen from actor A or from the feedback loop.

To ensure that actors A and C are executed only once, two *RoundBuffer* actors have been inserted: one will duplicate the token outputted by A and the other will select only the last token and send it to C. A *Broadcast* actor has also been added with the role of duplicating each token, and then forward it to either the feedback loop or the second *RoundBuffer*.

The *sel* actor determines which data token is sent to actor B. For our application, the *sel* actor should initially output a 0 so that the Sw actor accepts data first from actor A. Then the *sel* actor sends $N-1$ tokens with value equal to 1 so that the Sw actor will select tokens from the feedback loop, thus recreating the chain of Figure 5.2.

It can be seen from this case study, that there are many actors in the PiSDF graph in addition to the initial actors A, B and C. Some of these actors are special actors which can be handled by the runtime itself but there is consequently a cost in terms of computation at runtime.

With the PiSDF MoC, it is possible to describe the application from Figure 5.2 using two hierarchy levels. Figure 5.4 represents the corresponding two level PiSDF graph.

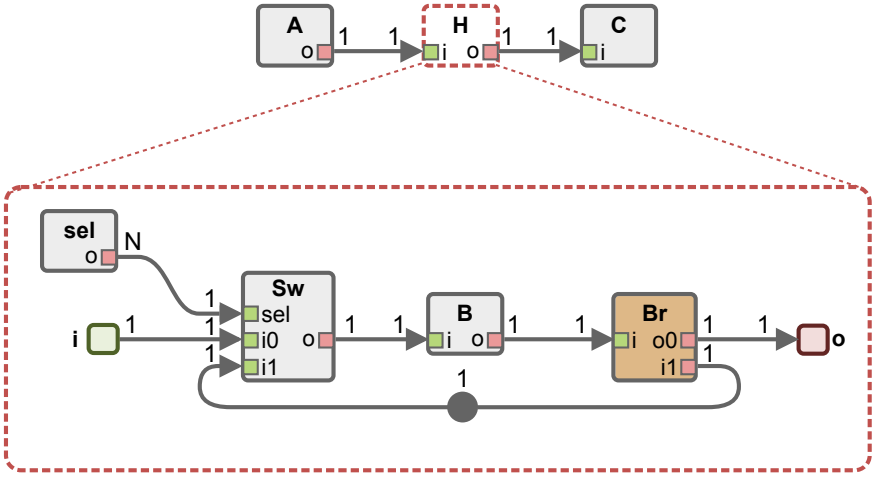


Figure 5.4 – *Hierarchical PiSDF representation of the example of Figure 5.2*

In this representation, the hierarchy has been used to remove the two *RoundBuffers* employed in Figure 5.3. As interfaces behave like *RoundBuffers*, the behavior of the whole graph is equivalent to the previous PiSDF graph. The top graph ensures that actors A and C are fired only once and that the *sel* actor ensures that actor B is fired N times.

However, the Sw actor is still present and this increase the number of single-rate actors for scheduling by N.

The proposed PiSDF extension allows the sources of initial tokens of delays to be defined within the graph. This results in two advantages. Firstly, it allows the PiSDF programmer to make concise graphs for the application. Secondly, it reduces the scheduling complexity by :

- Lowering the single-rate transformation complexity.
- Reducing the Mapping/Ordering complexity by decreasing the number of single-rate actors.

5.2.2 Proposed Extension

This work has chosen to explore the solution of allowing in-graph definition of initial delay tokens. These tokens are not permitted to remain undefined or to be set to zero as is the case with usual implementations. The introduction of the extension allows the delay tokens to be set by another actor. This extension is possible only through the addition of an input data port to the delay representation.

However, there are multiple restrictions to retain the schedulability properties of the PiSDF graph:

- The actor that set the delay, called the *setter*, must not be present in the same scheduling step as the delayed Fifo. Consequently, the setter can be either an interface or a configuration actor. This guarantees that interdependencies between the setter and the delayed Fifo will not lead to deadlocks.
- Token production of the setter must match the delay value.

In the example of Figure 5.2, the PiSDF graph becomes that of the representation in Figure 5.5.

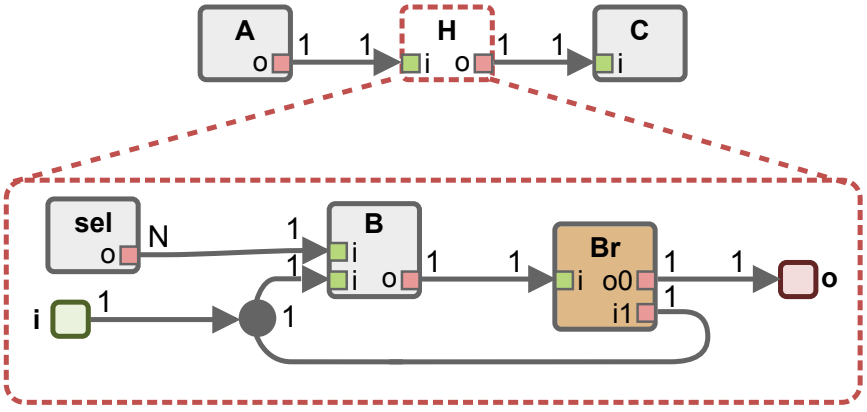


Figure 5.5 – *PiSDF extension representation of the example of Figure 5.2*

For Figure 5.5, the *Sw* actor of Figure 5.4 has been removed by using the proposed extension. This has been performed by directly feeding the tokens from the input data interface into the feedback delay. These tokens are then used for the first firing of actor B.

However, the *sel* actor has been added to ensure that actor B is fired N times. A unused input port has been added to actor B to connect with this *sel* actor and to force the repetition of actor B to be equal to N.

Finally, by removing the *Sw* actor, the proposed extension reduces the resulting complexity of the PiSDF graph. Furthermore, this method also lowers the complexity of the scheduling, as shown in the following section.

5.2.3 Impact of extension on the scheduling

In the classic PiSDF single-rate transformation, two new special actors called *Init* and *End* are used to transform delays. The *Init* actor is used to produce the initial tokens of the delay.

The purpose of the proposed PiSDF extension is to allow an actor of the PiSDF graph to set the initial value of the graph. When the limitations described in Section 5.2.2 are satisfied, the setter actor can simply replace the *Init* actor in the SRDAG. It is then not necessary to introduce additional actors in the SRDAG. In this case, only a simple connection between the setter and the first firing of the actor is required, as represented in Figure 5.6.

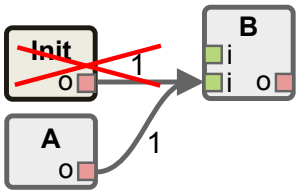


Figure 5.6 – *Impact of the proposed PiSDF extension on the single-rate transformation*

Finally, the extension reduces the number of actors in the final SRDAG (in our example, N *Sw* actors and 1 *Init* actor). Thus, the extension limits the Mapping/Ordering complexity, which is proven to be NP-complete in [PBL95].

5.3 Graph optimizations

5.3.1 Motivations

This section introduces six possible optimization patterns that can be applied at runtime to reduce the SRDAG complexity. The majority of these optimizations are fairly trivial to perform but still improve the scheduling performance. Since there are fewer actors to execute, the Mapping/Ordering task requires less computation time.

It can also be noted that reducing the number of Fifo leads to less interdependencies between actors of the SRDAG. This can unveil new task parallelism and improve the scheduling method performance.

Despite their small size, execute actors produce an overhead. This is due to actor communications and synchronizations.

Finally, fewer Fifos in a scheduled SRDAG, lead to a lower memory footprint (in shared memory platforms) or fewer communications (for distributed memory).

Hence, for the above reasons, reducing the SRDAG complexity is an excellent way to improve the scheduling performance. In the following sections, the causes of unoptimized graphs are explored and optimization patterns to circumvent these problems are then introduced.

5.3.2 Sources of sub-optimized graphs

If the single-rate transformation method of the Section 4.4 is used, special actors are created as many times as needed. This leads to an optimized SRDAG for each original PiSDF Fifo. However, when the graph is regarded as a whole, it is clear that certain simple optimizations can be performed to improve the overall graph performance.

A sub-optimized graph results from either : hierarchy or special actors in the PiSDF graph. These two areas will now be explored.

5.3.2.1 Hierarchy

Hierarchy flattening can result in sub-optimized graphs in SRDAG. An example is given in Figure 5.7. In this example, actors A and B are not at the same level of hierarchy, and so the Fifo between them is handled separately for the top and the lower hierarchy.

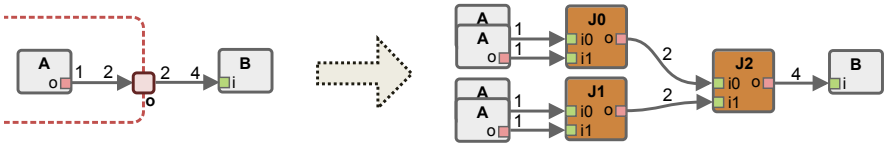


Figure 5.7 – *Sub-optimal SRDAG induced by the presence of hierarchy actors*

The first subgraph scheduled is that of the upper level which is connected to actor B. Two tokens are outputted from the hierarchical actor (fired twice for this example), so, a *Join* actor ($J2$) is created, as expected.

On lower hierarchy level, as the hierarchical actor is executed twice, two join actors ($J0$ and $J1$) are then created according to the single-rate transformation of each subgraph.

Once connected, the result is a sub-optimized SRDAG with cascaded *Join* actors. To ameliorate this situation, these three *Join* actors may be condensed into a single *Join* actor with four inputs.

5.3.2.2 Special Actors

The same behavior may occur on a single level PiSDF graph which contains embedded special actors. An example of a sub-optimized graph is given in Figure 5.8.

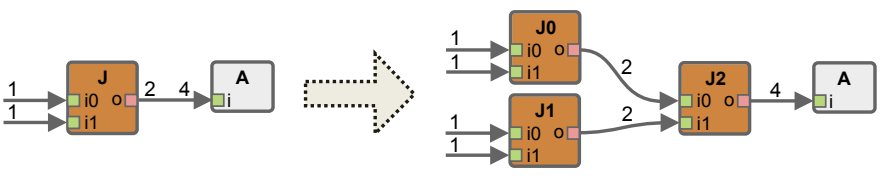


Figure 5.8 – *Sub-optimal SRDAG induced by special actors in PiSDF graph*

As shown in the figure, the *Join* actor has a basic repetition of two. Following the single-rate transformation, two *Join* actors are created in the SRDAG. Since the actor A is fired only once, a new *Join* actor is then required.

As with the above example containing hierarchy actors, this produce a sub-optimized graph with cascaded *Join* actors. Once again, the optimization condenses these three *Join* actors into a single *Join* actor with four inputs.

5.3.3 Optimization Patterns

In this section, six post-single-rate transformation optimizations are described. All these transformations are designed to reduce the number of actors in the SRDAG and/or to unveil parallelism on the graph.

5.3.3.1 Join/Join Optimization

The first pattern introduced is the *Join*/*Join* pattern and it has already been explored in the previous section. The SRDAG optimization is shown in Figure 5.9.

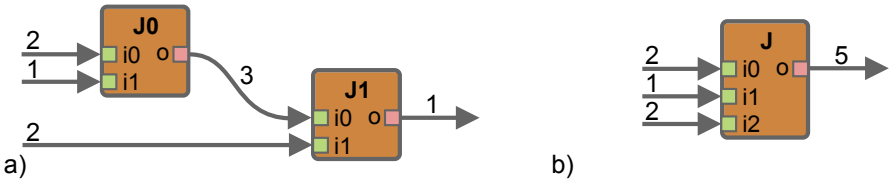


Figure 5.9 – *a) Chained Join/Join actors - b) Join/Join optimization in SRDAG*

This optimization is implemented by checking whether two *Join* actors are chained, as is the case with Figure 5.9-a. If so, the first one is merged into the second, carefully respecting the order of the inputs, as shown in Figure 5.9-b.

This optimization limits the number of actors in the SRDAG leading to faster Mapping/Ordering tasks. However, it also leads to a reduction in the number of synchronization points in the graph during execution of tasks.

5.3.3.2 Fork/Fork Optimization

The second pattern introduced is the *Fork*/*Fork* pattern. This pattern is equivalent to the *Join*/*Join* and is shown in Figure 5.10.

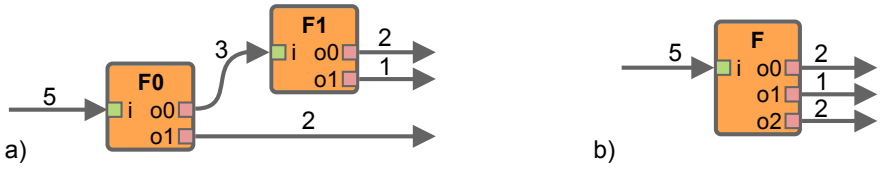


Figure 5.10 – *a) Chained Fork/Fork actors - b) Fork/Fork optimization in SRDAG*

This optimization is implemented by checking whether two *Fork* actors are chained, as per Figure 5.10-a. If so, the second is merged into the first, while carefully respecting the order of the outputs, as is the case of Figure 5.10-b.

As with *Join*/*Join*, this optimization leads to faster Mapping/Ordering tasks and reduces the number of synchronization points during the graph execution.

5.3.3.3 Fork/Join Optimization

The third optimization pattern is the *Fork/Join* pattern. This optimization pattern is shown in Figure 5.11.

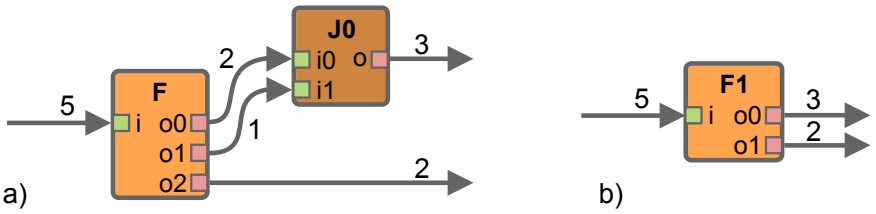


Figure 5.11 – *a) Chained Fork/Join actors - b) Fork/Join optimization in SRDAG*

This optimization is implemented by checking whether a *Fork* actor is chained with a *Join* actor, as is the case in Figure 5.11-a. If so, the *Join* actor can be merged into the *Fork*, while carefully respecting the order of outputs, as is shown in Figure 5.11-b.

As with the previous two optimizations, this method leads to faster Mapping/Ordering of tasks and a reduction of the number of synchronization points during the graph execution.

5.3.3.4 Join/Fork Optimization

The fourth pattern introduced is the *Join/Fork* pattern. The optimization is shown in Figure 5.12.

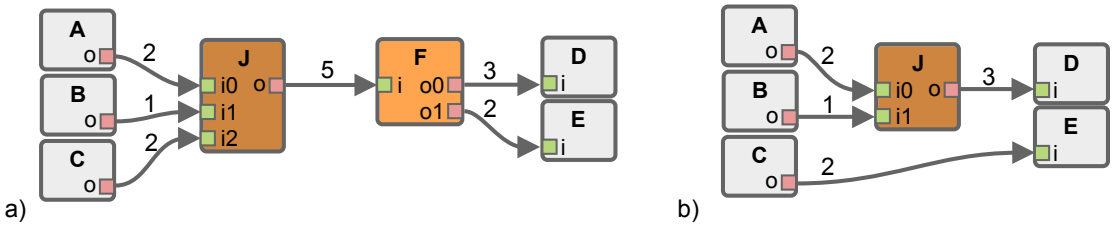


Figure 5.12 – *a) Chained Join/Fork actors - b) Join/Fork optimization in SRDAG*

This optimization is implemented by checking if a *Join* actor is chained with a *Fork* actor, as is the case in Figure 5.12-a. If this is the case, this whole connection can be restructured, as can be seen in Figure 5.12-b. In the graph of Figure 5.12, there are 3 input FIFOs (from actors A, B, anb C) with token rates of (2,1,2) and 2 output FIFOs (from actors D and E) with token rates (3,2). A new instance of the single-rate transformation must then be run on this "connection".

The *Join/Fork* optimization is likely to result in unveiling parallelism on the SRDAG. In Figure 5.12-b, the actor precedence that existed in Figure 5.12-a between actors A and E is broken allowing them to be executed in parallel. This optimization allows the Mapping/Ordering tasks to compute a better scheduling. But it also produces improved scheduling performance as the execution of the application has a higher parallelism.

5.3.3.5 Broadcast/End Optimization

The fifth optimization pattern defined is the *Broadcast/End* pattern. The optimization is shown in Figure 5.13.

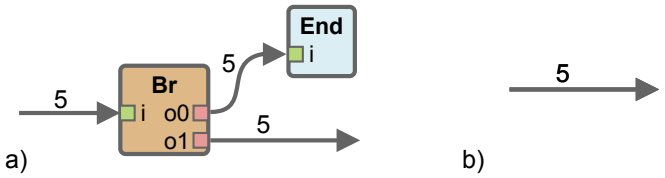


Figure 5.13 – *a) Chained Broadcast/End actors - b) Broadcast/End optimization in SRDAG*

This optimization is implemented by checking if one *Broadcast* actor is chained with one *End* actor, as is the case with Figure 5.13-a. Since *End* actor are used only to discard tokens, they are unneeded after *Broadcast* actors since there are duplicated ones. At this point, the *End* actors and their corresponding output ports on the *Broadcast* actor itself can be removed without consequence. Moreover, if only one output FIFO remains on the *Broadcast* actor, the *Broadcast* actor can be safely removed, as shown in Figure 5.13-b.

As is the case with the previously introduced optimizations, this pattern leads to faster Mapping/Ordering of tasks and a reduced number of synchronization points during the graph execution.

5.3.3.6 Init/End Optimization

The last pattern introduced is the *Init/End* pattern. The optimization is shown in Figure 5.14.

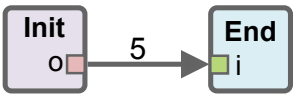


Figure 5.14 – *Chained Init/End actors*

This optimization is implemented by checking if one *Init* actor is chained with one *End* actor. *Init* actors are used to generate dummy tokens and *End* actors are used only to discard tokens. There is then no need of these actors if they are chained together, the pair can be removed without consequence.

This optimization, leads to faster Mapping/Ordering of tasks since there are fewer actors in the SRDAG.

All of these post-single-rate transformation optimizations leads to a reduction in the SRDAG complexity. However, these optimizations have an impact on computation complexity. The impact on computing of these optimizations is studied in Section 7.2.

5.4 Many-core oriented Mapping/Ordering algorithm

As discussed in Chapter 2, the current trend in embedded systems is to integrate increasingly more cores into processors. Many-core DSP processors which include a small number of GPP cores and numerous powerful DSP cores are made possible by higher integration density and may dominate other DSP processors over the next few years.

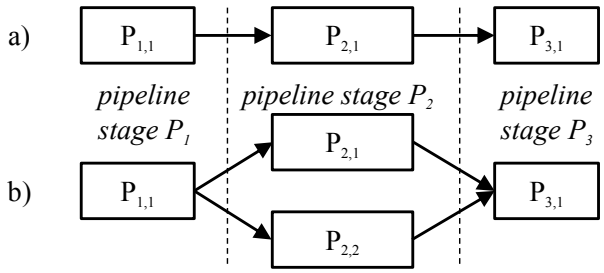


Figure 5.15 – *Three stage pipelines with a) one PE per stage, b) one or more PEs per stage.*

In this section, a new Mapping/Ordering algorithm is presented. This algorithm focuses on many-core devices where the PE count is dramatically more than hundreds.

The two major issues in the Mapping/Ordering algorithm are the nature of the interconnect topology [KAG+09] and the synchronization of computations. Pipelined interconnects and their scheduling are of high relevance in DSP and multimedia applications [SB12]. An example of pipelined interconnects can be seen in Figure 5.15. Figure 5.15 a) depicts a classical pipeline with three *stages*, each consisting of one PE. Figure 5.15 b) depicts a slightly more complex pipeline where the second stage has two PEs (#2 and #3) that can process data concurrently.

The classical flow-shop algorithm is based on the pipelined architecture proposed in Figure 5.15 a). This algorithm has been previously studied in [BBS07, Bou09]. The *Hybrid Flow-Shop (HFS) algorithm* [RVR10], is based on pipelined architectures as in Figure 5.15 b). For multicore architectures that allow bidirectional (isotropic) inter-processor communication, the HFS algorithm restricts communication to fixed, unidirectional links. This dramatically reduces the complexity of the Mapping/Ordering task and keeps its overhead low even for a large number of PEs.

In this section, the HFS Mapping/Ordering algorithm is described in Section 5.4.1. This is followed by, a comparison with a well-known Mapping/Ordering algorithm with experimental results in Section 5.4.2

5.4.1 HFS Mapping/Ordering Algorithm

The origin of the flow-shop scheduling problem formulation is in factory production lines. Multiple production machines work in parallel and products move from one machine to another as they are assembled. The flow-shop scheduling problem involves the processing of N *jobs* on M machines. In the *classical* flow-shop formulation each job consists of a set of *operations*, so that each operation must be processed on exactly one machine. In the context of the PiSDF Mapping/Ordering task, machines are named PEs and operations are named actors.

In the classical flow-shop formulation it is assumed that actors have deterministic execution timings. This constraint is respected in the PiSDF Mapping/Ordering context as each actor is assumed to have a Deterministic Actor Execution Time (DAET). One DAET is assigned to each PiSDF actor and may depend on the parameters of the actor.

The optimization objective of the HFS problem is the *makespan* minimization. Makespan is the latency between the start of the first actor firing and the end of the last actor firing.

The HFS algorithm extends the classical flow-shop methodology by allowing several PEs to process one type of task [RVR10]. Figure 5.16 a) shows an application that consists of three tasks, t_1, t_2 and t_3. This tiny task graph represents a flow-shop *job*. The job exists in three instances: J_a, J_b and J_c. Figure 5.16 b) shows the Gantt chart (schedule)

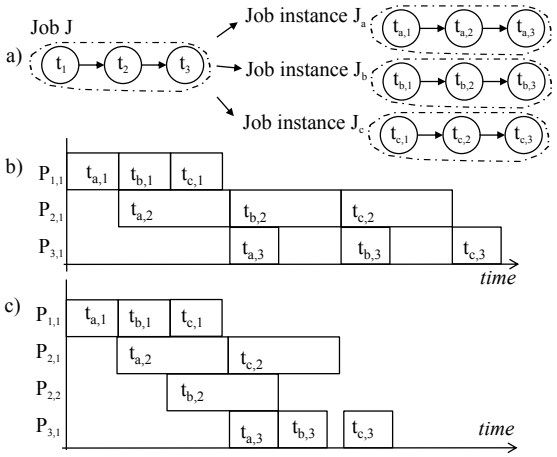


Figure 5.16 – *a) 3 instances of one job J consisting of three tasks. b) Instances of J scheduled on architecture shown in Figure 5.15 a). c) Instances of J scheduled on architecture shown in Figure 5.15 b).*

when the job instances J_a, J_b and J_c have been scheduled onto the pipelined multiprocessor architecture shown in Figure 5.15 a). Figure 5.16 c) shows the same jobs scheduled on the pipelined architecture of Figure 5.15 b), where PE2 and PE3 are both capable of executing task t_2.

Figure 5.16 highlights the common situation where one kind of task (t_2, in this case) is computationally more demanding than other tasks in the application and thus forms a bottleneck in the processing pipeline. The classical flow-shop formulation can not represent cases where a long latency task is distributed to more than one PE, whereas in HFS algorithm this poses no problem.

In this algorithm, some common flow-shop assumptions are used:

- jobs are processed in every stage in the same order;
- each actor is processed by at most one PE at each pipeline stage;
- all jobs are available simultaneously before the methodology starts;
- actor preemption is not allowed.

The Mapping/Ordering algorithm is introduced with an example displayed in Figure 5.17. The Figure 5.17 a) shows the inputted SRDAG which is composed of 22 actors. This example corresponds to a signal processing algorithm which forms a part of the 3rd Generation Partnership Project (3GPP) Long Term Evolution (LTE) uplink processing [HBP+13].

The HFS Mapping/Ordering algorithm is composed of three steps: actor and PE assignment, job extraction and actor Mapping/Ordering.

5.4.1.1 Actor and PE Assignment

The HFS algorithm requires that each job contains only one kind of actor for each pipeline stage. Each actor of the SRDAG is assigned to a pipeline stage, as a function of its connection with other actors.

In the Figure 5.17, actors of the same pipeline stage are marked with the same letter. To assign actors to the same stage necessities firstly classifying actors with their level. Actor level is the maximal number of preceding actors that lead to a source actor.

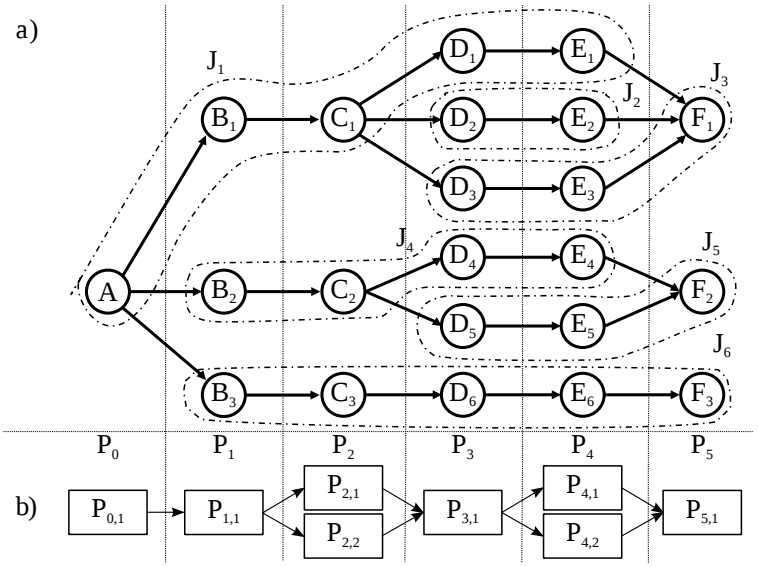


Figure 5.17 – *a) SRDAG with flow-shop jobs b) Mapping to 8 PEs.*

Next, PEs are assigned to each pipeline stage depending on the sum of the DAET of actors assigned to this stage. The goal of this step is to balance the load between stages. In the example, Figure 5.17 b) displays an example pipeline configuration with eight PEs. Actors C and E are the most computing intensive actors of the graph. Two PEs have then been assign to their stages.

5.4.1.2 Job Extraction

After the assignment of actors to pipeline stages, jobs can be extracted from the SRDAG. Figure 5.17-a shows the jobs resulting from considering the 6-stage pipeline with eight PEs in shown in Figure 5.17-b.

In Figure 5.17 a), individual job instances are delimited by dashed borders. The algorithm employed to create jobs from the SRDAG is shown in Algorithm 5.1.

It is important to note that the HFS jobs generated from the SRDAG are not independent; that is, data dependency may exist between them. When considering Figure 5.17, it must be noted that the causality relationships between the SRDAG actors still hold for HFS jobs. Consequently, the scheduling order of jobs is somewhat restricted.

Job ordering is a task that has been extensively studied [RVR10, Fre82]. In general, ordering strategies are very time-consuming and are not appropriate for runtime computation. The choice must then be made to keep jobs in their creation order that ensures no broken actor dependencies.

5.4.1.3 Actor Mapping/Ordering

After the set of jobs, $J_i, i \in [1, N]$, has been defined and has been mapped to pipeline stages $P_j, j \in [1, M]$, the actual HFS Mapping/Ordering can be conducted.

After job ordering has been completed, the *assignment* phase selects the processing element $P_{j,k}$ from the set of PEs in stage $P_{j,k}, k \in [1, M_j]$ to which task t_m has been assigned. In this work, the *earliest possible start assignment* was used: at each step, the PE chosen grants the earliest start of the current task.

Algorithm 5.1: Constructing Jobs from Actors

```
Input: A SRDAG G
Output: A queue of jobs L
1  begin
2      static currentJob = new empty job;
3      static jobList = new empty job list;
4      call jobMaker(first actor of G);
5      return jobList;
6  end
7  function jobMaker Input: currentActor
8  begin
9      if currentActor was visited then
10         return;
11     end
12     if one of currentActor predecessors was not visited then
13         push currentJob to L;
14         currentJob = new empty job;
15         return;
16     end
17     add currentActor to currentJob;
18     currentActor.isVisited = true;
19     if currentActor has no successor then
20         push currentJob to L;
21         currentJob = new empty job;
22     end
23     else
24         for each vertex in curVertex successors do
25             jobMaker(vertex);
26         end
27     end
28     return;
29 end
```

Figure 5.18 shows the multicore HFS schedule of the example of Figure 5.17. The pipeline shape of the schedule is already visible for this small number of actors.

By restricting the actor Mapping/Ordering to a only a portion of the architecture, the processing time of this Mapping/Ordering algorithm is reduced. Experiments have been performed on this Mapping/Ordering algorithm and the results are compared with those of a well-know list algorithm. The experimental results are detailed in the following section.

5.4.2 Experimental Results

In the future, the progression of computationally efficient DSPs is likely to consist of combinations of GPP cores for control and scheduling and many DSP cores for signal processing. Results have been generated by Mapping/Ordering a dataflow description of LTE Physical Uplink Shared Channel (PUSCH) decoding for a many-core DSP on an ARM Cortex-A9 GPP processor. The experiments study the evolution of the simulated makespan and the Mapping/Ordering time for increased PE count.

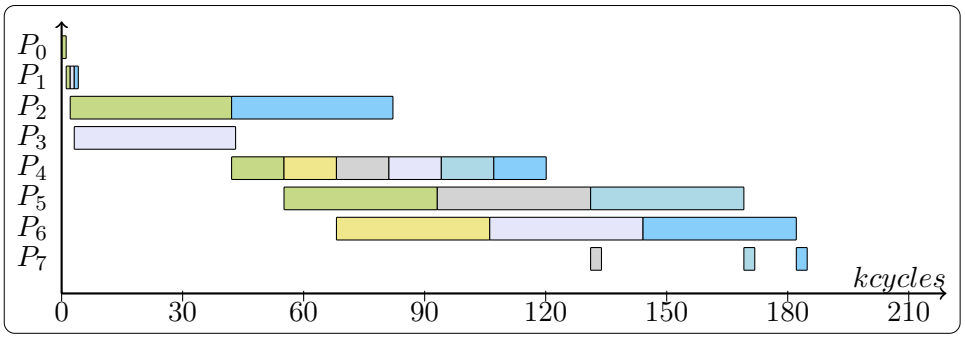


Figure 5.18 – *Multicore HFS schedule of the example in Figure 5.17.*

In this context, the experimental results highlight the differences between the list Mapping/Ordering algorithm and the HFS algorithm.

The Mapping/Ordering algorithm used for the comparison is a list scheduling algorithm called Modified Critical Path (MCP) [WG87]. This Mapping/Ordering algorithm is based path length computation where the critical path is the longest path of the whole graph. Path lengths are determined by the sum of the execution time for actors of the given path. MCP then computes the last possible start time of actors using the longest path of the current task to any exit task and the critical path of the graph. An ordered list of actors is derived from these computations, which gives the Mapping/Ordering order. Actors are then mapped and serially ordered to the most efficient PE according to this list.

The hardware platform contains a dual-core ARM Cortex-A9 processor with a clock frequency of up to 1 GHz. The experimental adaptive scheduler is written in C++ and runs as a Linux mono-threaded task on the ARM processor. The embedded Cycle Counter (CCNT) provides accurate cycle measurements.

In order to compare schedules, tasks need to have known durations. Timing measurements used for these experiments are based on a Texas Instrument c64x+ DSP. Communication times between PEs are assumed to be part of the task timings. The simulated platform is thus a homogeneous many-core platform with cores equivalent to c64x+ DSPs.

Two metrics are used to compare Mapping/Ordering algorithms:

- **Makespan:** The time difference between the start and the end of a given application graph execution.
- **Scheduling Overhead:** The time necessary to compute the Mapping/Ordering algorithm on the target platform. As this time is not dedicated to signal processing, it is an important parameter to minimize.

The results on makespan (Figure 5.19) are shown for both list and HFS Mapping/Ordering algorithms to Greedy Scheduling Theorem (GST) algorithm [Lei05]. The GST curve provides a benchmark for the speedup that can be obtained with greedy scheduling, no actor timing knowledge and a fully-connected isotropic architecture with infinite communication rates [PAPN12].

In the test case, the critical path of the algorithm limits the potential speedup improvement to about 50 times. It may be noted from Figure 5.19 that the list algorithm produces better speedup than the HFS algorithm. The list algorithm reaches its critical path length limitation at the point when fifty PEs are available. On average, the makespan generated by the HFS algorithm is 31% longer than that produced by the list algorithm. The limitation of HFS algorithm in terms of makespan is due to its reduced mapping choices, as all PEs must communicate to a restricted list of PEs in a single direction.

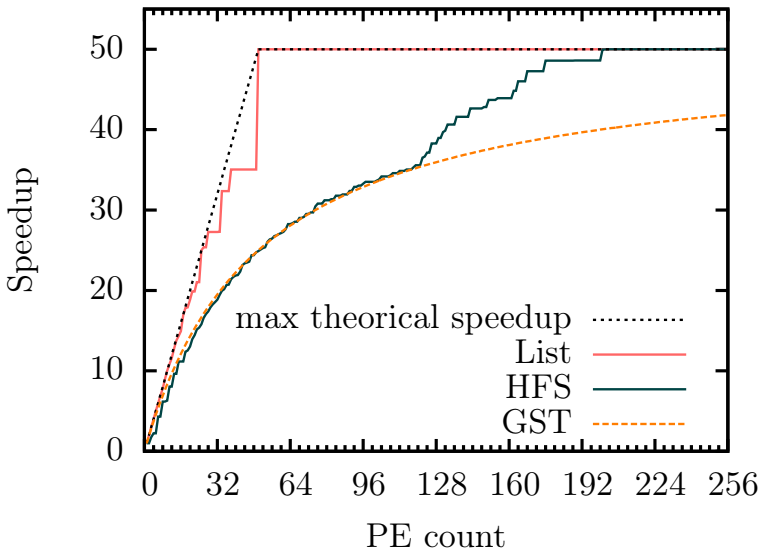


Figure 5.19 – *Makespan vs. # of PEs*

The major advantage of the HFS algorithm is that its computing overhead (Figure 5.20) increases very slowly with the number of PEs. It is for this reason that the HFS algorithm is a good solution as many-core Mapping/Ordering algorithm. This is not the case for a list scheduler algorithm whose computing overhead increases significantly with the number of PEs. In Figure 5.20, it may be seen that the HFS scheduler has a lower computing time once 17 PEs (or more) are available on the architecture. For fewer than 17 PEs, the time needed to construct HFS jobs makes the HFS algorithm requires computing time than list algorithm.

The conclusion of the experiments is that the HFS algorithm has a very low computing time compared to optimized list algorithm for higher PE count. The HFS algorithm presupposes pipelined PEs and orientates inter-PE communications. This simplifies PE assignment when compared to the general isotropic and fully-connected architecture case. It also groups computing tasks into linear jobs, simplifying actor assignment while maintaining task precedence.

For the case of 256 PEs and for all LTE PUSCH configurations, the HFS algorithm has a computing overhead under 1 Mcycles. This barrier corresponds to the period between two reconfigurations of this algorithm. This makes HFS algorithm suitable as a real-time Mapping/Ordering algorithm for the worst case (comprising 501 actor instances) for the LTE application onto a 256-core architecture.

5.5 Conclusion

This chapter presents three different improvements that can be applied to the JIT-MS scheduling algorithm.

The first is an extension of the PiSDF MoC which has been proposed to enhance scheduling performance. This extension focuses on delay initialization. In general, for the unenhanced scheduling method, these token are uninitialized or set to zeros. In the proposed extension, these tokens may be initialized by an interface or a configuration actor.

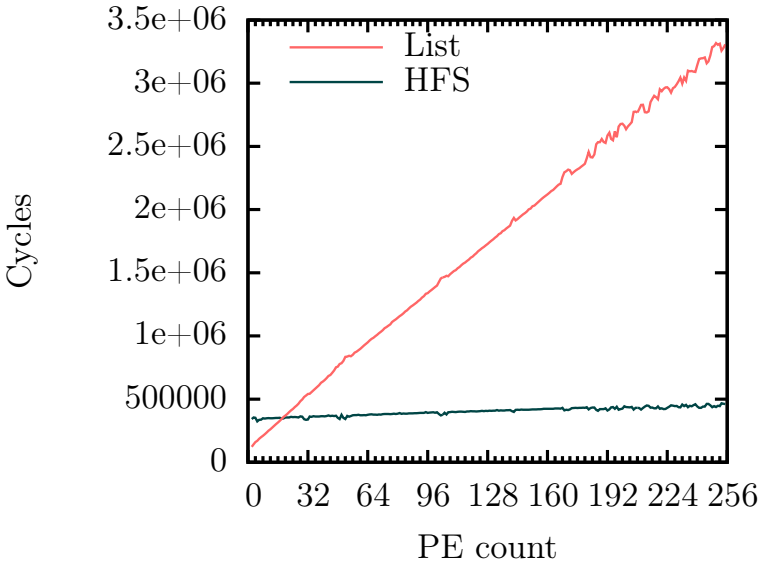


Figure 5.20 – *Computing overhead vs. # of PEs*

This extension simplifies the bulk of PiSDF graph and allows improved performance of the scheduling method.

Optimizations on post-single-rate SRDAG transformations have also been explored. These optimizations allow the reduction of the SRDAG complexity by merging or deleting actors, by employing known special actor behavior to simplify the SRDAG. Thus, synchronization within the whole graph is reduced and the Mapping/Ordering task has fewer actors to treat.

Finally, a new Mapping/Ordering algorithm has been proposed for many-core devices. This devices possess hundreds of PEs and the common Mapping/Ordering heuristics require a very long computation time. This Mapping/Ordering algorithm is demonstrated to be faster than the list scheduling algorithm with 17 or more PEs. Since none of the experimental platforms used in this thesis have a PE count greater than 17, the HFS Mapping/Ordering algorithm has not been employed for experiments. However, it can be used in the JIT-MS scheduling method if the targeted platform integrates a large number of PEs.

The purpose of this chapter was to introduce the optimizations of the JIT-MS scheduling method. The following chapter will describe the runtime called Spider that embeds this method.

CHAPTER 6

Spider: a dataflow multicore runtime

6.1 Introduction

In this chapter, the structure of the Synchronous Parameterized Interfaced Dataflow Embedded Runtime (Spider), designed during this thesis, is detailed. Spider is a dataflow based multicore runtime that targets heterogeneous embedded platforms. Spider is designed to be a low-level runtime that allows efficient and dynamic reconfiguration of applications on multicore platforms taking advantage of the dataflow model properties. This runtime takes a PiSDF graph as input and embeds the JIT-MS method described in Chapter 4.

The overall PiSDF framework is composed of two steps. First, the application is defined using the Eclipse-based Preesm tool which provides a graphical interface to describe and develop the application. Preesmcan also be used as a fast prototyping tool providing scheduling simulations with fixed parameters as described in Section 3.3.4. The Spider runtime is then employed as a low-level multicore operating system that drives the adaptive execution of the designed PiSDF graph.

In Spider, the PiSDF graph is stored in the local memory of a centralized manager called Global RunTime (GRT). The PiSDF graph is statically set at runtime initialization. The PiSDF graph is described using C/C++ code. Since Preesm already provides a graphical interface to design PiSDF graphs, only a Spider compatible code generator needed to be added to the Preesm tool. The code generation flow is described in Figure 6.1.

As with the case of the static scheduling flow of Preesm, the code generation destined for Spider requires three inputs. The first input is the PiSDF graph: one or multiple pi files, designed using the graphical interface within Preesm. Each actor of the PiSDF graph is associated with a C function or C++ method instantiated in the actor code. The second input is the architecture model described using the System-Level Architecture Model (S-LAM). This architecture model was introduced in [PNP+09]. The architecture model describes the targeted platform and is referenced by the third input, the scenario. The scenario provides information on actor firing for each PE of the platform such as constraints and timings. The scenario information is integrated into the PiSDF graph in Spider; it is needed by the Preesm integrated code generator for Spider.

Finally, the generated C++ code which represents the PiSDF graph, the architecture model and the scenario is compiled referencing the Spider library that embeds the runtime

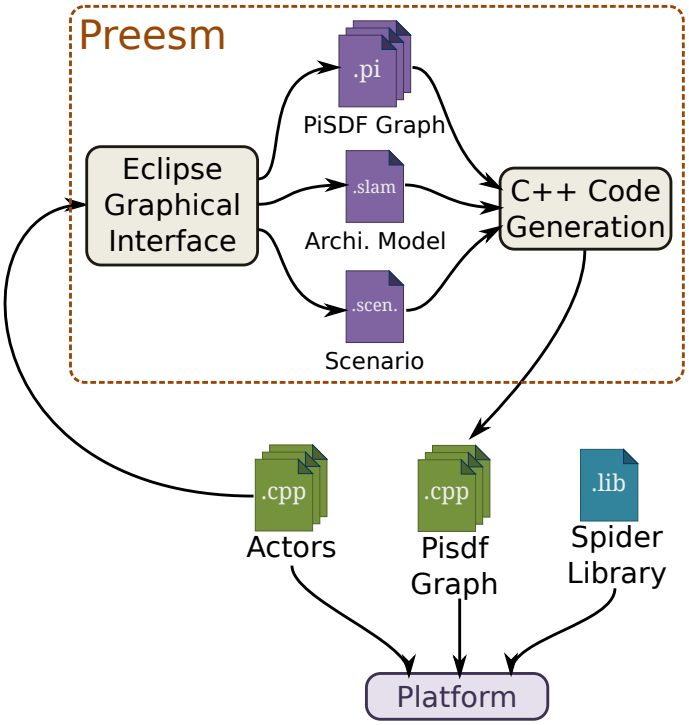


Figure 6.1 – *PiSDF based Programming Framework overview*

system. This allows the Spider runtime to load the PiSDF graph in local memory at initialization. The executable file generated is then able to execute the given application and to use the features of the Spider runtime.

In this chapter, implementation details of the Spider runtime are given. First, the design choices of the Spider runtime are explored. In Section 6.2, an abstract description of the runtime that ensures its portability on several platforms is then derived.

The implementation on different hardware platforms is discussed in Section 6.3. The Spider runtime is implemented on a desktop Linux, a Zynq (ARM+FPGA) platform and a Keystone platform (ARM+DSPs). The use of different hardware components to implement the Spider runtime description is detailed in this section.

Finally, Section 6.4 details certain optimizations of Spider which enhance runtime efficiency. In particular, the reduction of the runtime overhead due to synchronization mechanisms and the resulting improvement of data token management leads to the adoption of these solutions to increase the performance of the Spider runtime.

6.2 General runtime structure

In this section, the general structure of Spider is described. Features embedded in the Spider runtime are designed to be both platform and application independent. Depending on the target platform, each component may be software or hardware implemented.

Figure 6.2 shows an example of the Spider runtime with three elements grouped into a global structure. The platform is composed of two Local RunTime (LRT) and one Global RunTime (GRT). Each LRT must run concurrently on each PE driven by Spider. There is only one GRT; it drives the LRTs.

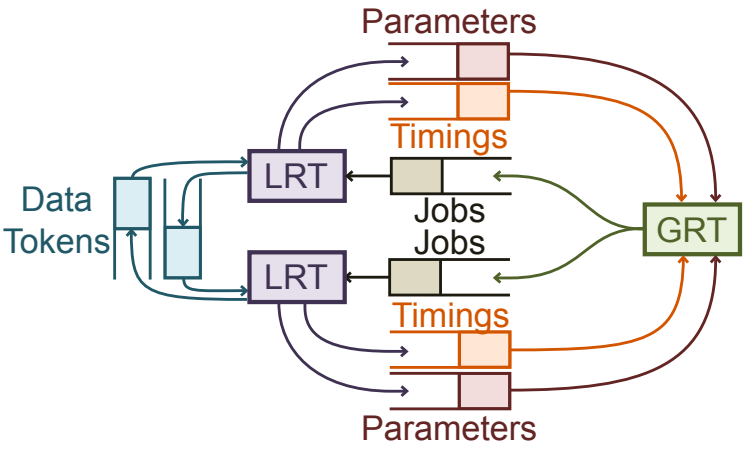


Figure 6.2 – *Spider runtime structure*

The Spider GRT is the master of the system. It is responsible for the multicore scheduling of the application. The LRTs are lightweight execution runtimes used for executing actor code.

Spider embeds multiple queues, which may be separated into two categories:

- Data queues are used to exchange the data tokens manipulated by LRTs.
- Control queues, consisting of parameter, timing and job queues, are used for the internal communication between the LRTs and the GRT all of which comprise the runtime.

In the following sections, the master/slave scheme for the runtime is explored. The role of data queues and control queues are then discussed. Finally the LRT and GRT elements and their function within the different layers of the runtime are presented.

6.2.1 Master/Slave structure

A very important feature of the Spider runtime is its compatibility with heterogeneous platforms. Performing Mapping/Ordering tasks on a heterogeneous platform is more complex than on a homogeneous platform. Indeed, a decision to start an actor firing based on local optimization criteria in a heterogeneous system may be globally inefficient. An example is given in Section 7.2.3: OpenMP which makes scheduling decisions locally is compared with the Spider runtime.

In [SSKH13], Singh et al. classify mapping methodologies by diverse criteria. They define a platform manager that has the responsibility of mapping tasks and handling both resources and configuration. One of their criteria to classify mapping methodologies is the control management behavior which can be either centralized (*global*), or distributed (*local*).

In global management, one core of the platform handles the entire task mapping process. This concept corresponds to a Master/Slave scheme. Alternatively, local management divides the platform into clusters which have their own local management. Each cluster handles the task mapping on its own PEs and communicates with other clusters. Typically local management is composed of a shared job queue which receives the output jobs from each cluster concurrently for immediate execution.

The Spider runtime implements a global approach. The primary advantage of the global approach is illustrated in Figure 6.3.

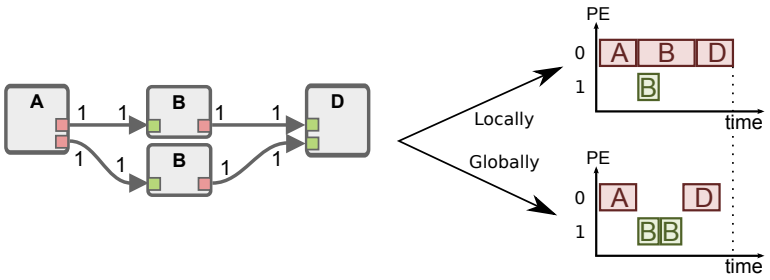


Figure 6.3 – *Local vs Global decision for scheduling*

In this example, a single-rate graph with four actors (A, $B1$, $B2$, D) is mapped onto a two-PE heterogeneous architecture. The first PE (ID 0) is a general purpose PE that can execute all actors. The second PE is a specialized co-processor that efficiently executes a single actor type (actor B for this example).

For the case where the runtime of each core takes local decisions using a shared pool of actors, the upper execution Gantt chart of Figure 6.3 is the probable outcome, as PE 0 is ready to start the B actor before PE 1. Immediately after A is fired, both PEs launch one instance of the actor B.

However, global mapping decisions translate into a knowledge of the graph topology and the execution time of actors on each PE. This results in a better decision, as illustrated in the lower execution Gantt chart of Figure 6.3. The global scheduling technique means that the two firings of the actor B are both mapped on the specialized PE 1, thus achieving an earlier completion time.

This example demonstrates that better multicore scheduling decisions can be taken from a global perspective. For this end, a Master/Slave scheme is adopted in Spider for runtime mapping and scheduling decisions. In this scheme, a single PE, the master, takes all mapping decisions and sends execution actors to the other PEs which function as slaves.

6.2.2 Data Queues

In the PiSDF MoC, actors exchange data through Fifo channels. When the graph is converted into a single-rate graph, Fifos are present in the transformed graph. However, a single-rate Fifo represents a one-time communication between two specific actor firings. Consequently, there is no need of a full Fifo implementation with simultaneous multi-read/multi-write capabilities within the single-rate graph. Nevertheless, the implementation of the single-rate Fifos must be adapted to the memory organization of the targeted hardware platform. The following cases must be considered:

6.2.2.1 Shared Memory Without Cache

In a shared memory system, a single-rate Fifo can be implemented using the two elements, that comprise a Spider data queue:

- A shared memory segment, in which to store data tokens.
- A semaphore which permits the synchronization of the token writer and the token reader. This semaphore guarantees the precedence between actor firings. In Spider,

this semaphore is the only Inter-Process Communication (IPC) mechanism used for data transmission between actors.

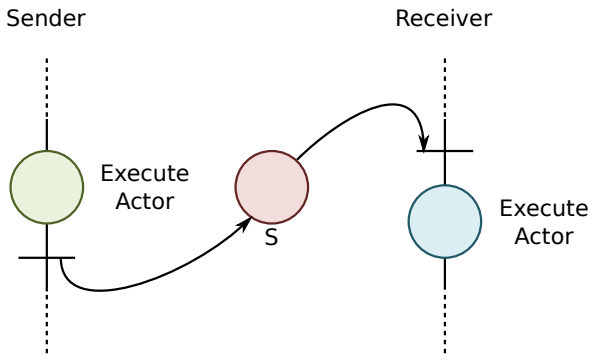


Figure 6.4 – *Data queue synchronization mechanism on shared memory without cache*

The simple synchronization mechanism is detailed in Figure 6.4. The synchronization mechanism is defined for the sender and the receiver. Firstly, the sender executes the source actor of the given single-rate graph Fifo. As the sender directly manipulates data in the shared memory, there is no need for data transmission when the execution is complete. When the data tokens are ready in memory, the sender posts a token into a shared semaphore to signal to the receiver that data tokens are ready. Once the receiver has successfully popped the semaphore, its actor code is executed with valid input data.

6.2.2.2 Shared Memory With Cache

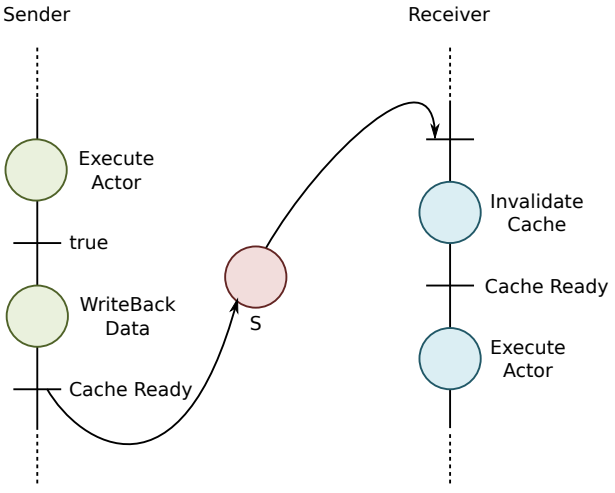


Figure 6.5 – *Data queue synchronization mechanism with shared memory with cache*

The synchronization mechanism for a platform with cache memory is detailed in Figure 6.5. In such a platform, data written in a shared memory by an actor executed on a PE may be temporarily stored in the private cache associated with this PE. To ensure that this data is transferred to the shared memory accessible to all PEs, the sender of a Fifo must execute a write-back operation thus forcing the transfer of the newly written data from

its local cache to the shared memory. After the cache subsystem finishes the write-back operation, the sender can post the token into the shared semaphore. Then, the receiver can pop the semaphore, and invalidate its cache to gather valid data. This invalidate operation guarantees that all data corresponding to the accessed segment of shared memory is flushed out from the local cache of the PE. This means that the next read operation on this shared segment will read the data from the shared memory, and not from the local cache. Once this has been completed, the receiver can launch the execution of the actor.

6.2.3 Parameters

In the PiSDF MoC, all parameters may influence the token rates and the execution timing of PiSDF actors. The values of the parameters are computed by specific actors called configuration actors.

When a configuration actor is fired, the newly computed parameter value is communicated to the runtime master. If a configuration actor is executed by a slave PE, a *parameter queue* is needed to send the parameter values back to the master, as shown in Figure 6.2. The master then resolves all token rates and timings that are influenced by this parameter.

Once all parameters are resolved for a given subgraph, the graph may be scheduled using the JIT-MS scheduling method as presented in Chapter 4.

6.2.4 Timings

To obtain feedback on an executed graph and to provide RTOS capabilities in Spider, timing information from previous executions is sent back to the master.

A dataflow system contains useful timing information, which is called trace, and includes the start and end instants of each actor firing. This information is gathered with a single reference and a single scale, and can be used to provide a Gantt chart of the real execution on the targeted platform.

Traces are useful on two levels:

- The master PE can monitor a set of real-time deadlines and identify those that have been missed. This information can be used to take better scheduling decisions during the subsequent graph iterations. Using traces, the master can also refine its time predictions for actor executions. Moreover, it can set PEs into idle or powered off modes in a power-aware system.

- In a rapid prototyping context, traces can be used to provide feedback to application programmers on the current platform usage. Traces are also helpful to profile applications, assisting programmers in updating their application description for more efficient platform use. For example, a computation-hungry monolithic actor can be decomposed (or refined) into a subgraph to unveil more parallelism resulting in a faster application computation.

6.2.5 Jobs

A job is an IPC data structure that embeds all information required to fire a single instance of an actor. A job is sent from the master PE to a slave PE each time that the master PE maps an actor firing onto the slave.

A job is composed of the following:

- Information about the actor which will fire. This information can be transmitted, for example, as a function id, or as a location of source code in shared memory.

- A reference to input and output data queues. In shared memory subsystems, this reference is comprised of both a semaphore id and a pointer to the shared memory segment where the single-rate FIFO is stored.

- Parameter values for this execution.

6.2.6 Local RunTime (LRT)

Slave runtime elements in SPIDER are called LRTs. The goal of a LRT is to execute all actors corresponding to the jobs assigned by the master to its slave. Each LRT contains a unique job queue, from which the jobs are sent. The jobs within the queue are ordered and specifically mapped to the PE which runs the given LRT. Similarly, each LRT has its own parameter and timing queues, which return configuration parameters and execution traces to the master. The LRT indefinitely executes the following steps:

1. Output one job from the Job queue.

2. Wait until the input data queue tokens are available.

3. Execute the actor code.

4. Send output tokens to the output data queues.

5. Eventually, send parameter values in the parameter queue if the fired actor is a configuration actor.

6. Push trace data in the timing queue.

6.2.7 Global RunTime (GRT)

The master of the SPIDER runtime is called GRT. The GRT is the core of SPIDER. Its primary objective is to compute and apply the JIT-MS scheduling algorithm to the input PiSDF graph. For this end, the GRT sends jobs to the job queues of the LRTs and outputs parameters values from the parameter queues. Once all pending scheduling activities are fulfilled, one LRT is integrated within the GRT in order to execute actors. As a consequence, the PE executing the GRT also processes actors.

The GRT and all the LRTs are each assigned to an execution thread. The GRT and the LRTs are software components and can be deployed on platforms over existing single or multicore operating systems, or as a bare-metal program on a PE. It is important to note that SPIDER considers that each LRT owns its specific computational resource at all times. When a SPIDER implementation does not respect this assumption, the result is altered performance but unchanged data integrity.

6.2.8 A multi-layered Runtime

The SPIDER runtime structure is designed to execute different applications on multiple platforms. The SPIDER runtime is a C/C++ multi-layered runtime composed of an application layer, a runtime layer and a hardware specific layer. The overall SPIDER implementation is described in Figure 6.6.

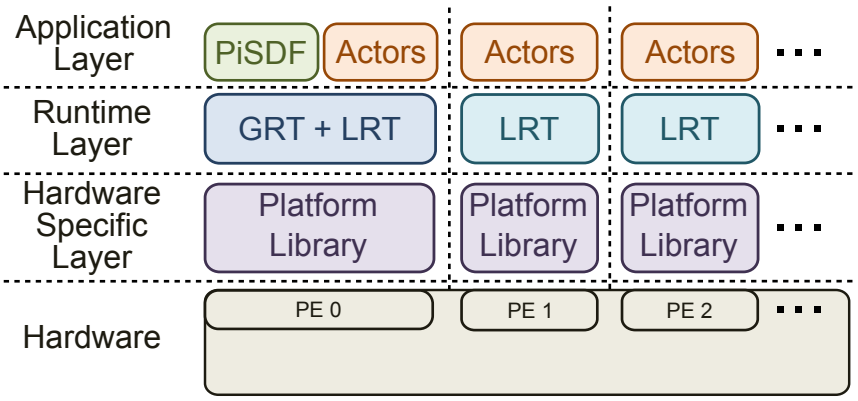


Figure 6.6 – *Spider Runtime layers*

6.2.8.1 Hardware specific layer

The hardware specific layer allows Spider to operate on different platforms. This layer is presented as a platform library which implements a communication API. A specific platform library is needed for each platform targeted by the runtime. This library is used to implement the following features in platform hardware:

- Data queue management
- Control queue management: each control queue is comprised of timing, parameter and job queues
- Global Time management

Three commonly used platform libraries are outlined in Section 6.3.

6.2.8.2 Runtime layer

The runtime layer embeds platform independent features of Spider. In the master PE, both a GRT and a LRT are integrated to enable the execution of actors in addition to scheduling activities. For each slave PE, a single instance of LRT is embedded.

6.2.8.3 Application layer

On the master PE, the application layer consists of the PiSDF graph and the code of the actors. On each slave PE, the application layer only consists of the code of all actors that can be fired in this PE. Specific, optimized actor code is often provided for each type of PE in a targeted heterogeneous platform. The PiSDF graph is loaded in the local memory of the GRT during the initialization phase of Spider. The PiSDF graph is described using C/C++ code.

6.3 Runtime implementation on Shared memory Platforms

In this section, specific implementations of Spider platform library are discussed. Three distinct target platforms are explored in this section: x86 with Linux, ARM+Field-Programmable Gate Array (FPGA) and ARM+DSPs. For all platforms, the design choices of data queues, control queues and time management are detailed.

6.3.1 x86 implementation

The x86 implementation of Spider was developed and maintained during the entire development process of the runtime. This implementation constitutes a prototyping and debugging platform for the embedded systems. It may be noted that desktop systems based on x86 processors usually provides better and faster debugging capabilities than their embedded counterparts.

This is due to the fact that desktop implementation is primarily used for prototyping and debugging features of the runtime manager. When compared to embedded implementations, its performance has not been greatly optimized. A more performance oriented version using more efficient Linux mechanism synchronization and communication services is planned as future work. This platform can also be used by application developers to validate a PiSDF description. Furthermore, functional testing is possible with x86 platforms as the runtime implementation employs all the Spider features specified in Section 6.2.

The x86 implementation was developed over an SMP Linux Desktop operating system. Spider is embedded into a shared library and relies on the Linux system IPC. Embedding Spider in a library permits the separation of application-specific code from the Spider implementation code.

When Spider is launched, its initialization process consists of forking itself to create as many LRTs as needed. Each LRT is associated with a unique LRT index. On completion of this initialization phase, the parent process becomes the GRT and launches the PiSDF application.

In the following subsections, the Spider components implemented are described. These components include data queues, control queues and global time management.

6.3.1.1 Data Queues

Spider handles each data Fifo of the single-rate graph by creating a corresponding data queue. These data queues are implemented using a Linux shared memory region, which is used by all the processes. This shared memory is separated into two sections.

The first section stores a dedicated flag for each data Fifo instantiated by the runtime. These flags serve as a synchronization mechanism, and indicate the runtime of data availability in queues. The index for each data queue is chosen by the GRT at runtime and is then embedded into the job description.

The second memory section stores data token values. Single-rate Fifos are allocated into memory segments by Spider. The resulting memory allocation of each single-rate Fifo is also incorporated into the job description.

Finally, a unique semaphore is used to protect the entire flag section. This semaphore acts to protect simultaneous writes into a single flag. This semaphore creates a bottleneck in the Spider implementation. The effect is particularly significant with a large number of LRTs running simultaneously. However, since the x86 platform does not solely prioritize computational performance, the issue of this bottleneck is not overly important.

6.3.1.2 Control Queues

To implement control queues employed for timings, jobs, and parameters, Linux pipes have been used. A Linux pipe is a unidirectional data channel that can be used for interprocess communication.

Since all control queues are unidirectional in the SPIDER implementation, implementing Linux pipes are a natural choice. Three Linux pipes are created in SPIDER for each instantiated LRT.

The first Linux pipe is created to transmit jobs. Jobs are identified by the single-rate actor index which acts like a job index.

The second Linux pipe is created to transmit timing data. The timing data and the single-rate actor index to which they refer are both transmitted as a trace packet.

The last Linux pipe is created to transmit parameter data. As with the case of timing data, the index of the configuration actor which generated this parameter value is also transmitted through this pipe with the parameter data.

Consequently, there are three slave pipes for each slave LRT in the system. This is a reasonable number for the current multicore desktop systems.

6.3.1.3 Time management

In order to capture coherent timing measurements and to build a valid execution Gantt chart, a global time reference is needed.

For this end, Linux operating systems provide access to a monotonic clock. As described in the Linux manual, the monotonic clock is a one that can never be set and represents monotonic time with an unspecified starting point. This clock is used as a time reference in the SPIDER x86 implementation.

At the beginning of the graph execution, the GRT retrieves the reference time which is subsequently communicated to all LRTs. Thus, when a LRT needs a time value, it simply reads the monotonic clock and subtracts the reference.

6.3.2 An ARM+FPGA Platform: Xilinx Zynq Zedboard

The previous system is now contrasted with an embedded platform: an ARM+FPGA platform, the Xilinx Zynq Zedboard [Zed15]. This platform embeds a dual core Cortex-A9 ARM processor in addition to a FPGA.

This combination offers advanced rapid prototyping due to its fully customizable, programmable logic region, which is in located in close proximity to its efficient hardwired processors. It is straightforward to implement the SPIDER data and control queue subsystems and a shared timer in the FPGA portion of the platform. The heterogeneous platform is composed of 2 ARM processors and multiple softcores called microblazes implemented into the FPGA. Actors may then be run by an ARM processor or a microblaze.

The ARM+FPGA implementation of SPIDER was tested in [HOP+13] on a simple application which allows easy parallelization: a Sobel filter. The memory required to run this image processing application was larger than the entire memory capacity of the platform. Consequently, communications between the ARM processors and the FPGA were supported through an external memory, which considerably decreased the performance. Another limitation of this implementation was the unbalanced computational behavior between 866MHz ARM processors and 100MHz softcores. As a result of the communication overhead and the poor performance of the softcore, offloading actor firings on the FPGA produced only a very small increase in speedup (only a few percent).

It is for these reasons that this platform is not selected for the experiments of Chapter 7. It is the hardwired heterogeneous platform of the next section which was chosen for all experiments, as the speedups produced are greater and so the results are more exploitable.

6.3.3 An embedded platform: the Keystone architecture

The third platform to implement the SPIDER runtime is the Texas Instruments Keystone architecture. This platform is an embedded heterogeneous MPSoC that contains many processing elements. The powerful DSP cores are designed to handle intensive signal and video processing of State-of-the-Art applications. Another advantage of this MPSoCs is the enhanced IPC capabilities.

6.3.3.1 Description of the two Keystone platforms

The first platform explored was the Keystone I architecture, part number: C6678 [Texc]. This MPSoC is composed of eight DSP cores running at speeds up to 1.25 GHz. This MPSoC is described in Figure 6.7.

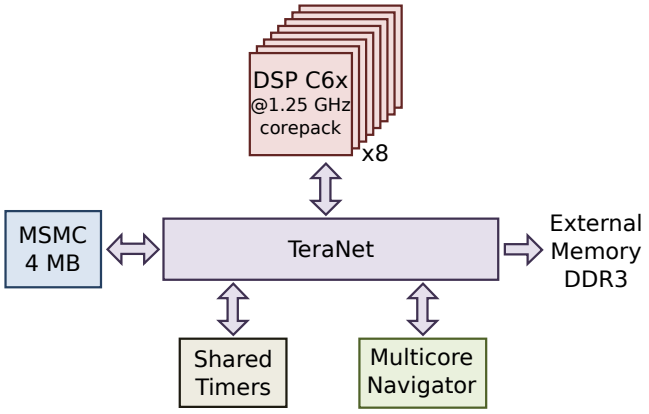


Figure 6.7 – *Keystone I Architecture*

Each of the eight processing cores is embedded in a corepack containing 512 KB of L2 cache. DSP cores are linked by an interconnect called TeraNet which enables efficient access to several shared components:

- A Shared internal memory called Multicore Shared Memory Controller (MSMC). In the Keystone I architecture, this memory is 4MB wide.
- Multiple Shared Timers.
- An External DDR3 memory. This memory is 512MB wide in the platform used.
- An IPC-related device called *Multicore Navigator* [Texa].

The Multicore Navigator is a hardware property which makes the Keystone platform particularly suitable for the SPIDER runtime. Its primary purpose is to provide many multicore-oriented features to the system thus enabling efficient multicore programming. The Multicore Navigator contributes a set of hardware queues embedded in the Queue Manager Sub-System (QMSS) and many Direct Memory Accesses (DMAs) called "Packet DMAs". These queues ensure atomic push and pop that enable efficient synchronization between cores. The Packet DMA subsystem is coupled with these queues and provides efficient memory transfers operating in background mode. All data going through the queues is incorporated into a descriptor, allowing the DMAs to conduct operations. The QMSS manages 8192 hardware queues and several Packet DMAs to facilitate background memory transfers.

This Keystone I architecture was notably used for generating early results of SPIDER in [HPD+14].

The evolution of this platform containing additional ARM cores is the Keystone II architecture, developed by Texas Instruments. The improved heterogeneity of the Keystone II make it a better platform for a SPIDER implementation. Subsequent developments were made on the 66AK2H14 [Texb] MPSoC that employs the Keystone II architecture.

The Keystone II architecture is described in Figure 6.8.

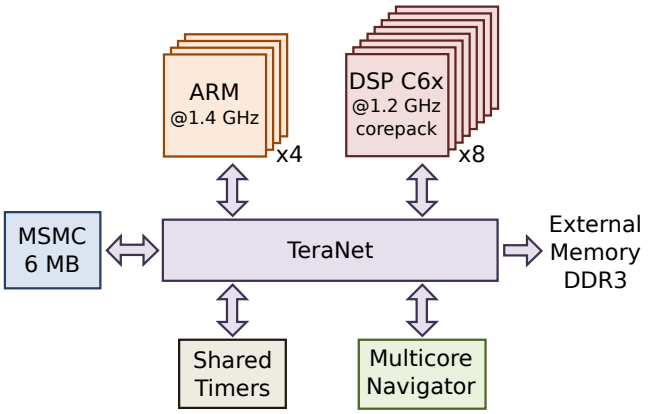


Figure 6.8 – *Keystone II Architecture*

Keystone II embeds 4 Cortex-A15 ARM processors running at speeds up to 1.4GHz. These cores are connected to the TeraNet interconnect, and access the same L2 shared memory as the DSP cores: the MSMC. The 66AK2H14 MPSoC also contains the following enhancements, as compared to the C6678:

- Bigger shared memory, upgraded from 4MB to 6MB.
- Doubled Multicore Navigator capabilities with 2 queue managers for a total of 16384 queues.
- Doubled capacity of the accessible DDR3 memory (4GB).

The Keystone II architecture was used for all experiments described in this thesis. In the following sections, the SPIDER implementation on this platform is detailed.

6.3.3.2 Data Queues

To implement data queues, the hardware components employed are: the Multicore Navigator for synchronization, the MSMC shared memory, and optionally the DDR shared memory.

The Multicore Navigator allows DSP and ARM cores to exchange messages through specific queues using atomic push and pop operations. There are three types of messages, called descriptors, that can be sent with these queues: monolithic, host packet and host buffer.

Monolithic descriptors refer to a non fragmented data message. Host packets and host buffers are used to link a portion of data to a descriptor even if it is in a memory location that is not accessible by the descriptor creator. This enables packet DMAs to make data transfers of this data portion in the background.

For the purpose of simplicity, data tokens in SPIDER are stored in a memory shared between all the cores (internal MSMC or external DDR). This was also the case with the x86 implementation. From the data embedded in the job structure, the sender and the receiver cores know where data tokens are stored within the shared memory. Furthermore, since data tokens are stored in shared memory, there is no need for the packet DMA to transfer data: all tokens are accessible by every cores. Consequently, only monolithic descriptors are used to handle the synchronization between cores in the SPIDER implementation.

For core synchronization, a hardware queue is arbitrary chosen by the GRT at runtime and embedded into the job description. The LRT inputs a descriptor from an input queue before running the actor and outputting the descriptor to its output queue when the execution finished. The memory allocation of data tokens is chosen by the SPIDER GRT and is also incorporated into the job description.

Synchronization with the Multicore Navigator makes access to shared memory predictable and suitable for enabling caches. Each LRT can write-back and invalidate the memory cache in order to retrieve uncorrupted data.

6.3.3.3 Control Queues

Control queues are responsible for transmitting parameters, jobs, and timings; the Multicore Navigator is used for their implementation. As the amount of data exchanged is limited, data is stored directly into the descriptor thus requiring no additional memory space.

As with the x86 implementation, three queues are used for each LRT instantiated at runtime. Hence, on the Keystone II platform, 36 of the 16384 available queues are used as control queues.

6.3.3.4 Time Management

The Keystone II architecture provides 16 shared timers. One of these shared timers dedicated to be the global time reference. This ensures relevant (global) timing information on current and previous executions.

To ensure that accesses to the global timer do not become a bottleneck of the SPIDER implementation, each DSP core uses a private local timer to get timestamped information. To create the timing information, a core accesses the shared timed once and then uses the captured value as a reference to correct timings provided by its local timer. This optimization reduces the impact of the timing retrieval, so lowering the SPIDER runtime overhead.

At the beginning of each graph execution, the GRT resets the shared timer and sends a signal to all LRTs to force each to refresh their global reference value from the shared timer.

6.3.3.5 Co-processor management

Keystone MPSoCs embed a co-processor, which is employed for a significant use case. The use case will be described in Chapter 7. This co-processor is the FFT co-processor called Fast Fourier Transform Co-processor (FFTC).

This co-processor is used to perform the FFT signal transform on a 1D array of complex fixed point values. The FFTC provides many useful features such as dynamic scaling, pipelining, and so on.

FFTC computation is triggered by posting a descriptor in a dedicated queue of the Multicore Navigator. This descriptor is a host packet and the data transfer is handled directly by packet DMAs. After the execution, the descriptor is pushed into a dedicated output queue.

Even if this behavior seems appropriate for a Spider implementation, one FFTC cannot be directly supported as a regular LRT. This is due to multiple reasons:

- Timing information will not be generated. This is because the FFTC is not a programmable core, and so this behavior cannot be specified
- The FFTC configuration cannot be popped from another descriptor as the case for regular LRT.
- The FFTC output location in memory is not configurable. The FFTC may only pop a descriptor from input queue and write its results directly into the data buffer attached to it.

Hence, in order to configure the FFTC, an additional core of the MPSoC must be used to feed the FFTC with the desired computation. In the Keystone II architecture, the use of the ARM processors to compute FFT actors is not less efficient than when employing to DSP cores. Since DSPs can also be suitable PEs for FFT actors, two FFTCs can be driven by two ARM processors. In this way, ARM processors are responsible for configuring, launching and retrieving results from FFTCs, and behave as interfaces between Spider and the co-processors.

It is this embedded platform that is used for all the experiments of Chapter 7. In the following section, three optimizations to enhance the performance of Spider are presented.

6.4 Runtime Optimizations

The three optimizations discussed in this section improve the IPC and shared memory usage. They target actor precedence, special actor memory allocation and special actor precedence.

6.4.1 Actor Precedence

In the Spider runtime, synchronization between cores is a key concern. Reducing the frequency of synchronization points between cores leads to a better performance. The following optimization is based on an analysis of actor precedence information to improve IPC. An example is used to illustrate this optimization.

Figure 6.9-a displays an example of a single-rate graph. After applying Mapping/Ordering tasks, the simulated Gantt chart is displayed in Figure 6.9-b.

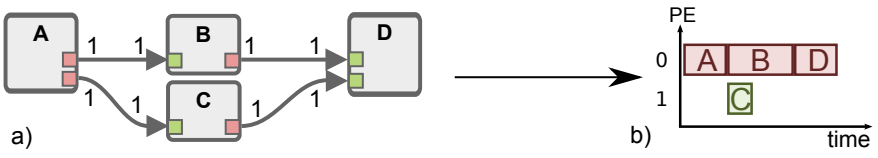


Figure 6.9 – *IPC optimization example*

In the single-rate graph, it can be that at the completion of actor A, a data token is sent to actor B and another to actor C. In the keystone implementation in Spider, where

each core embeds a private cache memory, this requires two descriptors to be sent and then corresponding memory regions must be written-back (see Section 6.2.2).

Then, actors B and C each pop a descriptor from their input FIFO and invalidate the corresponding data memory. These operations are required for actor C as this actor is mapped to a different PE.

However, for actor B, the precedence between firings of actors A and B is already established since they are mapped and ordered on the same core. Consequently, the write-back and invalidate operations are not necessary. Removing write-back and invalidate operations when not needed can have a major improvement on performance. This is especially significant when many data tokens are exchanged between actors.

To implement this optimization, each queue is associated with a flag. This flag indicates to the LRT whether either a synchronization or a cache operation is required for the single-rate FIFO. The flag is set to true only if source and end actors of a single-rate FIFO are mapped to different cores.

This method is a simple way to enhance the performance of the IPC process, with improvements of 51%, as will be Section 7.2.2.2. Moreover, a more detailed examination of graph precedence can further minimize the number of inter-core synchronization points. An equivalent approach can be found in the literature and is appropriate for application to this specific platform. In [BSL+95], the creation of a synchronization graph permits a better understanding. This additional step constitutes a potential improvement of SPIDER, which has not been explored in this thesis However, this method is limited by the scheduling overhead produced from these optimizations which may dominate performance gains.

6.4.2 Memory allocation of special actors

As explained in Section 4.4.1, special actors are actors that have specific behavior. This allows queue memory allocation to be automatically optimized. Since the behavior of special actors can be reduced to token transfer operations, the majority can be executed without requiring a data copy. For example, all outputs of a *Broadcast* actor can point to the same memory space.

This approach can make certain special actors transparent in terms of real data movement. In particular, this applies to *Fork*, *Join* and *Broadcast* actors. As a result, it is only the synchronizations which are then needed.

This optimization is implemented in the memory allocation phase of SPIDER. When allocating single-rate graph FIFOs in memory, the FIFOs connected to special actors are the first to be allocated. When possible, the same memory is used to allocate both input and output data queues of these special actors. This optimization has two effects: the execution time of special actors becomes zero, and the memory requirement of the overall graph is lowered.

As shown in Figure 6.10, the queue memory allocation of the *Fork* actor is reduced to only one memory section. The dummy allocation used in this example allocates queues in contiguous memory spaces. When memory allocation is based on special actor behavior, queue memory spaces can be merged to reduce the overall allocation space as detailed in [DPNA15a].

Similarly, all queues connected to an *End* actor of the single-rate graph are merged in the same memory space, since data tokens sent to the *End* actors should be discarded.

However, this proposed optimization has some limitations and cannot be applied to every special actor in the single-rate graph. An example of these limitations is presented in Figure 6.11.

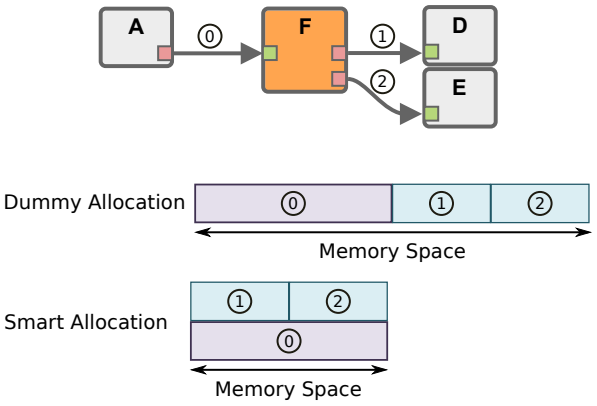


Figure 6.10 – *Special Actor Memory Allocation Optimization*

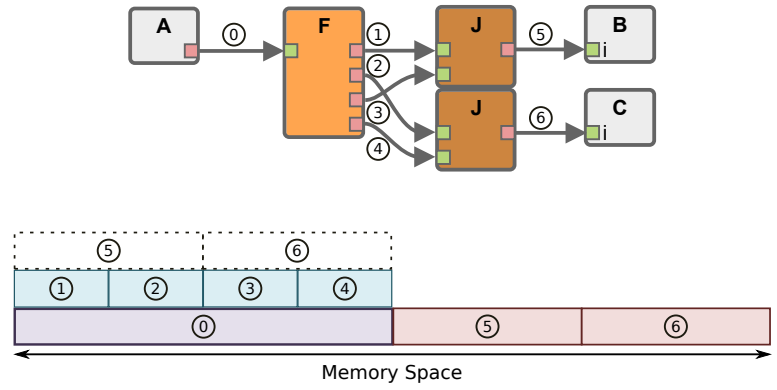


Figure 6.11 – *Special Actor Memory Allocation Limitations*

In this case, the allocation of the *Join* actor output queues cannot be merged with their input since they are not contiguous in memory. These *Join* actors then have to copy data memory to create macro-tokens.

This optimization handles the case when optimizations are not possible. Memory optimization of *Join* actors must be checked to verify whether input FIFOs are already allocated.

Better memory allocation methods can also be applied, as presented in [DPNA15a], through the use of memory exclusion graphs and actor behavior with scripts. However, since the memory allocation phase is executed at runtime, the computational resources required for memory allocation must be carefully analyzed as the resulting choice constraints may result in a deterioration of the overall performance.

The optimization of the special actor memory allocation is most efficient when combined with the optimization presented in the next section.

6.4.3 Special Actors Precedence

In the previous section, certain special actors are optimized so that they become simple synchronization points. Further optimization can be applied so that these special actors are handled directly by the SPIDER runtime. This concept is introduced with an example of a single-rate graph, which is presented in Figure 6.12.

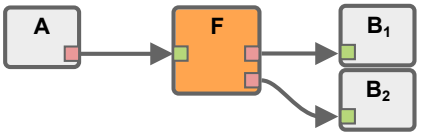


Figure 6.12 – *Special Actor Precedence Single-rate Graph Example*

Using the original precedence synchronization mechanism presented in Section 6.2.2, the resulting petri net is shown in Figure 6.13. Since mapping is disregarded, the petri net is composed of 4 execution threads (one for each single-rate actor) which are mapped onto a unique core. Actor A triggers the execution of actor F using semaphore S_1, then actor F triggers the execution of actors B_1 and B_2 using semaphores S_2 and S_3 respectively.

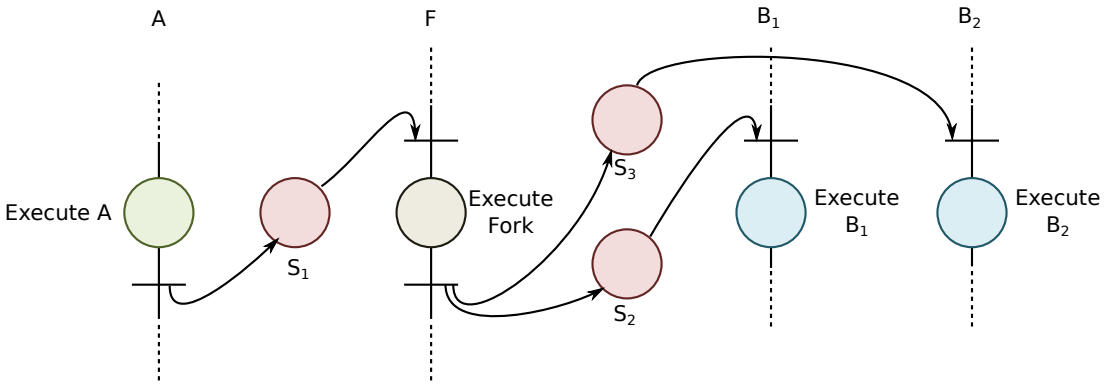


Figure 6.13 – *Example synchronization mechanism without optimizations*

If the *Fork* actor is allocated statically in memory, the fork execution does not modify data tokens since their location is already correct. Hence, firing the *Fork* actor is not necessary, but the precedence between actor A, B_1 and B_2 must still be enforced.

To allow this multiple end synchronization pattern, the semaphore which synchronizes a queue is modified to be capable of storing multiple tokens. In this way, it is possible for the input queue of this *Fork* actor to add n tokens into the semaphore. Then each of the n output queues of this *Fork* simply pops one token from this semaphore.

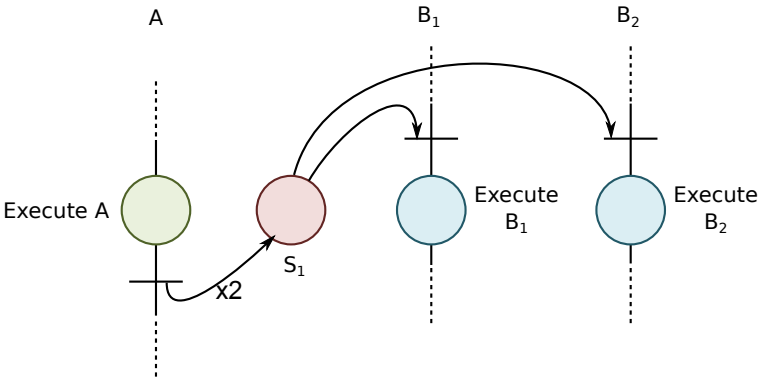


Figure 6.14 – *Example of synchronization mechanism with optimizations*

In our case, $n = 2$ and the resulting petri net of this synchronization is given in Figure 6.14. This results in a sole semaphore S_1 fed by 2 tokens from the thread A after the execution of actor A. Then, actors B_1 and B_2 will pop one token each, assuming that their input data is available.

This optimization can be extended to *Broadcast* and *Join* actors. This allows the *Fork*, *Join* and *Broadcast* actors to be handled directly by the runtime and are thus removed from the Mapping/Ordering task. So the overall Mapping/Ordering multicore scheduling task has fewer actors to handle which reduces its computation time.

Thus, it can be seen that this optimization reduces the need for synchronization resources but also lowers IPC cost of the overall graph. Resulting performance gain is discussed in Section 7.2.2.3.

6.5 Conclusion

In this chapter, an embedded multicore runtime called Spider is introduced. This runtime embeds the JIT-MS multicore scheduling method allowing the efficient dispatch of a given PiSDF application at runtime.

This runtime has a master/slave configuration where a master called GRT performs the multicore scheduling. It then sends jobs to all slaves called LRTs which contain the required information for firing the corresponding actor. Once the actor is fired, each LRT sends back execution traces to the GRT. The GRT then has a global view on the recently completed graph execution, allowing it to produce an execution Gantt chart.

The runtime is designed to be portable to multiple platforms. Spider is a layered runtime that may be efficiently ported to another platform, with only a small amount of additional code. Currently, Linux desktop and TI's Keystone II platform are supported by Spider but future work may involve more platforms.

At the runtime level, multiples optimizations are possible. These optimizations can either reduce synchronization between actor firings or reduce the shared memory footprint. With all optimizations activated, some special actor firings can be unneeded, allowing a reduction in the Mapping/Ordering complexity and execution time.

CHAPTER 7

Experimental Results and Use Cases

7.1 Introduction

In the previous chapters, the JIT-MS method was introduced. This is a multicore scheduling procedure based on the PiSDF dataflow MoC. Certain optimizations on different subparts of the JIT-MS method were then proposed in Chapter 5. Finally, an embedded runtime called Spider was detailed in Chapter 6. Spider is an implementation of the JIT-MS method.

In this chapter, the performance of Spider is evaluated on three applications. Experiments were conducted on the Texas Instruments Keystone II platform detailed in Section 6.3.3.

The first application, called HCLM-Sched, is a benchmark composed of multiple chains of Finite Impulse Response (FIR) filters of different lengths. FIR filters are used in many signal processing embedded systems in both audio and telecommunication applications to pass a predefined range of frequencies. FIR filters can be efficiently described using PiSDF and the descriptions can be managed by Spider at runtime. The HCLM-Sched benchmark evaluates the performance of the optimizations proposed in Chapter 5 in terms of the PiSDF descriptions. OpenMP is the reference framework for parallel programming on multicore embedded systems in industry. The primary advantage of OpenMP is its easy parallelization process offered to the developer starting from sequential code. OpenMP is known to be efficient for signal processing applications like FIR filters. A comparison between Spider and the OpenMP runtime in terms of performance on the Keystone II platform is thus very challenging. PiSDF descriptions are platform independent unlike the OpenMP approach.

Next, a canonical signal processing application FFT is benchmarked with Spider. The algorithm representation with the PiSDF MoC is discussed in Section 7.3. Moreover, this algorithm is also used to demonstrate the capacity of Spider to drive an heterogeneous platform. The FFT is a more complex algorithm than the FIR and showcases the performance of Spider.

The final case is an embedded vision algorithm, namely a stereo matching application, which computes a disparity map from two stereoscopic images. This algorithm is described in PiSDF and is implemented with Spider.

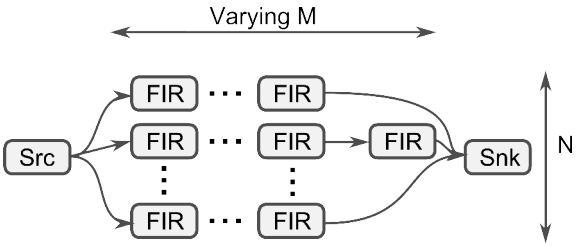


Figure 7.1 – *HCLMP-sched description.*

These three applications are used to benchmark the scheduling method on applications of different shapes and are composed of actors with varied needs for computational intensity (termed different granularities). The stereo matching application is composed of computational intensive actors while the FFT application is composed of smaller and more highly optimized actors.

All three applications yield information on PiSDF, JIT-MS and SPIDER because good execution performance results only from efficient modeling, scheduling method and runtime. These applications use general algorithm structures such as the nested for loop of HCLM-Sched and the butterfly structure of the FFT. The methodology of representing these structures in the PiSDF MoC is also a contribution of this thesis.

The HCLM-Sched is presented in Section 7.2, and the parallel FFT follows in Section 7.3. Finally, the stereo matching algorithm is presented in Section 7.4.

7.2 HCLM-Sched Benchmark Algorithm

This benchmark is an extension of the *MP-sched* benchmark used in [ZPB+13]. The MP-sched benchmark can be viewed as a two-dimensional grid involving N branches ("channels"), where each branch consists of M cascaded FIR filters. Here, the MP-sched benchmark is extended by allowing the parameter M to vary across different branches, as illustrated in Figure 7.1. The parameter M is used to change the order of the FIR filters. A better filter sharpness is obtained by increasing the order of the filter at the price of more computations. The best tradeoff between quality and complexity is found for an application by optimizing the value of the parameter M. This extended "or generalized" version of the MP-sched benchmark is referred to as the *heterogeneous-chain-length MP-sched* (*HCLM-sched*).

7.2.1 PiSDF representation

A PiSDF representation of the HCLM-sched benchmark is a graph with two hierarchy levels.

The top graph is shown in Figure 7.2. To represent the channels in the HCLM-sched benchmark, a hierarchical actor called *FIR_Chan* is introduced. The input data is inserted in the graph by the *src* actor and the sample results are processed and outputted by the *snk* actor. A configuration actor named *cfg* is instantiated in this graph. Its purpose is to configure the graph depending on external parameters, typically extracted from a file or from a previously received data packet header. This configuration actor sets the parameter N and outputs the value of the parameter M which will be different for each channel.

The values of the parameter M are stored in an array of size *Nmax* since configuration actors cannot have dynamic data output rates. So, only M values are kept at the output

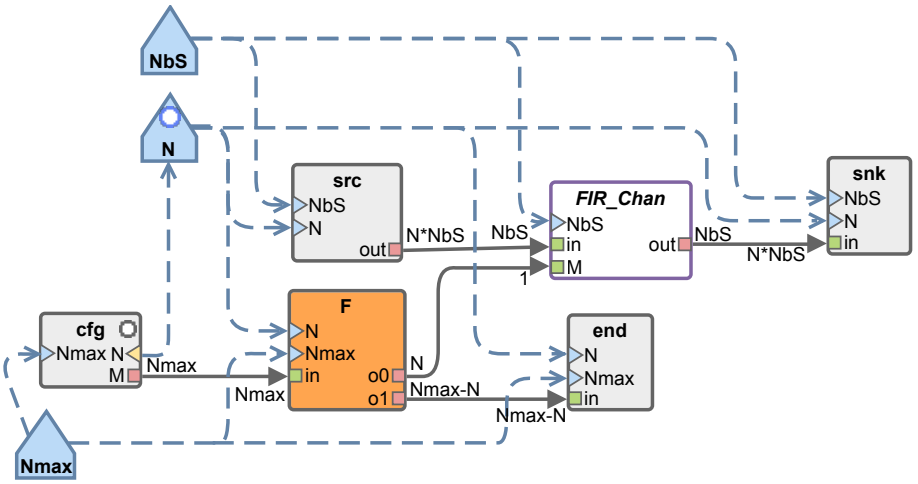


Figure 7.2 – *Top graph of the HCLM-sched benchmark*

of the actor F, which is an instance of the special *Fork* actor. The first N tokens are sent to the *FIR_Chan* hierarchical actor and the remainder of the tokens are sent to a sink actor called *end* whose sole purpose is to discard this unneeded data. The top level graph is designed to repeat this *FIR_Chan* actor N times; each instance receives both one token which contains the parameter M value and NbS tokens as input data samples.

The subgraph describing the behavior of the *FIR_Chan* actor is shown in Figure 7.3. The FIR filter is repeated M times due to a feedback loop and specific control actors (*initSw*, *Switch* and *Br*).

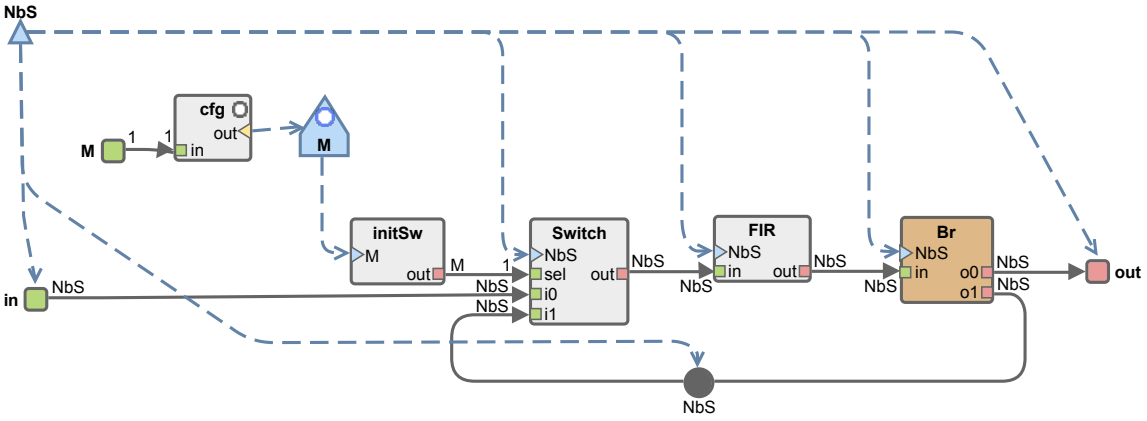


Figure 7.3 – *Subgraph of the HCLM-sched benchmark representing the FIR_Chan hierarchical actor*

The *Switch* actor is vital, as its purpose is to select the input data of the *FIR* actor from either the input interface or the feedback edge. It thus behaves as a multiplexer. The choice of the *Switch* actor is controlled by the *sel* input port value which is connected to the *initSw* actor.

For this use case, the *initSw* actor is implemented as follows. The *initSw* actor token value is first set to 0 which results in the *Switch* actor selecting the input interface as input data for the first firing of the *FIR* actor. The remaining value of the $M-1$ tokens outputted by the *sel* actor are set to 1, which results in the selection of data from the feedback loop for *FIR* actor input. The *initSw* also has another purpose. By generating M values, the BRV computation will be forced to fire the *FIR* actor M times.

In the final stage, the sole function of actor *Br* is to generate two identical data streams: one for the feedback loop and the other for the output port. This operation can be performed without any data movement; the actors are configured so that the subsequent actor points to the same data memory space. Since the output interface of the subgraph behaves as a RB, only the final output data computed by the *FIR* actor is passed to the upper graph as expected.

When the PiSDF delay extension described in Section 5.2 is employed, the subgraph of the *FIR_Chan* actor can alternatively be described as shown in Figure 7.4.

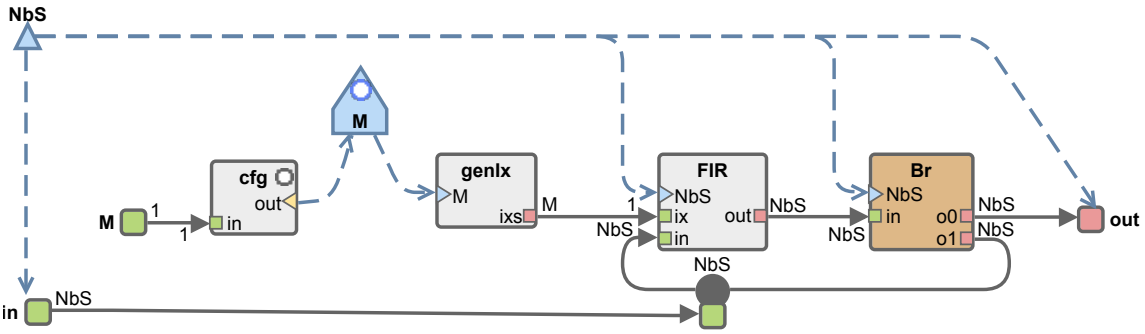


Figure 7.4 – *Subgraph of the HCLM-sched benchmark representing the FIR_Chan hierarchical actor using the delay extension*

In this graph, the initial delay tokens are set by the input interface. This removes the need for the *Switch* actor. Eliminating the *Switch* actor simplifies the graph and as a consequence its scheduling time.

However, the *initSw* actor (transformed here in *genIx*) is still required, it forces the BRV to fire the *FIR* actor M times. It is for this reason that a data input port called *ix* has been added to the *FIR* actor which will ensure the 1:M ratio but does not carry information data. The *genIx* is a basic actor that produces M tokens with the following values: $[0 \dots M-1]$. This then provides each *FIR* actor with the knowledge of the current iteration of the FIR filter. All *FIR* filters are equivalent, in the case of the HCLM-sched benchmark, so this information is unused. In a more general case, or where the actor is modified for each instance, this iteration information may be used.

This PiSDF description enables an algorithm description without a control path. The algorithm control is provided by the hierarchy description combined with the parameter tree.

7.2.2 Benchmarks of the Optimizations

In this section, the experimental result of the optimizations explored in previous chapters is presented. All experiments have been conducted on the Texas Instruments Keystone II platform that embeds 8 c66x DSP cores and 4 general-purpose ARM Cortex A-15 cores. The PiSDF description and Spider runtime are platform independent so that the same application can be run efficiently despite the number of cores. This example illustrates the efficiency of the approach on this last generation multicore embedded platform.

The GRT of Spider runs over a Linux environment on one of the Cortex A-15 cores. The FIR filters are fired only in the DSPs and are implemented using the corresponding function from the Texas Instruments DSPLib signal processing library.

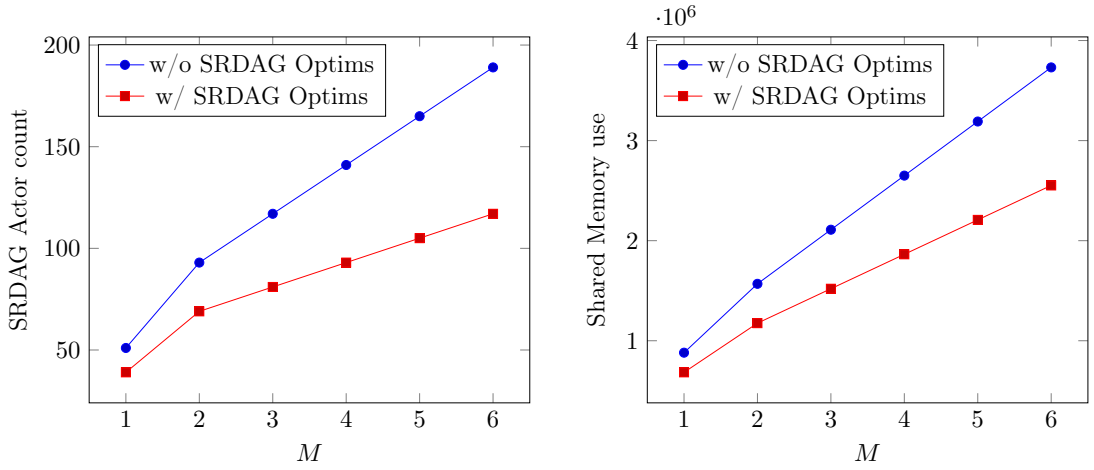
In all experiment in this section, 512-taps FIR filters are executed on 4000 integer data. For all experiments, the parameter N is fixed to 6. The parameter M is identical for all FIR channels, and is swept from 1 to 6.

First, the post single-rate transformations are benchmarked in Section 7.2.2.1, then runtime optimizations are benchmarked in Section 7.2.2.2 and 7.2.2.3. Finally, the impact of the PiSDF MoC extension is discussed in Section 7.2.2.4.

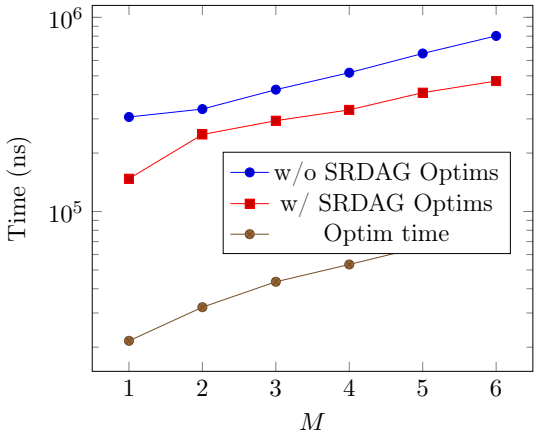
7.2.2.1 Post Single-Rate Tranformation Optimizations

As described in Section 5.3, certain optimizations can be applied to the SRDAG to reduce its complexity. In this section, the consequences of these operations on the following Mapping/Ordering task are evaluated.

For this experiment, no other optimization is employed. Special actors are treated in the same way as the other actors and the classical PiSDF MoC is used (without the proposed extension).



(a) *SRDAG Actor count vs M value* **(b)** *Shared Memory use vs M value*



(c) *Mapping/Ordering Time vs M value*

Figure 7.5 – *Post Single-Rate Transformation Optimizations Benchmark*

The SRDAG actor count is plotted in Figure 7.5a. It can be seen that the SRDAG actor count is dramatically reduced, by up to 53% in this benchmark.

Since there are fewer actors in the SRDAG, there are also fewer FIFOs. As no optimization is performed on special actors, the resulting large number of FIFOs has an impact on the overall memory allocated in shared memory. This is evident in Figure 7.5b where the shared memory use is displayed. The optimized version saves up to 48% of shared memory in this benchmark.

Figure 7.5c shows the difference between the sub-optimized SRDAG and the optimized, in terms of Mapping/Ordering times. The displayed numbers include the time necessary to send jobs to the different LRTs. It can be seen for the case of the optimization that the global Mapping/Ordering time is then reduced by up to 56% in this benchmark. This benchmark illustrates that the optimization of the post single-rate transformation is an effective path for global scheduling performance improvement.

7.2.2.2 Actor Precedence

Actor precedence optimization is described in Section 6.4.1.

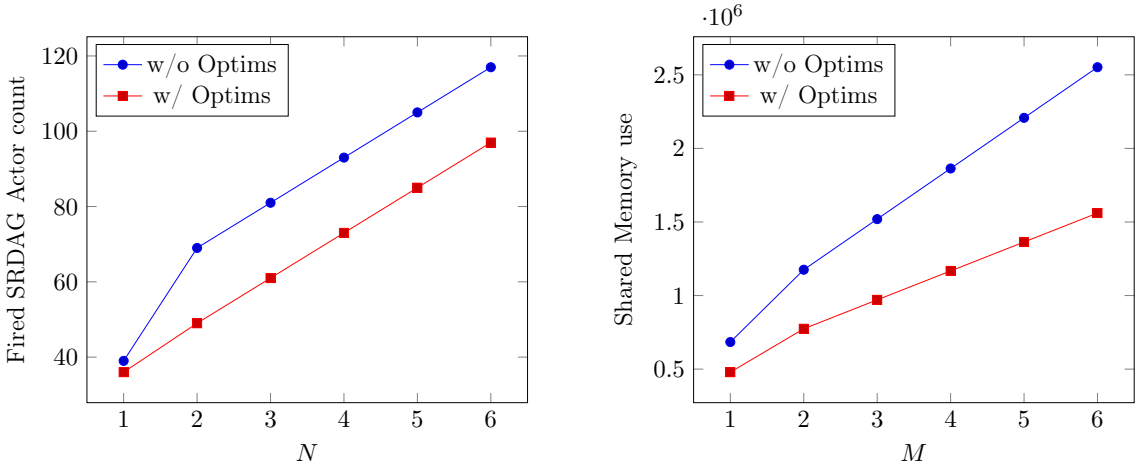
To benchmark the performance gain with this method, a timing measurement is made between two FIR actors exchanging data in the HCLM-Sched benchmark. Without the actor precedence operation, the time between two FIR actors exchanging 4000 samples is about 8.5 μs. This time includes output cache write-back, job fetching and input cache invalidate. With the actor precedence optimization, this time is reduced to 4.2 μs which is an improvement of 51%.

7.2.2.3 Special Actor Optimizations

In this section, the two special actor optimizations described in Section 6.4.2 and Section 6.4.3 are benchmarked together. Two metrics are used to benchmark the impact of these optimizations : the number of mapped/ordrerd actors and the shared memory footprint.

Special actors are handled directly by the runtime, and since the optimization removes the need to process these special actors, a smaller number of mapped/ordered actors are expected after the optimization. In Figure 7.6a, the fired SRDAG actor count is displayed with and without optimization. As expected, the fired actors count decreases by up to 29% after optimization. Consequently, it may be concluded that the Mapping/Ordering task will be faster, as is demonstrated in Section 7.2.2.1.

The shared memory footprint is correspondingly reduced since special actors no longer require memory allocations. Figure 7.6b shows an improvement of up to 38% resulting from the optimization.

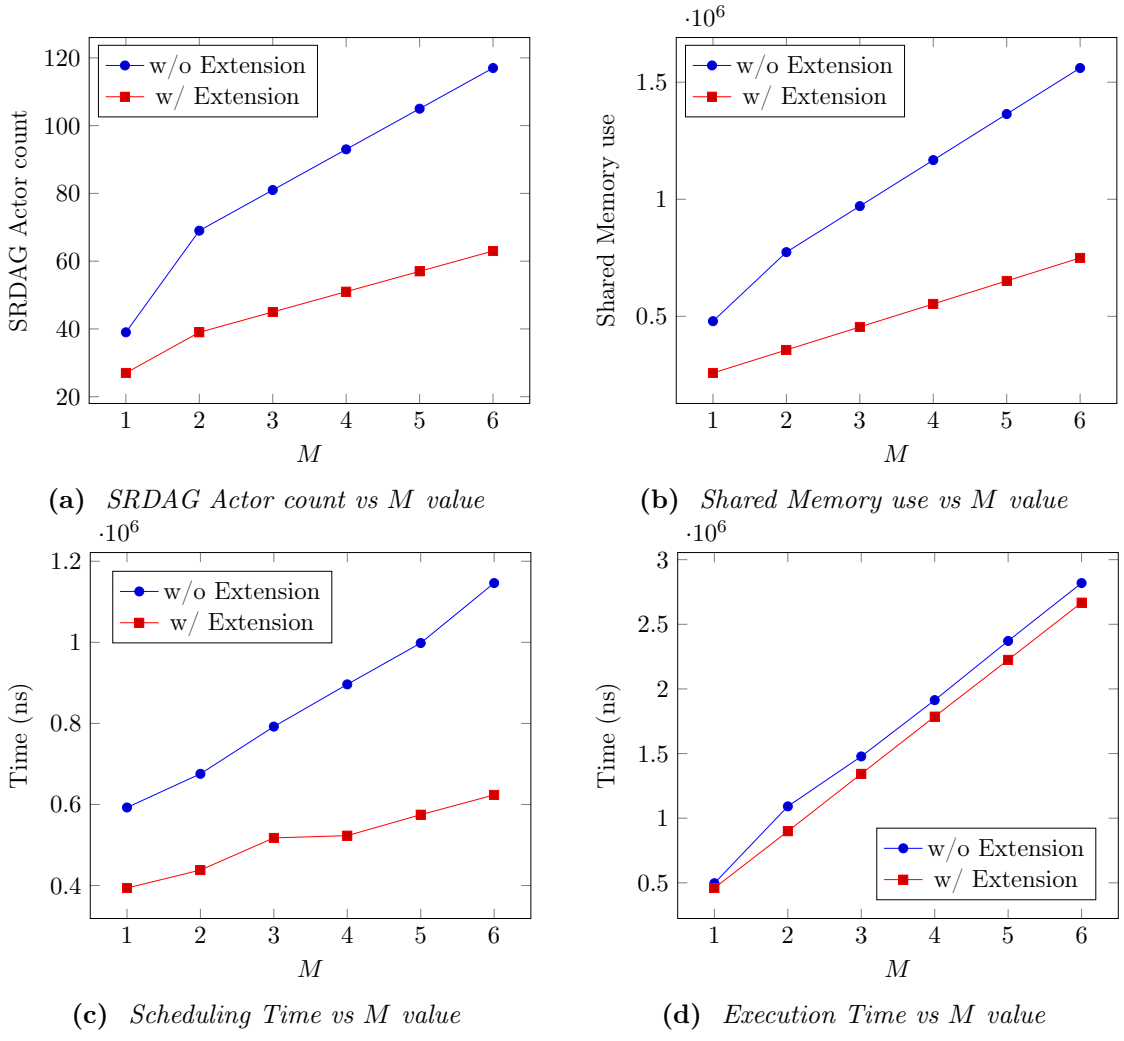


(a) *Fired SRDAG Actor count vs N value* **(b)** *Shared Memory use vs M value*

Figure 7.6 – *Special Actor Optimizations Benchmark*

7.2.2.4 PiSDF MoC Extension

In Section 5.2, a PiSDF extension was proposed to handle the initial values of delays. In this section, this extension will be added to the PiSDF MoC, as described in Section 7.2.1. The enhanced model will then be contrasted with the classical PiSDF MoC of the HCLM-Sched benchmark.



(a) *SRDAG Actor count vs M value*

(b) *Shared Memory use vs M value*

(c) *Scheduling Time vs M value*

(d) *Execution Time vs M value*

Figure 7.7 – *PiSDF Extension Benchmark*

The first metric examined is the SRDAG actor count. Figure 7.7a highlights the difference between actor counts without and with the extension. A reduction by up to 46% of SRDAG actor count is observed in this figure. This is due to the fact that the extension eliminates the need for *Switch* actors in the graph. For $N = M = 6$, the SRDAG actor count in Figure 7.5a is reduced from 189 (with no optimization) to 63 (for the optimizations of post-single rate transformation and PiSDF extensions). This is an overall reduction of 67%.

The PiSDF extension shows an improvement in terms of shared memory footprint. Figure 7.7b illustrates the shared memory footprint for the cases with and without optimization extensions. A reduction of up to 54% is observed. The shared memory footprint with $N = M = 6$ (Figure 7.5a) is reduced from 3.56 MB (with no optimization) to 732 kB

(for the optimizations of post-single rate transformation and PiSDF extensions). This is an overall reduction of 80%.

Next, the overall scheduling time is benchmarked. This timing includes the Mapping/Ordering task, and in the case of optimization, the time necessary to perform the single-rate optimization. An improvement of up to 46% is observed in the optimized case, as is seen in Figure 7.7c.

Finally, the execution time of the transfers though the FIR channels is displayed in Figure 7.7d. This time corresponds to the global application time less the scheduling time. Since the version with the extension has removed all the *Switch* actors in the SRDAG, this case has a lower execution time, as shown in the figure. An improvement of up to 17% is observed for this benchmark.

As is demonstrated in this section, the combined optimizations of post-single-rate transformation, Actor Precedence, Special Actor Optimization and PiSDF MoC extension allow the JIT-MS methodology and Spider to perform more efficiently at runtime. The proposed PiSDF extension significantly reduces both scheduling and execution times of an application that uses feedback loop delays.

7.2.3 Comparison of Spider with OpenMP

OpenMP (Open Multi-Processing) is an API that supports parallel multi-platform programming in C, C++ and Fortran [Ope15a]. The OpenMP API provides a model for parallel programming that is portable across shared memory architectures from different vendors.

The basic parallelization pattern of OpenMP, which uses thread forking, is illustrated in Figure 7.8. OpenMP programs are comprised of sequential and parallel regions. These regions are delimited by synchronization points. The HCLM-Sched benchmark is a good application for OpenMP because its global structure is a single parallel loop.

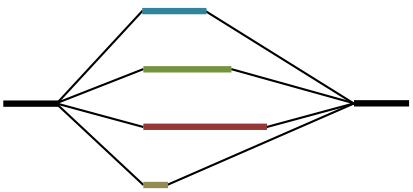


Figure 7.8 – *OpenMP Thread Representation*

The OpenMP framework can not implement the HCLM-sched as a double nested loop because the FIR filters are pipelined on each channel. There is thus a data dependency between two FIR filters of the same channel resulting in sequential computation. These data dependencies can not be implemented by an OpenMP "parallel for" compiler directive. However, the OpenMP framework is used to parallelize channels converting them into monolithic tasks. The OpenMP code is given in Listing 7.9. As a consequence of the internal loop sequentiality, there is no OpenMP #pragma involving the j variable.

Both the OpenMP and Spider on the HCLM-Sched algorithm were employed to benchmark the performance of three execution patterns: homogeneous, decreasing and increasing patterns.

```
static float pingpong[2*NBSAMPLES];
#pragma omp parallel for private(j,pingpong) schedule(dynamic)
for(i=0; i<n; i++){
        float *int_in, *int_out;

        /* Fetching input data */
        int_in  = input + i*NbS;
        int_out = pingpong;

        for(j=0; j<M[i]; j++){
                /* If last, save data into output array*/
                if(j == M[i]-1)
                        int_out = output + i*NbS;

                FIR(NbS, 0, int_in, int_out);

                int_in = pingpong + (j%2)*NbS;
                int_out = pingpong + ((j+1)%2)*NbS;
        }
}
```

Figure 7.9 – *OpenMP Code of the HCLM-Sched Benchmark*

7.2.3.1 Homogeneous Pattern

For the homogeneous pattern, the value of parameter M is identical for each channel. In these experiments, M is fixed to 12. The number of channels N is increased from 1 to 17. The resulting execution time (including scheduling time) is plotted in Figure 7.10 for both OpenMP and SPIDER.

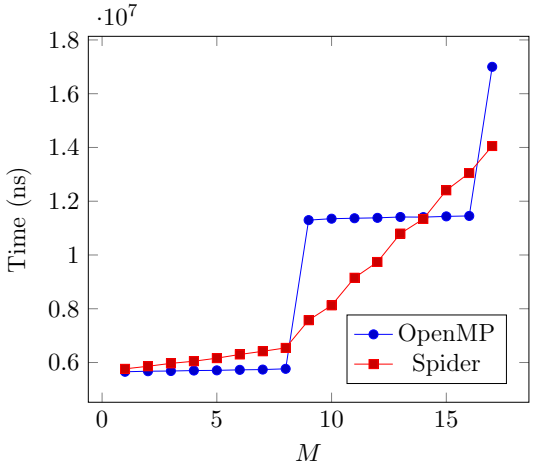


Figure 7.10 – *Overall Execution Time vs N value in homogeneous pattern*

As can be seen from Figure 7.10, the OpenMP execution time is constant at $6ms$ until $M = 8$. This is due to the fact that the number of PEs in the system is equal or greater than the number of channels in the graph. So all channels are executed in parallel, and there is no additional cost. The same phenomenon appears from $M \in [9, 15]$ since the

maximum computation time occurs with PEs executing two channels as it can be seen in Figure 7.11a. For the SPIDER implementation, the scheduling method computation creates a small overhead. This cost grows at an increasing rate with a higher number of channels because a bigger SRDAG has to be processed.

The Gantt chart of the execution for N equal to 9 can be seen in Figure 7.11a. In both Gantt charts, *FIR* of the same level have the same color independent of channel. Since channels are implemented as monolithic task, OpenMP cannot efficiently perform the computation. The OpenMP implementation thus executes two channels on the same core. With the use of SPIDER, a more efficient scheduling is applied since the FIR actors are considered as independent tasks. The result is displayed in Figure 7.11b. In the HCLM-sched benchmark with 9 channels, the overall latency is then reduced to up to 33%.

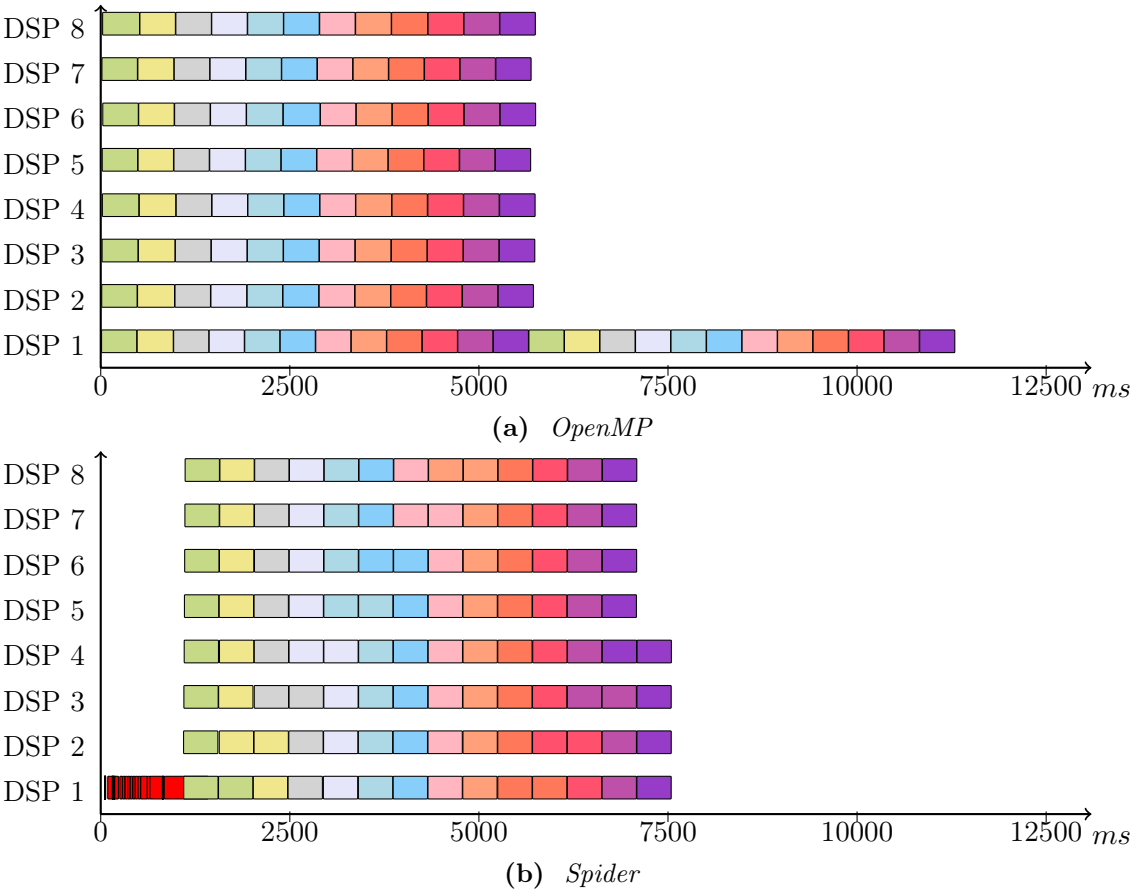
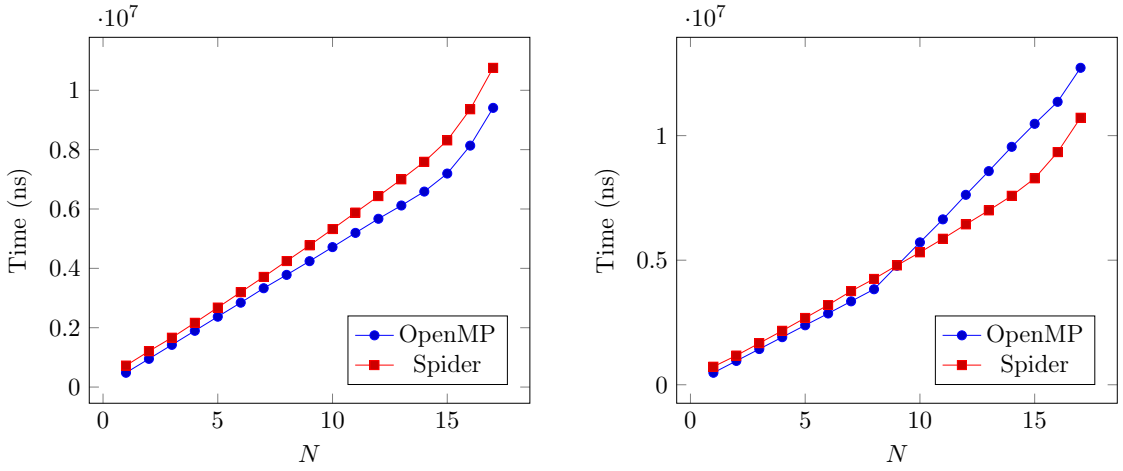


Figure 7.11 – *Gantt Chart of Homogeneous pattern with $N = 9$ and $M = 12$*

Decreasing and Increasing Patterns

The next two pattern types used are the decreasing and increasing patterns. The decreasing pattern is obtained by allowing the value of M to range from N to 1 for each channel. The increasing pattern has the opposite behavior: allowing the value of M to range from 1 to N for each channel. Overall execution time is then computed with N swept from 1 to 17.

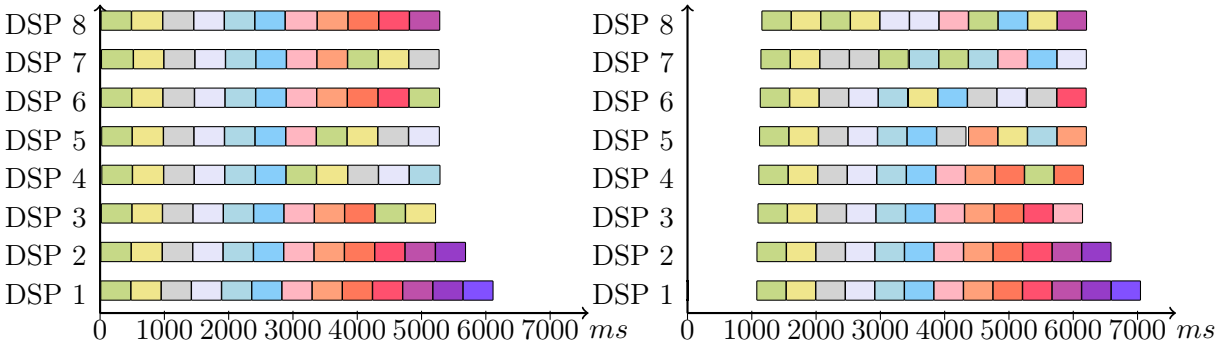
In Figure 7.12a, the decreasing pattern execution timings are compared for the two algorithm implementation. The OpenMP implementation is seen to be faster for all N,



(a) *Decreasing pattern* **(b)** *Increasing fromPattern*

Figure 7.12 – *Overall Execution Time vs N value*

since the scheduling time is lower. This is due to the fact that the OpenMP runtime always dispatches task in order. For this particular pattern, where M decreases, the longer tasks (with greater M) will be scheduled first, resulting in efficient scheduling decisions. As is shown in Figures 7.13a and 7.13b, the Gantt charts obtained from the decreasing pattern type for both multicore scheduling methods are equivalent in computation time. The difference is due to a later scheduling time for the SPIDER implementation. This has the overall result that SPIDER is slower by about 17%, on average.



(a) *OpenMP* **(b)** *Spider*

Figure 7.13 – *Gantt Chart of Decreasing pattern with* $N = 13$ *and* $M = 13..1$

In Figure 7.12b, the increasing pattern execution timings for both implementations are plotted. In this case, SPIDER is faster than OpenMP for N values greater than 10. This is the result of OpenMP dispatching tasks in order, meaning that smaller channels are scheduled first. Thus, the outcome is not efficient in scheduling decisions. As shown in the Gantt charts of Figures 7.14a and 7.14b, SPIDER has better scheduling of tasks than OpenMP for the increasing pattern. These better scheduling capabilities of SPIDER are obtained by the SRDAG computation giving an overall view of the application execution. It can be seen that for the increasing pattern, the overall latency is reduced by up to 21% (when $N = 15$).

As demonstrated in this benchmark comparison with the OpenMP scheduling method, SPIDER has a non-negligible scheduling overhead. However, the scheduling decisions taken

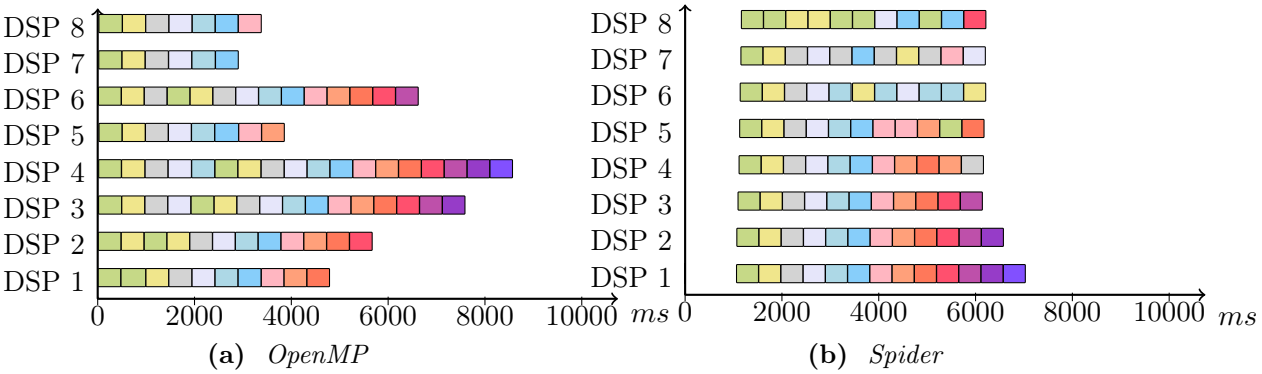


Figure 7.14 – *Gantt Chart of Increasing pattern with* $N = 13$ *and* $M = 1..13$

by Spider at runtime result in superior performance than a Texas Instruments OpenMP implementation on Keystone II for certain cases.

7.3 Parallel Discrete Fourier Transform Algorithm

7.3.1 Context

The Discrete Fourier Transform (DFT) is a common digital signal processing tool used in several domains in science and engineering. It is used in all fields of signal processing, from communications to radar applications. In our context, the computation of a DFT is used for large numbers of input samples. Moreover, the DFT computation is usually only a part of a larger application that could be globally managed by Spider. A DFT on complex samples is considered here.

7.3.2 DFT Theory

Considering $X = (X_0, X_1, \cdots, X_{N-1})^T$ the input vector of N samples and $F = (F_0, F_1, \cdots, F_{N-1})^T$ the vector of DFT output samples, the general DFT equation is :

$$\begin{aligned} F_k &= \sum_{n=0}^{N-1} X_k . e^{\frac{-2j\pi kn}{N}} \\ &= \sum_{n=0}^{N-1} X_k . W_N^{kn} \end{aligned}$$

where $W_N = e^{\frac{-2j\pi}{N}}$

Another way to represent the DFT equation using a matrix vector multiplication:

$$\begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ \vdots \\ F_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W_N & W_N^2 & \cdots & W_N^{N-1} \\ 1 & W_N^2 & W_N^4 & \cdots & W_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W_N^{N-1} & W_N^{2(N-1)} & \cdots & W_N^{(N-1)(N-1)} \end{bmatrix} \times \begin{bmatrix} X_0 \\ X_1 \\ X_2 \\ \vdots \\ X_{N-1} \end{bmatrix}$$

With the matrix representation, it is easy to see that the DFT algorithm requires N^2 complex multiplications and $N(N-1)$ complex additions.

The fundamental approach of all DFT fast algorithms is that of 'divide and conquer'. Dividing the problem into several subproblems that are simpler to solve results in :

$$\sum cost(problem) > cost(subproblems) + cost(merging)$$

In order to divide the problem into simpler subproblems, Cooley and Tukey [CT65] decimated the DFT by considering $k = n_2N_1 + n_1$ with $N = N_1N_2$. In this way, the periodicity of the input sequence is conserved:

$$\begin{aligned}
F_k &= \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} X_{n_2N_1+n_1} W_N^{k(n_2N_1+n_1)} \\
\Leftrightarrow F_k &= \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} X_{n_2N_1+n_1} W_N^{kn_2N_1} W_N^{kn_1} \\
\Leftrightarrow F_k &= \sum_{n_1=0}^{N_1-1}\sum_{n_2=0}^{N_2-1} X_{n_2N_1+n_1} W_{N_2}^{kn_2} W_N^{kn_1} \\
\Leftrightarrow F_k &= \sum_{n_1=0}^{N_1-1} W_N^{kn_1} \sum_{n_2=0}^{N_2-1} X_{n_2N_1+n_1} W_{N_2}^{kn_2}
\end{aligned}$$

Then they express k as $k = k_1N_2 + k_2$ where $k_1 \in [0, N_1[$ and $k_2 \in [0, N_2[$.

$$\begin{aligned}
F_{k_1N_2+k_2} &= \sum_{n_1=0}^{N_1-1} W_N^{(k_1N_2+k_2)n_1} \sum_{n_2=0}^{N_2-1} X_{n_2N_1+n_1} W_{N_2}^{(k_1N_2+k_2)n_2} \\
\Leftrightarrow F_{k_1N_2+k_2} &= \sum_{n_1=0}^{N_1-1} W_N^{k_1N_2n_1} W_N^{k_2n_1} \sum_{n_2=0}^{N_2-1} X_{n_2N_1+n_1} W_{N_2}^{k_1N_2n_2} W_{N_2}^{k_2n_2} \\
\Leftrightarrow F_{k_1N_2+k_2} &= \sum_{n_1=0}^{N_1-1} W_{N_1}^{k_1n_1} W_N^{k_2n_1} \sum_{n_2=0}^{N_2-1} X_{n_2N_1+n_1} W_1^{k_1n_2} W_{N_2}^{k_2n_2}
\end{aligned}$$

Since $W_N^K = e^{\frac{-2j\pi K}{N}}$, $\{k/N \in \mathbb{R}\} \Rightarrow W_N^K = 1$. So $W_1^{k_1n_2} = 1$

$$\Leftrightarrow F_{k_1N_2+k_2} = \sum_{n_1=0}^{N_1-1} W_{N_1}^{k_1n_1} W_N^{k_2n_1} \sum_{n_2=0}^{N_2-1} X_{n_2N_1+n_1} W_{N_2}^{k_2n_2}$$

If the following parameters are defined:

$$\begin{aligned}
F_{r,c} &= F_{rN_2+c} \\
X_{r,c} &= X_{rN_2+c} \\
X^T_{r,c} &= X_{rN_1+c}
\end{aligned}$$

The following steps can be obtained with two intermediate expression Y and Y':

$$Y_{n_1,k_2} = \sum_{n_2=0}^{N_2-1} X^T_{n_1,n_2} W_{N_2}^{k_2 n_2}$$
$$Y'_{n_1,k_2} = Y_{n_1,k_2} W_N^{k_2 n_1}$$
$$F_{k_1,k_2} = \sum_{n_1=0}^{N_1-1} Y'_{n_1,k_2} W_{N_1}^{k_1 n_1}$$

So to execute the complete “divide and conquer” DFT algorithm, three steps are:

1. N_1 DFTs of size N_2 on the input vectors.
2. Multiply the output of the first step with twiddle factors.
3. N_2 DFTs of size N_1 on the row vectors of the second step output.

7.3.3 Test algorithms and PiSDF representation

In this "divide and conquer" method, the choice of N_1 and N_2 is important. We consider only DFTs of lengths equal to the power of two's: $N = 2^n$. These transforms are frequent in signal processing because they ideally use the addressing capabilities of DSPs. We can derive 4 algorithms from this choice, which will all be described in the following subsections.

7.3.3.1 The Six-Step FFT

This method chooses N_1 and N_2 to be as close as possible to each other. The fundamental idea is to evenly distribute the required computation into both sets of DFTs.

In [Bai89], Bailey proposes an algorithm that provides fast computing on external and hierarchical memories. This algorithm is known as the 6-Steps FFTs.

This algorithm is composed of the following steps:

1. Transpose input samples considered as a $N_1 \times N_2$ matrix,
2. Perform N_1 unrelated N_2-points FFTs on the transposed matrix,
3. Multiply the resulting matrix with twiddle factors,
4. Transpose this matrix as a $N_2 \times N_1$ matrix,
5. Perform N_2 unrelated N_1-points FFTs on the transposed matrix,
6. Transpose this matrix as a $N_1 \times N_2$ matrix.

The major advantage of this method is that the memory accesses are consecutive when performing the FFTs as the algorithm operates on rows of matrices. The computation will then need less cache transfers and will perform better on embedded platforms.

In the SDF Model of Computation, interleaved access on different tokens of a Fifo is not permitted. This is also the case for the PiSDF MoC. This is why the matrix transposition implantation is required.

A PiSDF representation of the 6-Step algorithm can be found in Figure 7.15. Each step is represented by a single actor.

In this representation, the choice is made to permit the possibility of grouping multiple FFT operations into one actor. This is manageable via 2 parameters: one for each FFT

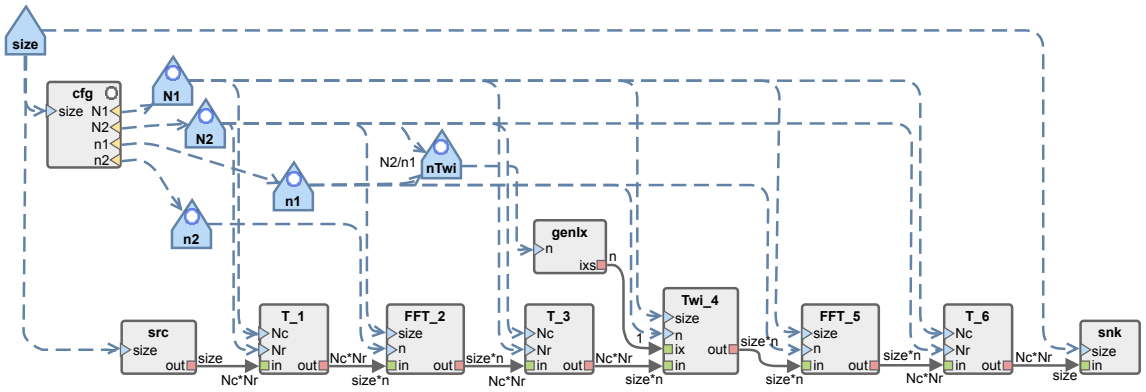


Figure 7.15 – *PiSDF Representation of the 6-Step FFT algorithm*

step. In this way, the number of single-rate actors can be maintained to a reasonably low level, reducing the runtime overhead and lowering the number of synchronization points needed in the system. The precomputed BRV can be found in Table 7.1.

Actor	Repetition
cfg	1
src	1
genIx	1
T_1	1
FFT_2	N_1/n_2
T_3	1
Twi_4	N_2/n_1
FFT_5	N_2/n_1
T_6	1
snk	1

Table 7.1 – *BRV Table of the 6-Step Algorithm*

In the PiSDF graph, the actor called *genIx* is used to apply the necessary twiddle factors to the incoming data of the twiddle actor. These factors are precomputed and stored in local memory. It is this index that allows the twiddle actor to apply the correct multiplication factor.

An example of a derived SRDAG for this PiSDF algorithm is shown in Figure 7.16. As can be seen, the PiSDF representation generates multiple iterations of the FFT and Twiddle actors, which are then executed in parallel. However, the transpose is a monolithic task that must be done sequentially. This is the only possible representation in PiSDF MoC.

7.3.3.2 Radix-2 FFT

Another approach which has been explored in literature is the Radix-2 approach. In the this approach, either N_1 or N_2 is fixed to 2, and the other is then set to 2^{n-1}. If N_1 is set to 2, the input samples are separated into 2 vectors; the first one containing the even samples and the second containing the odd samples. In this situation, this method is called Decimation in Time (DIT). However, if we choose $N_2 = 2$, the algorithm is called Decimation in Frequency (DIF). Representations of both DIT and DIF are shown in Figure 7.17.

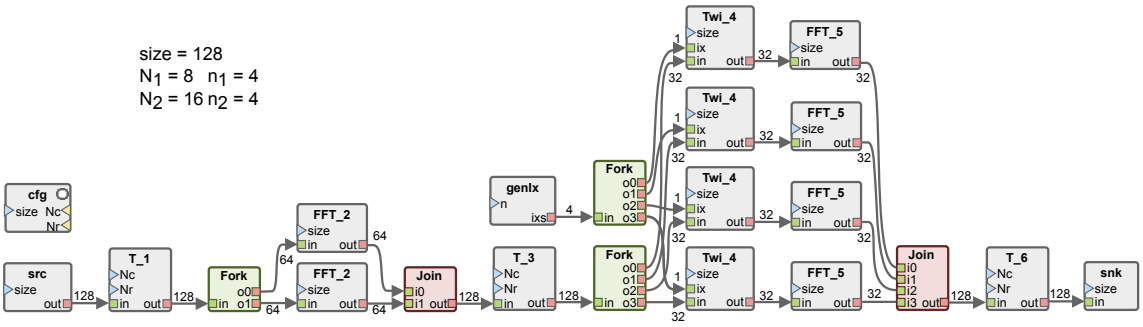


Figure 7.16 – *SRDAG derived from the 6-Step PiSDF Graph*

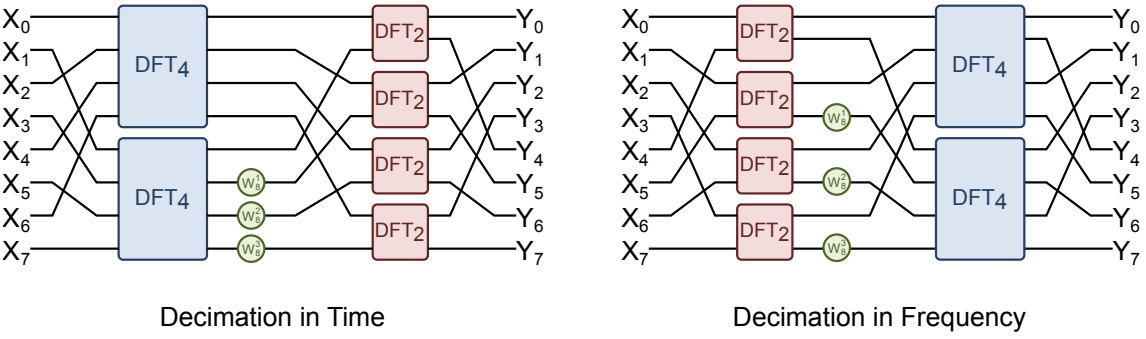


Figure 7.17 – *Decimation Possibilities in Radix-2 Approach*

The advantage of the Radix-2 method is that a DFT of size 2 does not require twiddle multiplications but only one addition and one subtraction for each sample.

For the PiSDF representation, the 6-Step representation can be reused. However, as the second FFT step is fixed with size 2, these inputs can be shown explicitly in the graph. In this way, the PiSDF MoC performs the second and third transpose. The PiSDF representation of the DIT FFT is shown in Figure 7.18.

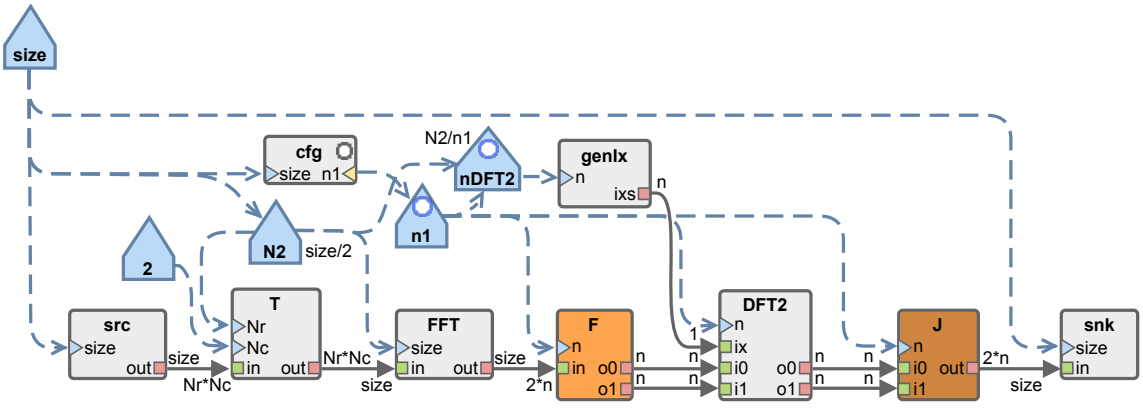


Figure 7.18 – *PiSDF Representation of the DIT Radix-2 FFT*

However, this graph simplification cannot be applied to the 6-Step algorithm. If this simplification was applied, it would lead to the creation of a FFT actor with a variable number of input and output FIFOs. This cannot be represented within the PiSDF MoC specification. The BRV of this graph is displayed in Table 7.2.

Actor	Repetition
cfg	1
src	1
genIx	1
T	1
FFT	2
DFT_2	N_2/n_1
snk	1

Table 7.2 – *BRV Table of the DIT Radix-2 FFT algorithm*

Exploring the DIF is outside the scope of this work. Only the DIT variant has been included. The DIT variant is preferred as the first data reordering computation can be considered as a pre-processing step [BYB08].

Currently, as shown in the BRV table, the PiSDF representation only provides an actor parallelism of 2 on the first FFT step of the graph. To improve the potential parallelism, the same decomposition of the FFT is applied on the FFTs of first step. A PiSDF representation that decomposes P times is shown in Figure 7.19. To handle the final bit reverse of the DIT algorithm, a preprocessing bit reordering is performed at the beginning of the dataflow graph, which is then merged with the transpose actor into a single actor.

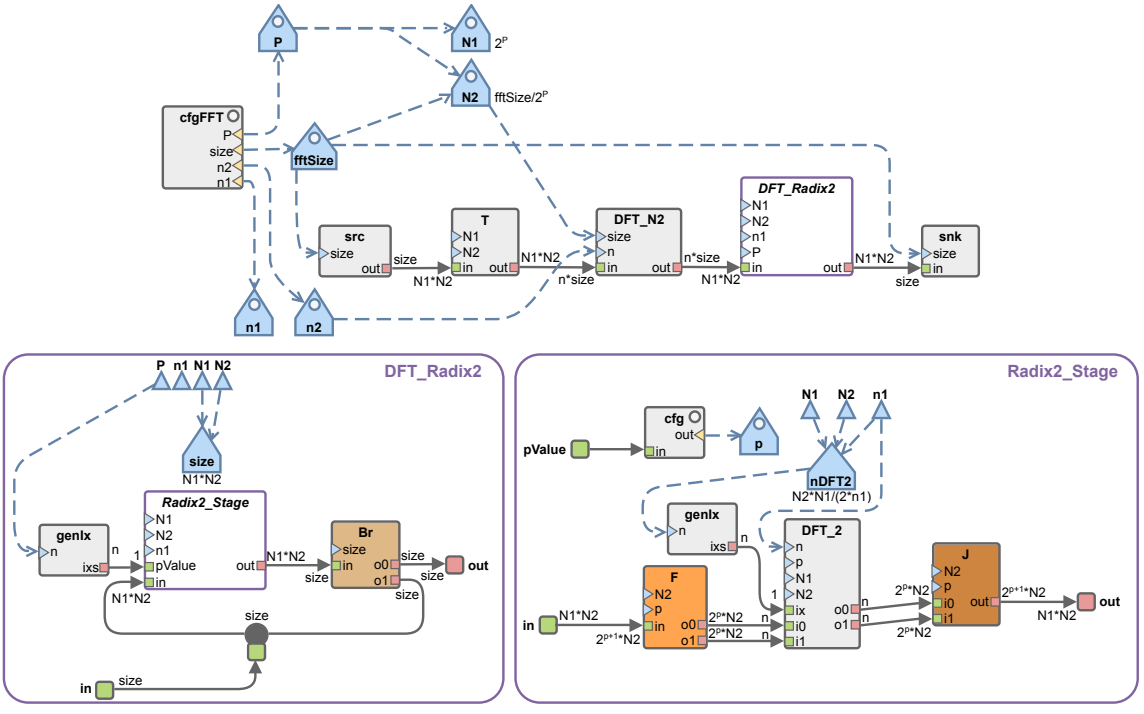


Figure 7.19 – *PiSDF Representation of the P Times Decomposition*

The first hierarchical actor used has the same pattern as that of the HCLM-Sched benchmark. The feedback loops provide a pipeline with dynamic length of *Radix2Stage* actor. Then, the Fork/Join pattern in the *Radix2Stage* subgraph permit the PiSDF dataflow MoC to recreate the classical butterfly shape of FFT. The precomputed BRV tables of the different hierarchy levels are displayed in Figure 7.20.

Actor	Repet.
cfgFFT	1
src	1
T	1
DFT_N2	N_1/n_2
DFT_Radix2	1
snk	1

(a) *Top Graph*

Actor	Repet.
GenIx	1
Radix2_Stage	P

(b) *Radix-2 DFT Subgraph*

Actor	Repet.
cfg	1
genIx	1
DFT_2	$N_2 * N_1/(2 * n_1)$

(c) *Radix-2 Stage Subgraph*

Figure 7.20 – *Precomputed BRV Tables of graph from Figure 7.19*

Both the 6-Step and Radix-2 implementations of the FFT application are benchmarked in the following section.

7.3.4 Experimental results

A complex FFT has been assumed for these experiments. The results have been benchmarked on the Keystone II platform. The large majority of the computation employs the DSP cores, however, the GRT which applies the scheduling method is run on one of the ARM processors. An FFT of 128k 16 bits fixed point complex samples is used in this benchmark.

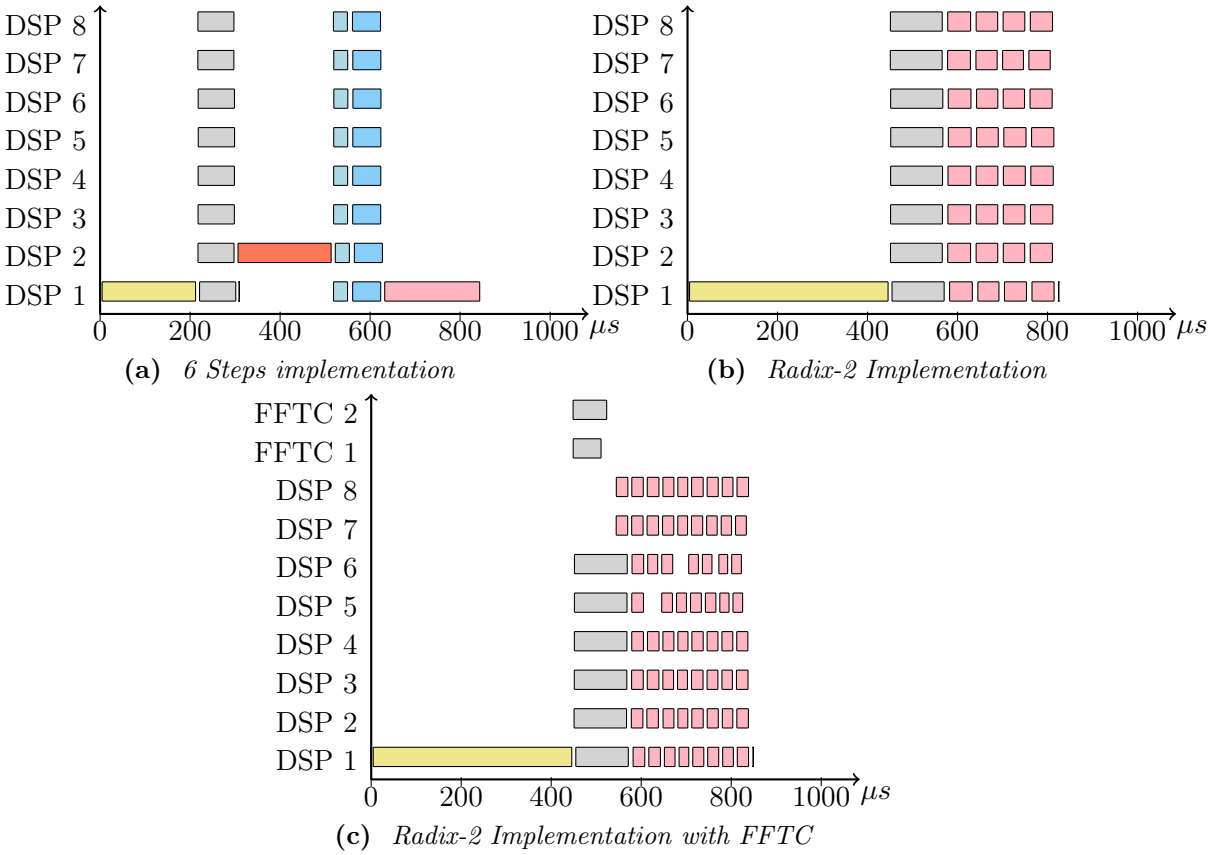
In this section, the scheduling time is not included. This is because the scheduling operates on a separate core which is never used for computation, and so the scheduling is pipelined with the computation. The consequence is that the next graph iteration is scheduled during the current graph iteration. This supposition is valid since scheduling takes around 700 μs for all algorithms. This is less than the graph execution time. Multiple schedules can even be processed simultaneously using the other ARM cores embedded in the platform.

As shown in Figure 7.21a, the Gantt chart of the 6 step algorithm is particularly costly in the transpose operation (yellow, orange and pink tasks). This implementation uses the embedded EDMA for these operations. However, the parallel execution of the FFT and the twiddle actors are very efficient.

The corresponding Gantt chart of the Radix-2 algorithm execution is displayed in Figure 7.21b. In this algorithm, the initial bit reordering actor is lengthy despite the help of the EDMA. However, when finished, the rest of the computation can be executed in parallel on each DSP.

Since the first reordering is a sequential task executed by hardware components, it may be considered as preprocessing [BYB08]. Even if the computation time is comparable, this makes the Radix-2 algorithm more suitable for FFT execution than 6-Step algorithm algorithm in this context.

Finally, it may be noted that the FFTCs embedded in the platform are used to enhance the algorithmic performance. The FFTC is a hardware accelerator for computing FFTs on the Keystone II. To take full advantage of the high capability of the FFTCs, parameters have been optimized to include more tasks in the second FFT step (pink). This configuration fires certain second step actors on the idle DSPs immediately after the end of the FFTC computation. Even with the synchronization overhead from more actors in the second step, the use of FFTCs provides a gain of 2.5% in terms of execution performance. These experiments demonstrate the capabilities of SPIDER to manage heterogeneous platforms.



(a) *6 Steps implementation*

(b) *Radix-2 Implementation*

(c) *Radix-2 Implementation with FFTC*

Figure 7.21 – *Gantt Charts of Parallel FFT Algorithms*

7.4 Stereo Matching Algorithm

A Stereo matching algorithm generates a disparity map from a pair of images by matching pixels from the left and right images. This disparity map is also called a depth map of the scene. For the current work, a dense local stereo matching algorithm [MSZ11] has been chosen.

This stereo matching algorithm is based on the computation of an error. This error is computed for all matching possibilities and is used to identify the most likely pixel in the other image that matches the current one. The difference between these pixels is called disparity.

The matching error is computed from the correlation of the two pixels (and their neighborhood) and is computed using a census algorithm. This error is then iteratively refined with a bilateral filter. This refinement step has the highest computing cost. It may be noted that since these errors are computed independently, this stereo matching algorithm can be easily parallelized.

As the errors are computed for each matching possibility (each disparity), the total execution time is linear with respect to the disparity range; hence reducing the disparity range will significantly decrease the total execution time. In order to optimize the algorithm and reduce its complexity without decreasing performance, the disparity range can be adjusted in real time using an algorithm like SIFT. The SIFT algorithm detects points of interest on an object or a scene. The disparity of these points of interest can be very quickly computed. An estimation of the disparity range is not within the scope of this work, but it illustrates another use for a fast configuration at runtime, such as SPIDER.

7.4.1 PiSDF Model

As described in [MSZ11], the stereo matching computation consists of four steps: cost initialization, cost aggregation, disparity computation and refinement. These same steps are used to describe the PiSDF graph of the application. As DSPs are targeted by SPIDER, the optimized implementation from [MPMN14] is used. The total graph for the stereo matching application is composed of a top graph (Figure 7.22) and two subgraphs (Figure 7.23 and Figure 7.24).

The cost initialization step, or cost construction, is comprised of two separate parts: census cost and truncated absolute difference cost.

The census cost in pixels of each images is computed by census actors: *Census_L*, *Census_R*. These actors take the gray scale in the right and left images. This computation is performed with the *RGB2Gray_L RGB2Gray_R* actors. The census operation produces an 8-bit signature for each pixel of an input image which is obtained by comparing each pixel to its 8 neighbors. If the value of any neighbor is greater than the value of the pixel, one signature bit is set to 1. If not, it is set to 0. This weight is correlated to local textures to ensure a good homogeneity of the results.

The truncated absolute difference cost simply computes the difference between the left and right gray scale images. This value is then truncated to remove some noise. This computation is performed in the *CostConstruction* actor of the *CostParallel* subgraph. This actor also sums the truncated difference cost with result of the census cost.

The second step of cost aggregation step is iterative. Since the cost construction step provides noisy matching cost maps, the cost aggregation step performs several passes on areas with the same color in the original image. The cost aggregation step is performed independently of each cost map and bilaterally, ie horizontally and then vertically. The weights of each pixel are precomputed by the *VWeight* and *WWeight* respectively for vertical and horizontal weights. Finally the costs are refined in the *AggregateCost* actor.

Actor	Repet.
Config	1
Camera	1
RGB2Gray_R	1
RGB2Gray_L	1
Census_R	1
Census_L	1
CostParallel	1
Split	1
MedianFilter	nSlice
Display	1

(a) *Top Graph*

Actor	Repet.
GenDisp	1
GenIx	1
CostConstruction	nDisp
HWeight	nIter
VWeight	nIter
DispComp	1

(b) *CostParallel Subgraph*

Actor	Repet.
AggregateCost	nDisp
DisparitySelect	nDisp

(c) *DispComp Subgraph*

Table 7.3 – *Precomputed BRV Tables of Stereo PiSDF Graph*

The disparity computation and refinement steps are made by the *DisparitySelect* and *MedianFilter* actors respectively. The *DisparitySelect* actor produces the disparity map by comparing the cost of each disparity and selecting that with the lowest value. This *DisparitySelect* actor is then fired *nDisp* times to perform the *nDisp* comparisons. The *MedianFilter* is then used to reduce some of the residual noise of the disparity levels. In the implementation of this thesis, the *MedianFilter* is parallelized by slicing the input

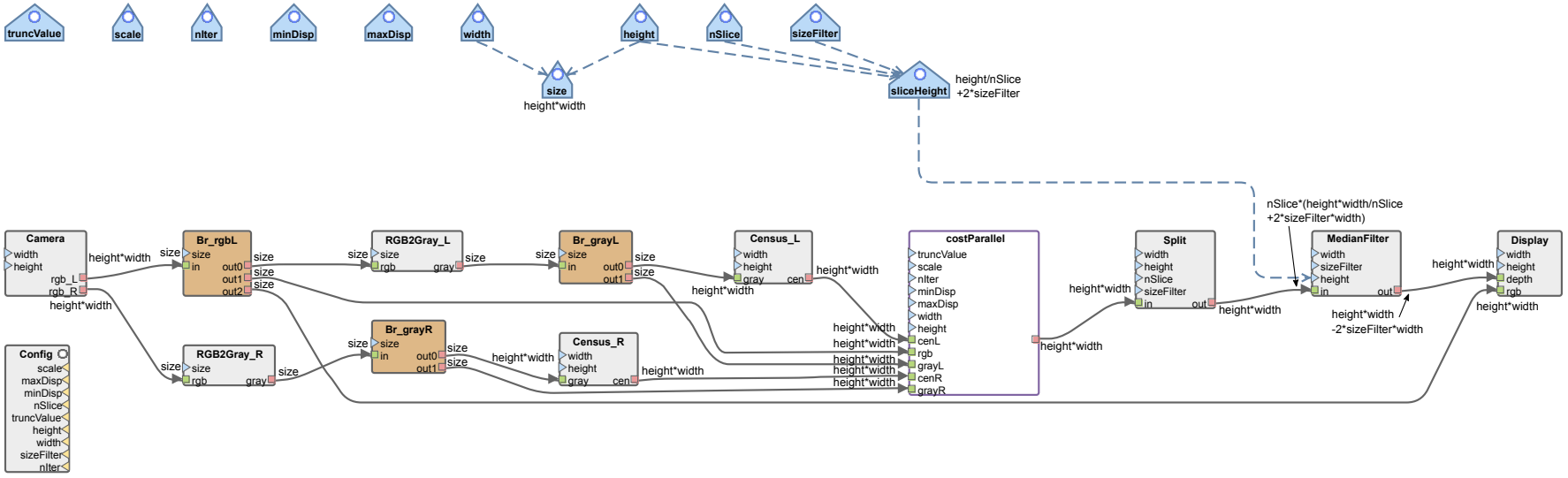


Figure 7.22 – *Top graph of the Stereo Matching algorithm*

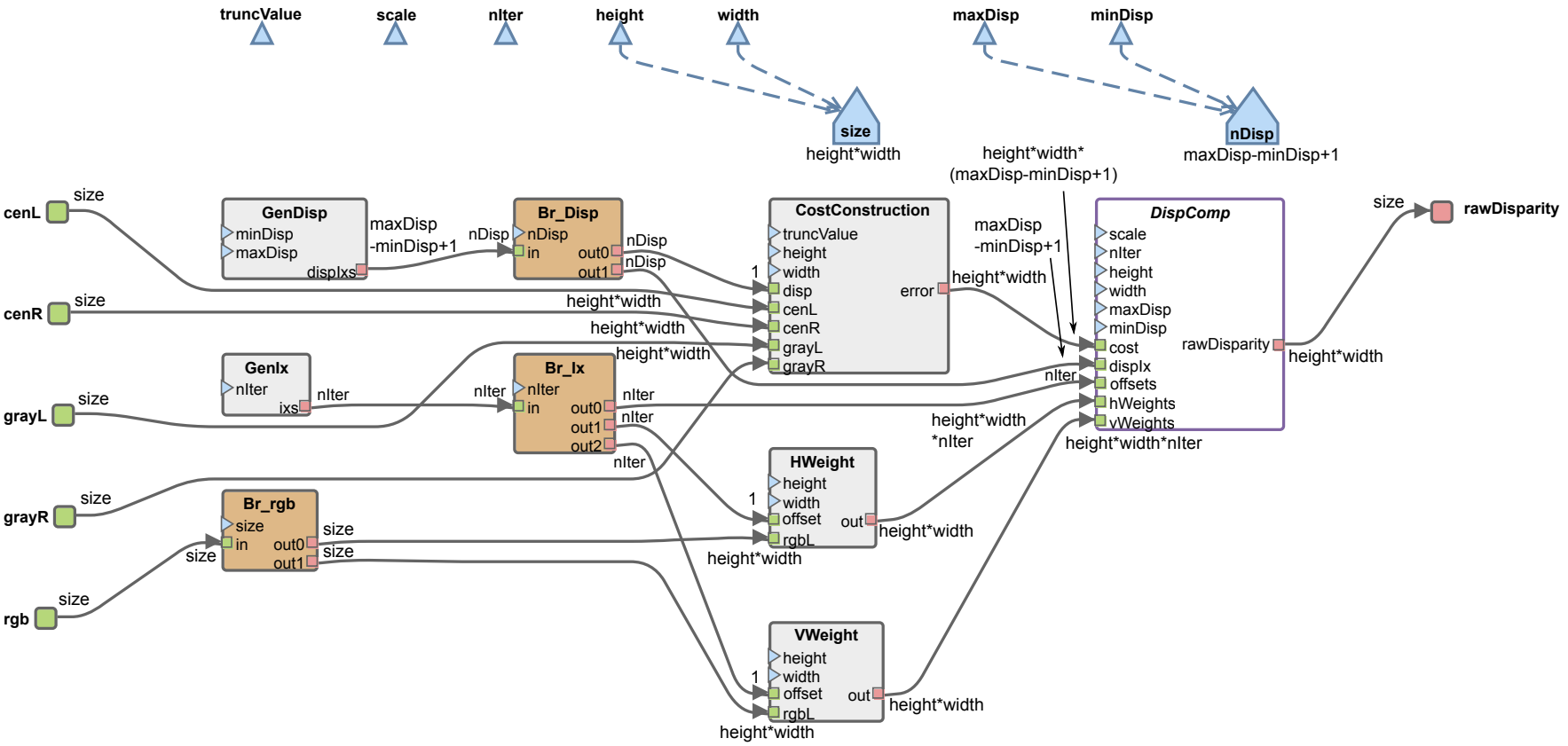


Figure 7.23 – *CostParallel Subgraph*

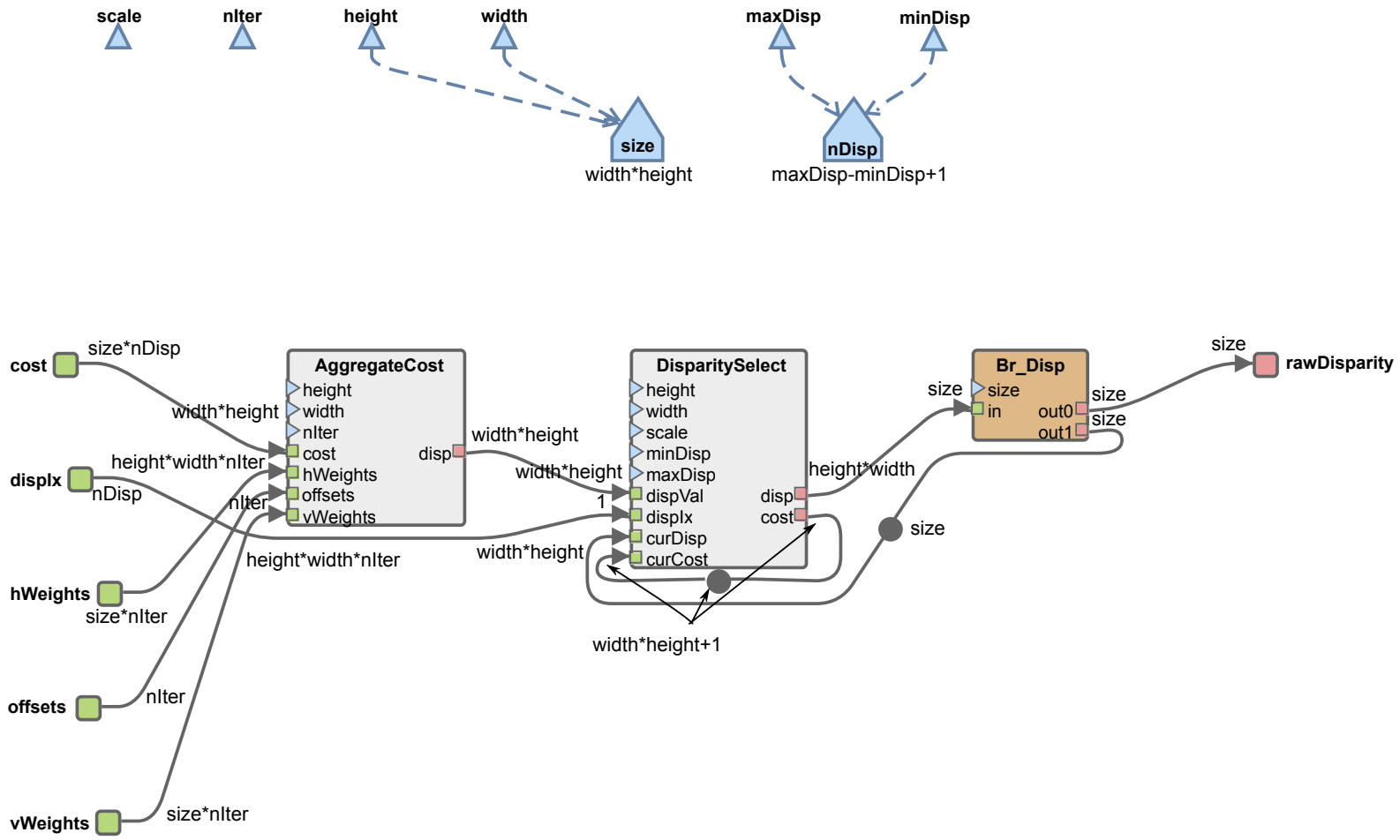


Figure 7.24 – *DisparityComp Subgraph*

images *nSlice* times. To perform the parallelization, the image is separated over the *nSlice* iterations, and then the border pixels of the slices are duplicated by the *Split* actor.

The BRV of all graphs from the stereo applications can be found in Table 7.3.

In this example, both hierarchical actors (*DispComp* and *CostParallel*) are fired only once. Hierarchy has been employed to use the RB behavior implicit on the input and output data interfaces. For the *CostParallel* subgraph, incoming tokens from the *grayL*, *grayR*, *cenL*, *cenR* interfaces are duplicated *nDisp* times. The same implicit RB behavior occurs with the weight interfaces in the *DispComp* subgraph. In *DispComp* subgraph, the output data interface of the *rawDisparity* actor filters output tokens, so that only the last tokens are retained, which give the disparity levels after the completion of the comparison of all levels.

7.4.2 Experimental results

The purpose of these experimental results is to demonstrate that Spider is suitable for the computer vision domain.

The experimental setup was as follows:

- TI's Keystone II platform was used
- All actors are run on DSPs
- External DDR was used, since the memory requirements were greater than the MSMC shared memory (approximately 30MB was needed in total).

The number of DSP cores employed varies: the experiment swept this variable from 1 to 8 producing the plots of Figure 7.25.

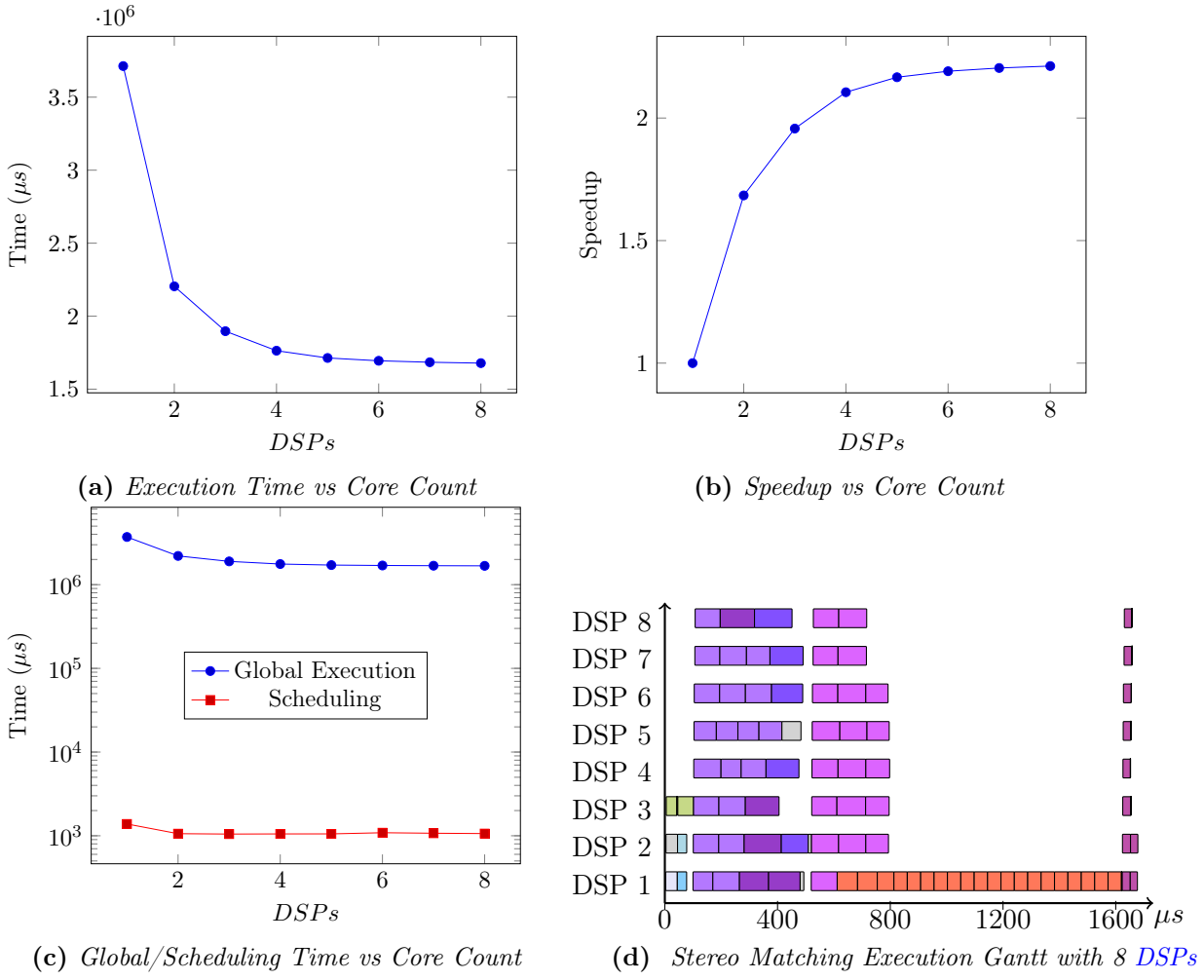
In Figure 7.25a and Figure 7.25b, the parallelization of the stereo application performance is benchmarked. In Figure 7.25d, the Gantt chart shows that the sequential disparity selection actors (in orange) are the cause of the low parallelization potential of the algorithm. Consequently, other actors are distributed to minimize the impact of the sequential disparity selection actors on the platform performance.

As shown in the Figure 7.25c, the scheduling time for this case study is negligible. The scheduling time represents up to 0.06% of the global execution time in this benchmark. This demonstrates that Spider has a negligible impact on performance when used for applications with actors which have long computation time. It can be then concluded that Spider is suitable for computer vision applications. The advantages of reconfiguration and good scheduling performance are balanced against by a negligible scheduling cost.

7.5 Conclusion

In this chapter, three applications are described using the PiSDF MoC. Each application uses the features provided by the PiSDF MoC including the interfaced hierarchy and the in-graph reconfiguration.

The three applications are used to benchmark the optimizations described in Chapter 5 which are all demonstrated to be real improvements in the scheduling method. A comparison with the OpenMP framework indicates that in certain cases, Spider has better performance than OpenMP even with an application that is a good match for the OpenMP parallelization process.



(a) *Execution Time vs Core Count*

(b) *Speedup vs Core Count*

(c) *Global/Scheduling Time vs Core Count*

(d) *Stereo Matching Execution Gantt with 8 DSPs*

Figure 7.25 – *Stereo Application Benchmark*

The three applications described in this chapter may also be seen to be suitable for use in benchmarking applications. They may be employed to benchmark any modifications to the MoC or to the scheduling method used in the PREESM/SPIDER framework.

Future work could include applying this description methodology to more complex applications such as the High Efficiency Video Coding (HEVC) decoder [SOHW12]. This application embeds a great number of data dependencies between tasks within a dynamic reconfiguration domain. Benchmarking this application with an efficient scheduling method such as JIT-MS may lead an improvement in speedup on similar applications.

Additionally, analyzing the impact of hierarchy on scheduling overheads is a motivating objective for application studies. Since smaller graphs are scheduled more quickly, it would be interesting to investigate the trade-off between complexity introduced by many hierarchy levels compared with the complexity resulting from scheduling bigger graphs.

CHAPTER 8

Conclusion

8.1 Summary

The recent evolution of embedded systems has resulted in the integration of an increasing number of processing elements into Multi-Processor Systems-on-Chip. Thus, new programming models and languages must be found to exploit the parallel processing capabilities of these new devices. For this purpose, the dataflow MoC has emerged as a popular programming technique to unveil the parallelism of applications.

This thesis studies programming techniques for embedded multicore devices and proposes a novel multicore scheduling method. This new multicore scheduling method is called JIT-MS and is based on a parameterized dataflow MoC called PiSDF [DPN+13]. It targets the efficient dispatch of applications at runtime on MPSoCs.

The new scheduling method of JIT-MS was introduced in Chapter 4. This scheduling method uses the PiSDF model to schedule actors when reconfiguration points are reached. This allows the exploitation of the parallelism of the application that are present at these reconfiguration points. The scheduling method is based on an intermediate graph called SRDAG that is employed to unveil all the task precedences of the application. The scheduling method iterates over multiple steps that correspond to the different hierarchy levels of the PiSDF description of the application.

Optimizations that can be applied to the JIT-MS method were presented in Chapter 5. These optimizations target on overall performance improvement, and are applied at different points in the multicore scheduling flow. Notably, a PiSDF MoC extension focusing on Fifo delay provides scheduling performance improvements in terms of shared memory footprint, scheduling overhead and application execution performance.

An embedded runtime system called Spider was then detailed in Chapter 6. This runtime embeds the JIT-MS method and is used to schedule PiSDF applications. Runtime and synchronization mechanisms are studied as well as certain runtime optimizations which reduce the number of synchronizations between cores. This runtime is designed to be portable and is implemented on a cutting edge embedded MPSoC: the TI Keystone II platform.

Finally, a case study of PiSDF programming was presented in Chapter 7. Benchmarks of the JIT-MS method and of the Spider runtime were presented. A study on the PiSDF programming of two signal processing applications and one vision application were pro-

vided, allowing the assessment of the performance of PiSDF programming. The Spider runtime is compared to the OpenMP implementation by Texas Instruments and the results demonstrate that better schedules can be obtained with a manageable overhead with Spider.

8.2 Future Work

The results of this thesis open new opportunities for future research on parameterized dataflow MoCs. Parameterized dataflow provides new possibilities to a software designer to describe the dynamic behavior of the application while retaining some predictability when compared to dynamic dataflow models. The scheduling overhead introduced by the runtime scheduling of PiSDF applications is shown to be manageable for signal and vision applications. As a consequence, many applications scheduled at runtime can benefit from parameterized dataflow programming. Potential future work can be divided into three directions, which are described below.

8.2.1 A Larger Range of Platforms

Future MPSoCs will integrate more PEs, making multicore scheduling even more complex. The current heuristics may not suit these platforms. A potential solution for scheduling applications on a large number of cores is the HFS Mapping/Ordering method.

A further consideration for MPSoCs with hundreds of PEs such as the Kalray MPPA [MPP15], is that a shared memory architecture would necessarily become the bottleneck of the system. Thus, distributed memory systems are likely to become the new trend in embedded MPSoCs. Since dataflow MoCs are not specific to either shared or distributed memory architectures, the use of dataflow MoCs will continue with this paradigm shift towards distributed memory MPSoCs. Hence, studying these new larger platforms with the JIT-MS method and the Spider runtime is a promising perspective.

8.2.2 A Larger Range of Applications

Currently, signal and video applications are becoming increasingly more complex and require significant computing power. Since the scheduling performance of Spider is demonstrated to be efficient when the scheduled actors are computationally intensive, the future use of Spider with these applications is a natural step. For example, the HEVC [SOHW12] video encoders and decoders are complex to parallelize and are composed of computationally intensive tasks. Studying these new applications with the JIT-MS method would make an interesting benchmark for this method.

8.2.3 Quasi-Static Scheduling Techniques

Spider is an embedded runtime that provides dynamic execution to dataflow applications. The content of this thesis could be used to evaluate the scheduling impact of any application. With this information, the programmer can decide if the overhead introduced by this adaptive management is acceptable for the considered application.

Quasi-static scheduling provides adaptive management to a system at a low cost when only a finite number of predefined cases can occur. This scheduling method is restricted to dynamic behaviors but has a very low scheduling overhead. Now that a dynamic scheduling method of PiSDF MoC has been designed, quasi-static scheduling should become straightforward making it an interesting perspective for applications with limited variations.

Finally, PREESM may be enhanced to become a complete Static/Quasi-Static/Dynamic framework that can provide valuable rapid prototyping metrics to the software designer for a very large set of applications. This allows the software designer to take informed decisions on the scheduling method of a given system, within the design constraints.

APPENDIX A

French Summary

A.1 Contexte

A.1.1 Les Systèmes Embarqués

L'évolution récente des systèmes embarqués a donné lieu à un grand nombre de progrès remarquables. Du contrôleur très complexe d'un moteur d'une Formule 1 FIA à un podomètre comptant chaque pas d'un piéton, les systèmes embarqués peuvent être trouvés un peu partout dans la société du début du 21ème siècle. Même un cœur humain, l'organe le plus vital, peut maintenant être remplacé par une machine [Car]. Un système embarqué est un système électronique et informatique intégré conçu dans un but précis. Les téléphones mobiles, tablettes, cartes de crédit sans contact, montres connectées et même les drones domestiques sont quelques-uns des innombrables appareils modernes contenant un ou plusieurs systèmes embarqués.

A.1.2 Les Contraintes des Systèmes Embarqués

Une spécificité majeure du processus de conception des systèmes embarqués est le fait que le développement soit contraint par un ou plusieurs critères précis. Ces contraintes peuvent provenir de diverses sources.

Tout d'abord, certaines contraintes résultent directement de l'application visée. Le temps de réaction d'un airbag dans un véhicule, la vitesse de décodage d'un lecteur musical ou encore la consommation d'énergie d'une sonde spatiale sont des exemples de contraintes que doit respecter un système embarqué.

Ces contraintes peuvent aussi être introduites par des considérations économiques. Passer des années de développement ou intégrer des appareils électroniques plus coûteux nécessitant des mises à jour logicielles fréquentes ne sont sans doute pas nécessaires à un beeper de garage.

Enfin, l'environnement physique peut introduire de nouvelles contraintes pour un système embarqué. Ces contraintes ne sont pas liées aux besoins, mais doivent être respectées pour assurer un fonctionnement sûr et durable. La haute pression subie par un système immergé profondément dans l'océan ou les perturbations causées par le rayonnement solaire reçu par une sonde spatiale sont des exemples de contraintes environnementales.

Ces diverses contraintes peuvent souvent être contradictoires, ou du moins difficiles à satisfaire simultanément. Par conséquent, des arbitrages sont établis lors du processus de conception du système embarqué. Par exemple, la réduction de la fréquence de fonctionnement d'une puce peut réduire ses performances, mais va également réduire sa consommation d'énergie.

A.1.3 Conception d'un Système Embarqué

Le processus de conception d'un système embarqué est souvent séparé en deux parties : la conception matérielle et logicielle.

L'objectif principal de la conception matérielle est d'adapter les ressources informatiques du système à l'application et à l'environnement. Capteurs, actionneurs et interfaces utilisateurs sont quelques-unes des nombreuses fonctionnalités qui peuvent être nécessaires dans un système embarqué. La conception matérielle fournit également les services nécessaires par le dispositif de calcul principal pour le fonctionnement du système. La génération de l'horloge, l'alimentation et l'accès à une mémoire externe sont des exemples de services matériels utilisés par la partie logicielle. L'objectif principal de la conception matérielle est d'intégrer tous ces composants et services ainsi qu'adapter leurs inter-connections, en décrivant l'architecture matérielle du système.

Les éléments de calcul d'un système sont désignés comme processeurs. Un processeur est un élément programmable qui peut exécuter une fonctionnalité de l'application. Les processeurs peuvent appartenir à différentes catégories: depuis les micro-contrôleurs de faible puissance aux GPPs et/ou DSPs avec une forte capacité de calcul. Actuellement, les systèmes embarqués peuvent intégrer plusieurs processeurs, créant ainsi un système multiprocesseur. Si les processeurs sont de types différents, le système est défini comme un système multiprocesseur hétérogène.

Pour piloter et exécuter la majeure partie de l'application, un logiciel est déployé sur les différents processeurs du système. Ces logiciels sont développés lors de la conception logicielle du système embarqué. Un logiciel, ou programme, est une liste d'instructions qui sont exécutées par un processeur. Ces instructions sont stockées dans une mémoire dédiée et sont traitées de façon séquentielle. Ces instructions sont écrites à l'aide d'un langage machine qui peut être différent pour chaque type de processeur.

Actuellement, les programmeurs ne programment généralement pas les processeurs en utilisant le langage machine, mais avec un langage de plus haut niveau. Java, Fortran, C/C++ et leurs dérivés sont tous des exemples de langages de programmation de haut niveau. Ces langages de haut niveau sont ensuite convertis en langage machine en utilisant des compilateurs dédiés, qui sont dépendant du processeur visé.

La conception des parties matérielles et logicielles sont étroitement liées et un choix effectué dans l'une des deux parties impose souvent des choix dans l'autre.

A.1.4 Systèmes embarqués parallèles

En 1965, Moore a prédit que le nombre de transistors dans un circuit intégré doublerait tous les deux ans [Moo65]. Malgré le fait que cette tendance ralentit actuellement, cette prédiction a été respectée pendant des années, rendant les systèmes de calcul de plus en plus complexes au cours du temps.

Pendant de nombreuses années, les systèmes informatiques ont fait l'objet du mythe dit du "mégahertz" [meg]. Il promeut l'idée qu'une fréquence d'horloge plus élevée produit une exécution plus rapide des tâches. Ce mythe a été maintenu par les fondeurs de puces pendant des décennies jusqu'à ce que les besoins en dissipation de chaleur des puces aient

augmenté de façon spectaculaire. Ainsi, à partir de la première décennie du 21e siècle, l'intégration d'un nombre grandissant de processeurs dans les systèmes multiprocesseurs est devenue la nouvelle solution pour augmenter continuellement la puissance des systèmes informatiques.

Actuellement, un domaine de recherche occupe une importance grandissante : les techniques de programmation parallèle. Puisque de plus en plus de processeurs sont intégrés dans les systèmes, il est devenu essentiel de concevoir des méthodes automatiques pour gérer la répartition des calculs et la synchronisation des différents processeurs. De nouvelles techniques de programmation ont vu le jour pour permettre aux programmeurs de gérer les systèmes à plusieurs processeurs plus facilement. L'une de ces techniques est appelée programmation flux de données.

A.1.5 Programmation Flux de Données

Les modèles de calcul flux de données ont été conçus pour représenter le parallélisme des applications. Les applications sont décrites en utilisant des blocs de calcul, appelés acteurs, qui échangent des données à l'aide de communications dirigées appelées canaux. Un avantage majeur de modèles flux de données est la possibilité de réutiliser le code existant pour exprimer le comportement des blocs de calcul. Le premier modèle flux de données utilisé dans la littérature a été introduit par Kahn [Kah74].

Dès lors, de nombreux modèles flux de données ont émergé dans la littérature. Ces modèles sont souvent basés sur un modèle pré-existant en ajoutant de nouvelles fonctionnalités ou en assurant certaines propriétés. Certains de ces modèles apportent la possibilité d'inclure des changements dynamiques dans la description de l'application. Ces changements dynamiques permettent d'ajouter ou de supprimer des tâches de calcul à l'exécution. Puisque le modèle est capable de détecter ces changements, de meilleures décisions peuvent être prises quant à la répartition de l'application sur les systèmes multiprocesseurs.

Le modèle utilisé comme point d'entrée de cette thèse est le modèle PiSDF. Ce modèle de calcul étend un modèle connu dans la littérature appelé SDF.

Le modèle PiSDF est défini formellement ci-dessous:

Definition A.1.5.1 (Définition du modèle PiSDF)
Un graphe PiSDF $G = (A, F, P, I, D)$ est composé de :

- *A, un set d'acteurs. Un acteur est un élément du graphe qui représente des calculs à effectuer.*
- *F, un set de Fifos, i.e. canaux de communications unidirectionnels. Les acteurs échangent des données appelés* jetons *via ces Fifos.*
- *P, un set de paramètres. Un paramètre est un élément du graphe qui est utilisé pour configurer l'application et modifier son comportement.*
- *I, un set d'interfaces hiérarchiques. Une interface est un élément du graphe qui permet la transmission de jetons ou de paramètres entre deux niveaux de hiérarchie.*
- *D, un set de dépendances. Une dépendance permet de propager les paramètres entre les différents éléments du graphe.*

Un exemple de graphe PiSDF est représenté sur la Figure A.1. Y sont représentés 5 acteurs: *GenN*, *Src*, *H*, *Snk* et *Filter*, et 2 paramètres: N et L.

L'acteur H est un acteur hiérarchique. Son comportement est ainsi modélisé par un autre graphe représenté sous celui-ci. Afin de connecter le graphe supérieur au graphe inférieur, des interfaces sont utilisées et représentées par des carrés de couleurs verte et rouge pour les ports de données entrants et sortants ainsi que par un triangle bleu pour les paramètres.

L'acteur *GenN* est un *acteur de configuration*. En effet, il permet de déterminer la valeur du paramètre N dynamiquement durant l'exécution du graphe. Les acteurs de configurations ne peuvent être exécutés qu'une seule fois par niveau de hiérarchie.

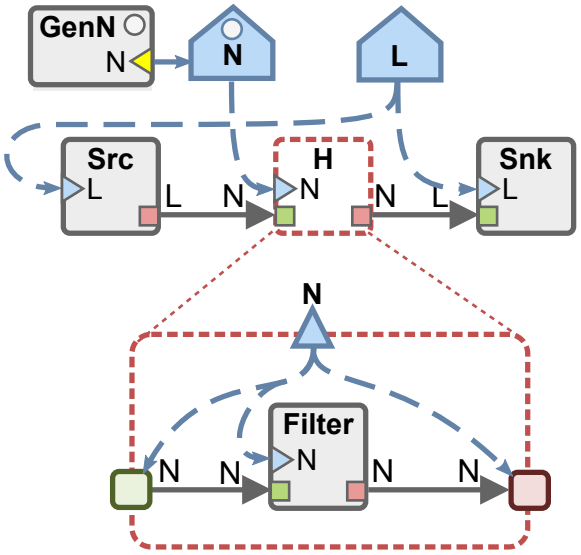


Figure A.1 – *Un graphe d'exemple PiSDF*

A.1.6 Contributions de la thèse

Lors de cette thèse, les techniques de programmation pour les systèmes embarqués multiprocesseurs ont été étudiées. Le point de départ est le modèle de flux de données PiSDF, utilisé pour représenter l'application et pour répartir efficacement les calculs sur des systèmes embarqués multiprocesseurs.

Ce modèle de calcul flux de données fournit des fonctionnalités permettant l'adaptation d'une application en modifiant des paramètres prédéfinis. Ces paramètres déclenchent des changements dynamiques dans la structure d'exécution de l'application qui sont traités à l'exécution.

Dans ce cadre, une nouvelle structure d'exécution en ligne a été développée pour effectuer une planification efficace d'applications PiSDF sur des systèmes embarqués multiprocesseurs. Les contributions de cette thèse sont énumérées ci-dessous:

- L'introduction d'une nouvelle méthodologie de programmation multiprocesseur. Cette méthode permet de révéler le parallélisme d'une application PiSDF et d'envoyer différentes tâches sur le système ciblé. Cette méthode est nommée JIT-MS.

- Diverses améliorations sur cette méthode ont été introduites. Ces améliorations comprennent une extension au modèle de calcul PiSDF ainsi que des optimisations sur la représentation intermédiaire utilisée lors de la méthode.

- Une nouvelle structure d'exécution appelée Spider est introduite. Spider effectue la méthode JIT-MS à l'exécution et exploite le parallélisme des applications PiSDF. Spider vise les systèmes embarqués multiprocesseurs hétérogènes et peut facilement être adaptée à plusieurs plateformes.
- Une étude de deux applications de traitement du signal et de vision par ordinateur sur un système embarqué multiprocesseur hétérogène. Ces applications utilisent Spider et une discussion sur leur représentation en PiSDF est aussi apportée.

Toutes les contributions ont été développées dans le cadre d'une collaboration scientifique entre le laboratoire IETR et Texas Instrument France, et dans le cadre du projet ANR COMPA.

A.2 JIT-MS: une Méthode d'Ordonnancement Multiprocesseur basée sur le Modèle de Calcul PiSDF

L'un des principaux défis de la conception et la réalisation de systèmes multiprocesseurs est de répartir efficacement les tâches de calcul sur les processeurs disponibles. La répartition peut inclure dans ses décisions les modifications dynamiques des fonctionnalités de l'application ou de ses besoins en ressources. Ce processus d'attribution, de classification et de synchronisation des acteurs sur les processeurs dans ce contexte est appelé *ordonnancement multiprocesseur* et est représenté sur la Figure A.2.

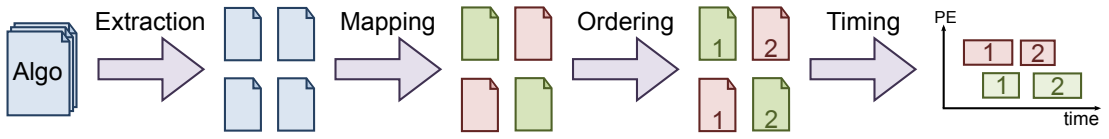


Figure A.2 – *Ordonnancement Multiprocesseur d'une application sur une plateforme composée de deux processeurs*

Tout d'abord, des tâches parallèles sont extraites à partir du programme initial (Extraction). Ensuite, la phase de répartition attribue un processeur à chaque tâche (Mapping). Cette phase doit gérer divers paramètres tels que l'utilisation processeur et le coût de la communication. La phase suivante de classification (Ordering) est parfois regroupée avec la phase de répartition. Elle crée une liste d'exécution ordonnée des tâches sur chaque processeur. Enfin, la phase de synchronisation (Timing) attribue les instants de début pour chaque tâche.

L'utilisation inefficace des processeurs peut conduire à un temps d'exécution global plus long. De plus, les périodes d'inactivités des processeurs qui en résultent entraînent une consommation électrique inutile. Ces considérations font de l'ordonnancement multiprocesseur un élément important pour avoir un système embarqué efficace. Cependant, la dynamicité des tâches de calculs et de leur dépendances de données font de l'ordonnancement multiprocesseur un problème complexe à résoudre [MTK+11].

Les sources de parallélisme d'une application peuvent être multiples. En général, il existe trois sources de parallélisme: le *parallélisme de tâches*, le *parallélisme de données* et le *parallélisme de pipeline*. La figure A.3 illustre ces trois sources de parallélismes.

Le parallélisme de tâche se produit lorsque deux tâches distinctes sont appliquées sur des ensembles de données d'entrée séparées. Cette source de parallélisme est habituellement facilement extraite de l'application.

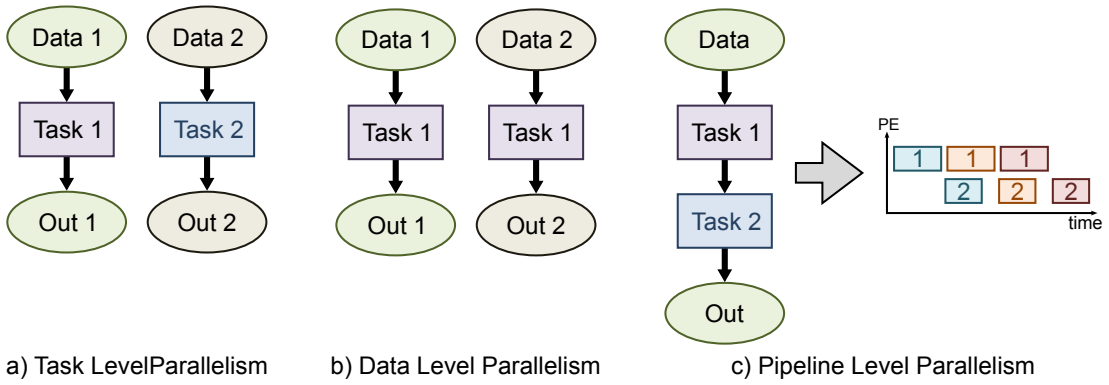


Figure A.3 – *Les sources de parallélisme*

Le parallélisme de données se produit lorsque la même tâche doit être exécutée sur deux ensembles de données différents. Le parallélisme de données est seulement possible quand il n'y a pas de dépendance entre les deux ensembles de données. De cette manière, les deux tâches peuvent être exécutées en parallèle sans problèmes.

Le parallélisme pipeline se produit lorsque deux processeurs du même système multiprocesseur calculent simultanément les tâches sur les données provenant d'itérations différentes du même algorithme.

La contribution majeure de cette thèse est l'élaboration d'une nouvelle méthode d'ordonnancement multiprocesseur pour répondre à ces défis de conception. Cette méthode est basée sur le modèle de calcul PiSDF, et est appelée Ordonnancement Multiprocesseur Juste-à-Temps (JIT-MS). JIT-MS est une méthode d'ordonnancement flexible qui prend les décisions d'ordonnancement durant l'exécution. Cette méthode se concentre sur l'optimisation de la répartition des tâches de l'application sur les ressources disponibles du système. En ce qui concerne la taxonomie de programmation définie par Lee et Ha [LH89], JIT-MS utilise une stratégie d'ordonnancement *fully dynamic*. De plus, JIT-MS permet d'exploiter les parallélismes de données, de tâches et de pipeline. Cette méthode est intégrée dans un système d'exécution appelé Spider.

Le choix du modèle de calcul impacte la méthode d'ordonnancement en termes de performance et de mise en œuvre. La sélection du modèle de calcul adéquat en terme de prédictibilité peut potentiellement fournir plus de parallélisme sur l'application. Inversement, le manque d'expressivité est une limite typique à la bonne prédictibilité d'un modèle. Une expressivité basse limite la gamme des applications qui peuvent être modélisées.

Dans cette thèse, le modèle de calcul PiSDF a été choisi pour exploiter le parallélisme par les périodes localement statiques qu'il propose lorsque tous les paramètres qui influencent l'algorithme ont été déterminés. L'idée fondamentale est que la détection et l'exploitation des propriétés localement statiques permettent de produire des solutions d'ordonnancement plus efficaces. Le modèle de calcul PiSDF permet au développeur de modéliser son application avec des paramètres variables. JIT-MS exploite les caractéristiques du modèle PiSDF pour extraire efficacement le parallélisme potentiel de l'application pour réduire la latence globale de l'application.

Une méthode itérative est utilisée par JIT-MS pour gérer la reconfiguration de l'application à travers les différents niveaux hiérarchiques. Le procédé d'ordonnancement multiprocesseur est ensuite divisé en plusieurs étapes et se concentre sur la construction progressive d'un graphe intermédiaire appelé SRDAG. Ce graphe intermédiaire est utilisé pour dévoiler

le parallélisme de l'application et agit comme un graphe de dépendance de tâches, qui est ensuite utilisé pour la répartition des tâches sur les différents processeurs.

Par l'utilisation de ce SRDAG, la méthode JIT-MS permet d'exploiter le parallélisme maximal entre les acteurs grâce à l'utilisation d'une heuristique d'ordonnancement. L'une des tâches d'ordonnancement la plus cruciale dans la méthode JIT-MS est la transformation *Single Rate*. Cette tâche est traitée immédiatement après que tous les paramètres d'un niveau de hiérarchie aient été déterminés. Cela permet de générer le SRDAG depuis le graphe PiSDF d'origine en entraînant la création d'acteurs supplémentaires appelés acteurs spéciaux. Ces acteurs sont utilisés pour gérer la distribution non-directe des jetons entre les différents acteurs.

A.3 Amélioration de la Méthode d'Ordonnancement Multiprocesseur

Il y a plusieurs possibilités pour optimiser la méthode JIT-MS afin d'obtenir de meilleures performances à l'exécution. Chaque optimisation vise une sous-partie de la méthode JIT-MS, et plusieurs améliorations seront détaillées.

Tout d'abord, une extension au modèle PiSDF est proposée pour améliorer les performances de l'ordonnancement mais aussi afin d'ajouter une nouvelle fonctionnalité aux développeurs PiSDF. Cette extension se concentre sur les valeurs initiales des délais dans le modèle PiSDF. Ces délais sont des jetons présents dans le graphe PiSDF à l'initialisation. En général, dans le modèle de calcul PiSDF, ces jetons sont non-initialisés ou mis à zéro. Dans l'extension proposée, ces jetons peuvent être initialisés par une interface ou un acteur de configuration. Cette extension simplifie les graphes PiSDF et permet une amélioration des performances de la méthode d'ordonnancement.

La deuxième optimisation possible agit sur le SRDAG, le but étant de réduire la complexité de ce graphe. En réduisant la complexité de ce graphe, l'empreinte mémoire sur la mémoire partagée ainsi que le temps de traitement de l'algorithme d'ordonnancement ont été réduits. Cette optimisation peut aussi permettre de révéler plus de parallélisme de tâches dans l'application et ainsi réduire le nombre de synchronisations au sein de l'ensemble du graphe. Cette optimisation est basée sur les acteurs spéciaux précédemment définis.

La dernière optimisation présentée est un nouvel algorithme de répartition des tâches sur les différents processeurs. Comme indiqué précédemment, la méthode JIT-MS est indépendante de l'algorithme utilisé pour la répartition des tâches. Tous les algorithmes proposés dans [SSKH13] peuvent donc être utilisés ici. Un des plus célèbre algorithme de répartition des tâches est appelé *list scheduler*. Cet algorithme possède un certain nombre de variantes : pour les résultats présentés dans cette thèse, la version avec chemin critique modifié (MCP) est utilisée [WG87]. Cependant, pour une architecture avec un grand nombre de processeurs l'algorithme *list scheduler* possède un temps d'exécution long. Pour les plateformes possédant des centaines de processeurs, un nouvel algorithme de répartition des tâches est proposé.

A.4 Spider: une Structure d'Exécution Multiprocesseur Flux de Données

Spider est une structure d'exécution multiprocesseur qui cible les systèmes embarqués hétérogènes. Il a été conçu pour être une structure d'exécution de bas niveau qui permet la

reconfiguration dynamique et efficace d'applications sur les plateformes multiprocesseurs. Il utilise pour cela les propriétés du modèle de calcul flux de données. Cette exécution prend un graphe PiSDF comme entrée et intègre la méthode d'ordonnancement JIT-MS.

Le développement d'une application PiSDF se décompose en deux étapes. Premièrement, l'application est modélisée en utilisant l'outil Preesm basé sur le logiciel Eclipse. Cet outil fournit une interface graphique permettant de décrire et développer l'application. Preesm peut également être utilisé comme un outil de prototypage rapide fournissant des simulations d'ordonnancement avec des paramètres fixés. La structure d'exécution Spider est alors utilisée pour l'exécution adaptative de ce graphe PiSDF.

Dans Spider, le graphe PiSDF est chargé dans la mémoire locale d'un gestionnaire centralisé appelé GRT. Le graphe PiSDF est chargé statiquement lors de l'initialisation de l'exécution. Le graphe PiSDF est ainsi décrit en utilisant du code C/C++. L'outil Preesm fournit déjà une interface graphique pour développer des graphes PiSDF, un générateur de code compatible à Spider a donc été ajouté à l'outil Preesm.

La structure d'exécution Spider possède une structure maître/esclave, où un maître appelé GRT effectue l'ordonnancement multiprocesseur. Il envoie ensuite des ordres d'exécutions de tâches à tous les esclaves appelés LRTs. Ces ordres contiennent les informations requises pour l'exécution des acteurs correspondants. Une fois que l'acteur est exécuté, chaque LRT renvoie des traces d'exécution au GRT. Le GRT a alors une vue globale sur l'exécution du graphe PiSDF récemment exécuté, lui permettant de produire un diagramme de Gantt de l'exécution passée.

La structure d'exécution Spider est conçue pour être portable sur plusieurs plateformes. Spider est une structure d'exécution séparée en plusieurs couches, lui permettant d'être efficacement portée sur un autre système avec peu de code supplémentaire. Actuellement, les systèmes Linux et Keystone II de Texas Instruments sont pris en charge par Spider, mais des travaux futurs peuvent impliquer plusieurs nouveaux systèmes.

A.5 Résultats Expérimentaux

Les performances de la structure d'exécution Spider ainsi que la méthode d'ordonnancement JIT-MS ont été évaluées sur trois applications. Des expérimentations ont été menées sur la plateforme Texas Instruments Keystone II.

La première application, appelée HCLM-Sched, est utilisée comme application de test. Elle est composée de plusieurs chaînes de filtres à réponse impulsionnelle finie (FIR). Chaque chaîne possède un nombre différent de filtres et le nombre de chaînes et de filtres peuvent varier à l'exécution. Les filtres FIR sont utilisés dans de nombreux algorithmes de traitement du signal des systèmes embarqués, notamment dans domaines des télécommunications et du traitement audio. Cette application HCLM-Sched a permis d'évaluer la performance des optimisations proposées dans cette thèse.

De plus, cette application a permis de comparer la structure d'exécution Spider et OpenMP. OpenMP est la structure d'exécution de référence dans l'industrie pour la programmation parallèle sur les systèmes embarqués multiprocesseurs. Le principal avantage d'OpenMP est de développer une application parallèle rapidement à partir de code séquentiel. OpenMP est connu pour être efficace pour les applications de traitement du signal. Une comparaison en performance entre Spider et OpenMP sur la plate-forme Keystone II est donc un défi. Il a été montré sur cet exemple que Spider prends de meilleures décisions qu'OpenMP mais il possède un surplus en temps d'ordonnancement.

Ensuite, un algorithme de traitement du signal de transformée de Fourier (FFT) a été implémenté avec Spider. La représentation de cet algorithme avec le modèle de calcul

PiSDF est proposée. En outre, cet algorithme est également utilisé pour démontrer la capacité de Spider à utiliser des plateformes hétérogènes.

La dernière application est un algorithme de vision par ordinateur, un algorithme de correspondance stéréo. Cette application calcule une carte de disparités à partir de deux images stéréoscopiques. Cet algorithme est décrit en PiSDF et est exécuté par Spider.

Ces trois applications sont utilisées pour étalonner la méthode d'ordonnancement sur des applications de différentes formes et ayant des besoins variés en termes de puissance de calcul. La représentation en graphe PiSDF de chacun de ces algorithmes est intéressante car elle implique différents types de graphe. La méthodologie de la représentation de ces différents algorithmes dans le modèle PiSDF est aussi une contribution de cette thèse.

A.6 Conclusion

L'évolution récente des systèmes embarqués a abouti à l'intégration d'un nombre croissant d'éléments de traitement dans les systèmes multiprocesseurs. Ainsi, de nouveaux modèles de programmation et langages sont étudiés pour exploiter les capacités de traitement parallèle de ces nouveaux systèmes. À cet effet, les modèles de calculs flux de données ont émergé comme une technique de programmation populaire pour exploiter le parallélisme des applications.

Cette thèse étudie les techniques de programmation pour les appareils embarqués multiprocesseurs et propose une nouvelle méthode d'ordonnancement multiprocesseur. Cette nouvelle méthode de programmation multiprocesseur est appelé JIT-MS et est basée sur un modèle de calcul flux de données paramétré appelé PiSDF [DPN+13]. Le but étant la répartition rapide et efficace des tâches d'une application sur un système multiprocesseur durant l'exécution.

Cette méthode de programmation utilise le modèle de calcul PiSDF afin de planifier l'exécution des tâches dès que certains points de reconfiguration ont été atteints. Cette méthode d'ordonnancement est basée sur un graphe intermédiaire appelé SRDAG qui est utilisé pour dévoiler les interactions entre toutes les tâches de l'application.

Des optimisations sur cette méthode ont aussi été introduites. Elles visent à l'amélioration de la performance globale de la méthode, et sont appliquées à différents niveaux dans la méthodologie proposée. Notamment, une extension au modèle de calcul PiSDF permet des améliorations de performance de la méthode d'ordonnancement en termes d'empreinte mémoire et de temps d'exécution de l'ordonnanceur.

Une structure d'exécution appelée Spider a aussi été développée. Cette exécution intègre la méthodologie JIT-MS et est utilisée pour exécuter des applications PiSDF. Les mécanismes d'exécution et de synchronisation utilisés sont étudiés ainsi que certaines optimisations permettant de réduire le nombre de synchronisations entre les différents processeurs. Cette structure d'exécution est conçue pour être portable sur différents systèmes multiprocesseurs hétérogènes.

Enfin, une étude de différentes applications modélisées en PiSDF a été présentée. Une étude sur la représentation en PiSDF de deux applications de traitement du signal et de l'image a été fournie, permettant l'évaluation de la performance de la programmation d'application en PiSDF. La structure d'exécution proposée et appelée Spider a été comparée à OpenMP et les résultats montrent que de meilleures décisions peuvent être obtenues par Spider avec un faible surcoût en temps d'exécution de la méthode.

A.7 Ouvertures

Les résultats de cette thèse ouvrent de nouvelles possibilités sur les modèles flux de données paramétrées. Ces modèles flux de données paramétrées offrent de nouvelles possibilités aux concepteurs d'applications pour décrire le comportement dynamique de celles-ci tout en conservant une certaine prédictibilité par rapport à des modèles flux de données purement dynamiques. Les travaux futurs peuvent être dirigés vers trois directions qui sont décrites ci-dessous.

A.7.1 Un Plus Grand Nombre de Plateformes

Les systèmes multiprocesseurs intégreront de plus en plus de processeurs, faisant de la programmation multiprocesseur de plus en plus complexe. Les heuristiques actuelles ne peuvent pas convenir à ces nouvelles plateformes. Une solution potentielle pour les applications d'ordonnancement sur un grand nombre de processeurs est l'heuristique HFS proposée dans cette thèse.

De plus, les systèmes multiprocesseurs avec des centaines de processeurs telle que la plateforme Kalray MPPA [MPP15] regroupant 256 processeurs, ne peuvent plus supporter une architecture avec mémoire partagée. En effet, celle-ci deviendrait nécessairement le goulot d'étranglement du système. Ainsi, les systèmes à mémoire distribuée sont susceptibles de devenir la nouvelle tendance dans les systèmes multiprocesseurs. Puisque les modèles de calcul flux de données ne sont pas réduits aux architectures avec mémoire partagée, l'utilisation de ces modèles devrait être possible. Ainsi, l'étude de ces nouvelles plateformes avec un nombre grandissant de processeurs est une perspective prometteuse pour la méthode d'ordonnancement JIT-MS et la structure d'exécution Spider.

A.7.2 Un Plus Grand Nombre d'Applications

Actuellement, les applications de traitement du signal et de l'image sont de plus en plus complexes et nécessitent une puissance de calcul importante. Puisque les performances d'ordonnancement Spider ont été démontrées comme étant efficaces lorsque les tâches composantes de l'application sont intensives, l'utilisation de Spider avec ces applications devrait être efficace.

Par exemple, les encodeurs et décodeurs vidéo HEVC [SOHW12] sont des algorithmes complexes à paralléliser et sont composés de tâches de calcul intensif. L'étude de ces nouvelles applications avec la méthode JIT-MS devient donc intéressante.

A.7.3 Une Méthode d'Ordonnancement Quasi-Satique

Spider est une structure d'exécution qui permet une exécution dynamique d'applications représentées avec un modèle flux de données. L'impact de l'ordonnanceur sur l'exécution peut maintenant être caractérisé en utilisant les travaux de cette thèse. Avec ces informations, le programmeur peut décider si la surcharge introduite par cette méthodologie adaptative est acceptable pour l'application considérée.

Un ordonnancement quasi-statique permet un ordonnancement adaptatif à faible coût si les valeurs possibles des paramètres sont restreint à des cas prédéfinis. Maintenant qu'une méthode d'ordonnancement dynamique a été conçue, un ordonnancement quasi-statique devrait être simple à mettre en œuvre et utile pour les applications nécessitant une faible latence.

Enfin, l'outil PREESM peut être étendu afin de devenir un ensemble permettant l'ordonnancement statique, quasi-statique et dynamique d'applications flux de données. Il peut ainsi fournir de précieux indicateurs de prototypage rapide aux concepteurs d'applications en respect avec leurs contraintes de conception du système.

List of Figures

List of Tables

Acronyms

3GPP 3rd Generation Partnership Project. 71

ADF Affine DataFlow. 30

API Application Programming Interface. 17, 84, 102

ASIMD Autonomous SIMD. 11

ASIP Application-Specific Instruction-Set Processor. 11

BDF Boolean DataFlow. 33, 34, 137

BRV Basic Repetition Vector. 28, 32, 35, 36, 39, 46–48, 50, 52, 56, 58, 97, 98, 108–112, 117, 139, 141

CISC Complex Instruction Set Computer. 11

CSDF Cyclo-Static Dataflow. 29, 30, 35, 137

DAET Deterministic Actor Execution Time. 70, 72

DAG Directed Acyclic Graph. 28, 46, 146

DDF Dynamic DataFlow. 41

DFT Discrete Fourier Transform. 106–108, 110

DIF Decimation in Frequency. 110, 111

DIT Decimation in Time. 109–111, 139, 141

DMA Direct Memory Access. 87–90

DPN Dataflow Process Network. 22, 23, 25, 26, 30, 34, 36, 37, 137

DSP Digital Signal Processor. 4, 10, 11, 14, 19, 69, 70, 73, 74, 78, 84, 87–90, 98, 108, 111, 113, 118, 126

Fifo First-In First-Out queue. 22–24, 26–29, 31, 34, 38, 40, 41, 47–58, 61, 63–66, 68, 69, 80, 81, 83, 85, 91, 92, 99, 108, 110, 121, 127, 137

FFT Fast Fourier Transform. 19, 89, 90, 95, 96, 108–113, 132, 139, 141

FFTC Fast Fourier Transform Co-processor. 89, 90, 113

FIR Finite Impulse Response. 95–98, 100, 102, 104, 132

FPGA Field-Programmable Gate Array. 84, 86

GPP General Purpose Processor. 4, 11, 69, 73, 126

GPU Graphics Processing Unit. 10, 11

GRT Global RunTime. 77–79, 83–86, 89, 94, 98, 111, 132

GST Greedy Scheduling Theorem. 74

HEVC High Efficiency Video Coding. 119, 122, 134

HFS Hybrid Flow-Shop. 70–76, 122, 134, 138

IBSDF Interface-Based SDF. 31–33, 37, 38, 137

IDF Integer-controlled DataFlow. 33

ILP Instruction-Level Parallelism. 14, 15, 17

IoT Internet of Things. 9

IPC Inter-Process Communication. 80, 82, 85, 87, 90, 91, 94

ISA Instruction Set Architecture. 11

JIT-MS Just-In-Time Multicore Scheduling. 5, 6, 19, 45–47, 57–61, 75–77, 82, 83, 94–96, 102, 119, 121, 122, 128–134, 137

KPN Kahn Process Network. 22, 137

LRT Local RunTime. 78, 79, 83–86, 89–91, 94, 100, 132

LTE Long Term Evolution. 71, 73, 75

MAPS MPSoC Application Programming Studio. 17

MIMD Multiple Instructions, Multiple Data. 11

MISD Multiple Instructions, Single Data. 10

MMX MultiMedia eXtensions. 14

MoC Model of Computation. 5, 6, 17–19, 21–26, 28–37, 39–41, 45, 48, 50–52, 56, 57, 59, 61–63, 75, 80, 82, 95, 96, 99, 101, 102, 108–111, 119, 121, 122

MPMD Multiple Programs, Multiple Data. 13, 14

MPSoC Multiprocessor System-on-Chip. 4, 11, 12, 14, 16, 17, 19, 87–90, 121, 122, 144

MSMC Multicore Shared Memory Controller. 87–89

NoC Network on Chip. 12

NORMA NO Remote Memory Access. 12

NUMA Non Uniform Memory Access. 12

Preesm Parallel and Real-time Embedded Executives Scheduling Method. 39–41, 77, 119, 123, 132, 135, 137

PE Processing Element. 4, 9, 11, 12, 14, 16, 17, 19, 21, 45, 46, 57, 61, 70–84, 90, 91, 103, 122, 137, 138

PiMM Parameterized and Interfaced dataflow Meta-Model. 37, 38

PiSDF Parameterized and Interfaced SDF. 5, 6, 21, 33, 37–41, 45–53, 56–66, 70, 75–78, 80, 82–85, 94–96, 98, 99, 101, 102, 108–111, 113, 117, 119, 121, 122, 127–133, 137–139, 141

PSDF Parameterized SDF. 33, 35–37, 39

PUSCH Physical Uplink Shared Channel. 73, 75

QMSS Queue Manager Sub-System. 87

RB Round Buffer. 47, 98, 117

RISC Reduced Instruction Set Computer. 11, 14, 19

RTOS Real-Time Operating System. 15, 16, 18, 82

S-LAM System-Level Architecture Model. 77

Spider Synchronous Parameterized Interfaced Dataflow Embedded Runtime. 5, 6, 45, 77–80, 82–92, 94–96, 98, 102–106, 113, 118, 119, 121, 122, 129–134, 138

SADF Scenario-Aware Dataflow. 33

SDF Synchronous Dataflow. 5, 21, 24, 26–37, 41, 45–47, 50, 52, 57, 60, 108, 127, 137, 144, 145

SIMD Single Instruction, Multiple Data. 10, 11, 14, 143, 146

SISD Single Instruction, Single Data. 10

SMP Symmetric Multi-Processor system. 15, 85

SPDF Schedulable Parametric Dataflow. 33–35

SPMD Single Program, Multiple Data. 13, 14

SRDAG Single-Rate DAG. 28–33, 46–49, 57–61, 65–69, 71–73, 76, 99–102, 104, 105, 108, 121, 130, 131, 133, 137, 138

SSE Streaming SIMD Extensions. 14

UMA Uniform Memory Access. 12

VLIW Very Long Instruction Word. 14

Personal Publications

[HBP+13] Julien Heulot, Jani Boutellier, Maxime Pelcat, Jean-Francois Nezan, and Slaheddine Aridhi. Applying the adaptive hybrid flow-shop scheduling method to schedule a 3gpp lte physical layer algorithm onto many-core digital signal processors. In *Adaptive hardware and systems (AHS), 2013 NASA/ESA conference on*, pages 123–129. IEEE, 2013. 71

[HDN+12] Julien Heulot, Karol Desnos, Jean François Nezan, Maxime Pelcat, Mickaël Raulet, Hervé Yviquel, Pierre-Laurent Lagalaye, and Jean-Christophe Le Lann. An experimental toolchain based on high-level dataflow models of computation for heterogeneous mpsoc. In *DASIP*, page xx, 2012.

[PDH+14] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi. PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 36–40. IEEE, 2014. 18, 39

[HOP+13] Julien Heulot, Yaset Oliva, Maxime Pelcat, Jean-Francois Nezan, and Jean-Christophe Prevotet. Dataflow-based adaptive multicore execution on a xilinx zynq platform. In *DATE Conference University booth, Grenoble, France.*, 2013. 86

[HPD+14] Julien Heulot, Maxime Pelcat, Karol Desnos, Jean-François Nezan, and Slaheddine Aridhi. Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 167–171. IEEE, 2014. 88

[HPN+14] Julien Heulot, Maxime Pelcat, Jean-François Nezan, Yaset Oliva, Slaheddine Aridhi, and Shuvra S Bhattacharyya. Just-in-time scheduling techniques for multicore signal processing systems. In *Signal and Information Processing (GlobalSIP), 2014 IEEE Global Conference on*, pages 25–29. IEEE, 2014.

[HMP+14] Julien Heulot, Judicaël Menant, Maxime Pelcat, Jean-François Nezan, Luce Morin, Muriel Pressigout, and Slaheddine Aridhi. Demonstrating a dataflow-based rtos for heterogeneous mpsoc by means of a stereo matching application. In *DASIP*, 2014.

[DH14] Karol Desnos and Julien Heulot. Pisdf: Parameterized & interfaced synchronous dataflow for mpsocs runtime reconfiguration. In *1st Workshop on MEthods and TOols for Dataflow PrOgramming (METODO)*, 2014.

Bibliography

[66A13] 66ak2h12 - Multicore DSP+ARM KeyStone II System-on-Chip (SoC). http://www.ti.com/lit/ds/symlink/66ak2h12.pdf, 2013. Accessed: 2015-07. 11

[AJLA14] M. Aguilar, R. Jimenez, R. Leupers, and G. Ascheid. Improving performance and productivity for software development on TI Multicore DSP platforms. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 31–35, September 2014. 17

[Bai89] David H. Bailey. FFTs in external of hierarchical memory. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, pages 234–242. ACM, 1989. 108

[Bar84] Hendrik Pieter Barendregt. *The lambda calculus*, volume 3. North-Holland Amsterdam, 1984. 22

[BB01] Bishnupriya Bhattacharya and Shuvra S Bhattacharyya. Parameterized dataflow modeling for dsp systems. *Signal Processing, IEEE Transactions on*, 49(10):2408–2421, 2001. 35, 36, 37

[BBS07] Jani Boutellier, Shuvra S Bhattacharyya, and Olli Silvén. Low-overhead run-time scheduling for fine-grained acceleration of signal processing systems. In *Signal Processing Systems, 2007 IEEE Workshop on*, pages 457–462. IEEE, 2007. 70

[BELP96] Greet Bilsen, Marc Engels, Rud Lauwereins, and Jean Peperstraete. Cycle-static dataflow. *Signal Processing, IEEE Transactions on*, 44(2):397–408, 1996. 28

[BFFM12] Luca Benini, Eric Flamand, Didier Fuin, and Diego Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, pages 983–987. EDA Consortium, 2012. 14

[BHLM94] Joseph T Buck, Soonhoi Ha, Edward A Lee, and David G Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems. 1994. 18

[BL06] S.S. Bhattacharyya and W.S. Levine. Optimization of signal processing software for control system implementation. In *2006 IEEE Computer Aided Control System Design, 2006 IEEE International Conference on Control Applications, 2006 IEEE International Symposium on Intelligent Control*, pages 1562–1567, October 2006. 25

[BLo93] Joseph T. Buck, Edward Lee, and others. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *Acoustics, Speech, and Signal Processing, 1993. ICASSP-93., 1993 IEEE International Conference on*, volume 1, pages 429–432. IEEE, 1993. 25, 33, 34, 35

[BN92] T. Blank and J.R. Nickolls. A Grimm collection of MIMD fairy tales. In *Frontiers of Massively Parallel Computation, 1992., Fourth Symposium on the*, pages 448–457, October 1992. 13

[Bou09] Jani Boutellier. *Quasi-static scheduling for fine-grained embedded multiprocessing*. PhD thesis, Ph. D. dissertation, University of Oulu, 2009. 70

[BSL+95] Shuvra S Bhattacharyya, Sundararajan Sriram, Edward Lee, et al. Minimizing synchronization overhead in statically scheduled multiprocessor systems. In *Application Specific Array Processors, 1995. Proceedings. International Conference on*, pages 298–309. IEEE, 1995. 91

[BTV12] Adnan Bouakaz, Jean-Pierre Talpin, and Jan Vitek. Affine data-flow graphs for the synthesis of hard real-time applications. In *Application of Concurrency to System Design (ACSD), 2012 12th International Conference on*, pages 183–192. IEEE, 2012. 30

[Buc94] Joseph T Buck. Static scheduling and code generation from dynamic dataflow graphs with integer-valued control streams. In *Signals, Systems and Computers, 1994. 1994 Conference Record of the Twenty-Eighth Asilomar Conference on*, volume 1, pages 508–513. IEEE, 1994. 33

[BYB08] Jun Ho Bahn, Jungsook Yang, and Nader Bagherzadeh. Parallel FFT algorithms on network-on-chips. In *Information Technology: New Generations, 2008. ITNG 2008. Fifth International Conference on*, pages 1087–1093. IEEE, 2008. 111, 113

[Car] Carmat website. `www.carmatsa.com`. Accessed: 2015-09-27. 3, 125

[CT65] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965. 107

[cud14] CUDA Zone. `https://developer.nvidia.com/cuda-zone`, 2014. Accessed: 2015-06-30. 17

[DA92] Rene David and Hassane Alla. *Petri Nets and Grafcet: Tools for Modelling Discrete Event Systems*. Prentice-Hall, New York, 1992. 21

[dD13] Benoit Dupont de Dinechin. Dataflow language compilation for a single chip massively parallel processor. In *2013 IEEE 6th International Workshop on Multi-/Many-core Computing Systems (MuCoCoS)*, pages 1–1. IEEE, 2013. 23

[DH14] Karol Desnos and Julien Heulot. Pisdf: Parameterized & interfaced synchronous dataflow for mpsocs runtime reconfiguration. In *1st Workshop on MEthods and TOols for Dataflow PrOgramming (METODO)*, 2014.

[DPN+13] Karol Desnos, Maxime Pelcat, Jean-François Nezan, Shuvra S Bhattacharyya, and Slaheddine Aridhi. PiMM: Parameterized and Interfaced Dataflow Meta-Model for MPSoCs Runtime Reconfiguration. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIII), 2013 International Conference on*, pages 41–48. IEEE, 2013. 5, 37, 41, 47, 57, 62, 121, 133

[DPNA15a] Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi. Buffer Merging Technique for Minimizing Memory Footprints of Synchronous Dataflow Specifications. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1111–1115, 2015. 48, 91, 92

[DPNA15b] Karol Desnos, Maxime Pelcat, Jean-François Nezan, and Slaheddine Aridhi. Memory analysis and optimized allocation of dataflow applications on shared-memory MPSoCs. *Journal of Signal Processing Systems*, 80(1):19–37, 2015. 40, 48

[EJ03] Johan Eker and Jörn W. Janneck. *CAL language report: Specification of the CAL actor language*. Electronics Research Laboratory, College of Engineering, University of California, 2003. 23

[Emb] Embedded system. https://en.wikipedia.org/wiki/Embedded_system. Accessed: 2015-09-21. 9

[Epi15] Epiphany – A breakthrough in parallel processing. www.adapteva.com, 2015. Accessed: 2015-09-21. 14

[Far07] William M. Farmer. Chiron: A multi-paradigm logic. *From Insight to Proof: Festschrift in Honour of Andrzej Trybulec*, 10(23):1–19, 2007. 25

[FGP12] Pascal Fradet, Alain Girault, and Peter Poplavko. Spdf: A schedulable parametric data-flow moc. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2012*, pages 769–774. IEEE, 2012. 34

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972. 10

[Fre82] Simon French. *Sequencing and scheduling: an introduction to the mathematics of the job-shop*, volume 683. Ellis Horwood Chichester, 1982. 72

[HBP+13] Julien Heulot, Jani Boutellier, Maxime Pelcat, Jean-Francois Nezan, and Slaheddine Aridhi. Applying the adaptive hybrid flow-shop scheduling method to schedule a 3gpp lte physical layer algorithm onto many-core digital signal processors. In *Adaptive hardware and systems (AHS), 2013 NASA/ESA conference on*, pages 123–129. IEEE, 2013. 71

[HDN+12] Julien Heulot, Karol Desnos, Jean François Nezan, Maxime Pelcat, Mickaël Raulet, Hervé Yviquel, Pierre-Laurent Lagalaye, and Jean-Christophe Le Lann. An experimental toolchain based on high-level dataflow models of computation for heterogeneous mpsoc. In *DASIP*, page xx, 2012.

[Hea02] Steve Heath. *Embedded Systems Design*. Newnes, October 2002. 10

[HMP+14] Julien Heulot, Judicaël Menant, Maxime Pelcat, Jean-François Nezan, Luce Morin, Muriel Pressigout, and Slaheddine Aridhi. Demonstrating a dataflow-based rtos for heterogeneous mpsoc by means of a stereo matching application. In *DASIP*, 2014.

[HOP+13] J Heulot, Y Oliva, M Pelcat, JF Nezan, and JC Prevotet. Dataflow-based adaptive multicore execution on a xilinx zynq platform. In *DATE Conference University booth, Grenoble, France.*, 2013. 86

[HPD+14] Julien Heulot, Maxime Pelcat, Karol Desnos, Jean-Francois Nezan, and Slaheddine Aridhi. Spider: A synchronous parameterized and interfaced dataflow-based rtos for multicore dsps. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 167–171. IEEE, 2014. 88

[HPN+14] Julien Heulot, Maxime Pelcat, Jean-François Nezan, Yaset Oliva, Slaheddine Aridhi, and Shuvra S Bhattacharyya. Just-in-time scheduling techniques for multicore signal processing systems. In *Signal and Information Processing (GlobalSIP), 2014 IEEE Global Conference on*, pages 25–29. IEEE, 2014.

[KAG+09] Lina J Karam, Ismail AlKamal, Alan Gatherer, Gene Frantz, David V Anderson, Brian L Evans, et al. Trends in multicore dsp platforms. *Signal Processing Magazine, IEEE*, 26(6):38–49, 2009. 70

[Kah74] Gilles Kahn. The semantics of a simple language for parallel programming. In *In Information Processing'74: Proceedings of the IFIP Congress*, volume 74, pages 471–475, 1974. 5, 22, 127

[Kal91] J. C. Kalb. Programmability with flexibility: Autonomous single instruction multiple data systems. *MasPar Computer Corporation, Sunnyvale, CA*, 1991. 11

[KL79] H. Kungt and Charles E. Leisersont. Systolic arrays (for VLSI). *Society for Industrial$\backslash$backslash$ & Applied*, page 256, 1979. 10

[KSB+12] Hojin Kee, Chung-Ching Shen, Shuvra S Bhattacharyya, Ian Wong, Yong Rao, and Jacob Kornerup. Mapping parameterized cyclo-static dataflow graphs onto configurable hardware. *Journal of Signal Processing Systems*, 66(3):285–301, 2012. 35

[Kwo97] Yu-Kwong Kwok. *High-performance Algorithms of Compile-time Scheduling of Parallel Processors*. PhD thesis, Hong Kong University of Science and Technology (People's Republic of China), 1997. AAI9820493. 40

[LC10] Rainer Leupers and Jeronimo Castrillon. MPSoC programming using the MAPS compiler. In *Design Automation Conference (ASP-DAC), 2010 15th Asia and South Pacific*, pages 897–902. IEEE, 2010. 17

[Lee06] Edward A. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006. 15

[Lei05] Charles E Leiserson. A minicourse on dynamic multithreaded algorithms, 2005. 74

[LH89] E. Lee and Soonhoi Ha. Scheduling strategies for multiprocessor real-time DSP. In *Global Telecommunications Conference and Exhibition'Communications Technology for the 1990s and Beyond'(GLOBECOM), 1989. IEEE*, pages 1279–1283. IEEE, 1989. 16, 45, 130

[LHGQ11] Guoxin Liu, Yeping He, Liang Guo, and Fang Qi. Static Scheduling of Synchronous Data Flow onto Multiprocessors for Embedded DSP Systems. In *2011 Third International Conference on Measuring Technology and Mechatronics Automation (ICMTMA)*, volume 3, pages 338–341, January 2011. 50

[LM87] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987. 26, 28, 30, 47

[LP95] Edward A. Lee and Thomas M. Parks. Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801, 1995. 22, 25

[MBFBJ02] Stephen J Mellor, Marc Balcer, and Ivar Foreword By-Jacoboson. *Executable UML: A foundation for model-driven architectures*. Addison-Wesley Longman Publishing Co., Inc., 2002. 21

[meg] Megahertz myth | Technology | The Guardian. `http://www.theguardian.com/technology/2002/feb/28/onlinesupplement3`. Accessed: 2015-09-27. 4, 126

[Moe14] Filip Moerman. Open event machine: A multi-core run-time designed for performance. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 41–45. IEEE, 2014. 18

[Moo65] Gordon E Moore. *Cramming more components onto integrated circuits*. McGraw-Hill New York, NY, USA, 1965. 4, 126

[MPMN14] Judicaël Menant, Muriel Pressigout, Luce Morin, and Jean-Francois Nezan. Optimized fixed point implementation of a local stereo matching algorithm onto C66x DSP. In *DASIP 2014*, 2014. 113

[MPP15] MPPA MANYCORE: a multicore processors family. `www.kalray.eu`, 2015. Accessed: 2015-09-21. 14, 122, 134

[MSZ11] Xing Mei, Xun Sun, and Mingcai Zhou. On building an accurate stereo matching system on graphics hardware. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 467 –474, November 2011. 113

[MTK+11] Peter Marwedel, Jürgen Teich, Georgia Kouveli, Iuliana Bacivarov, Lothar Thiele, Soonhoi Ha, Chanhee Lee, Qiang Xu, and Lin Huang. Mapping of applications to MPSoCs. In *Proceedings of the seventh IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 109–118. ACM, 2011. 45, 129

[NL04] Stephen Neuendorffer and Edward Lee. Hierarchical reconfiguration of dataflow models. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 179–188. IEEE, 2004. 25, 33

[Ope15a] OpenMP website. http://openmp.org/wp/, 2015. Accessed: 2015-06-30. 17, 102

[Ope15b] Open MPI: Open Source High Performance Computing. www.open-mpi.org, 2015. Accessed: 2015-07-01. 17

[PAPN12] Maxime Pelcat, Slaheddine Aridhi, Jonathan Piat, and Jean-François Nezan. *Physical layer multi-core prototyping: a dataflow-based approach for LTE eNodeB*, volume 171. Springer Science & Business Media, 2012. 74

[PBL95] José Luis Pino, Shuvra S. Bhattacharyya, and Edward A. Lee. *A hierarchical multiprocessor scheduling framework for synchronous dataflow graphs*. Electronics Research Laboratory, College of Engineering, University of California, 1995. 26, 50, 65

[PBR09] 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. IEEE, 2009. 30, 31, 38

[PDH+14] Maxime Pelcat, Karol Desnos, Julien Heulot, Clément Guy, Jean-François Nezan, and Slaheddine Aridhi. PREESM: A Dataflow-Based Rapid Prototyping Framework for Simplifying Multicore DSP Programming. In *Education and Research Conference (EDERC), 2014 6th European Embedded Design in*, pages 36–40. IEEE, 2014. 18, 39

[Pel10] Maxime Pelcat. *Rapid Prototyping and Dataflow-Based Code Generation for the 3GPP LTE eNodeB Physical Layer Mapped onto Multi-Core DSPs*. phdthesis, INSA de Rennes, September 2010. 48

[Pia10] Jonathan Piat. *Data Flow modeling and multi-core optimization of loop patterns*. PhD thesis, PhD thesis, INSA Rennes, 2010. 53, 99, 2010. 28, 31

[PNP+09] Maxime Pelcat, Jean Francois Nezan, Jonathan Piat, Jerome Croizer, and Slaheddine Aridhi. A system-level architecture model for rapid prototyping of heterogeneous multicore embedded systems. In *Conference on Design and Architectures for Signal and Image Processing (DASIP) 2009*, pages 8–pages, 2009. 40, 77

[POH09] Hae-Woo Park, Hyunok Oh, and Soonhoi Ha. Multiprocessor SoC design methods and tools. *Signal Processing Magazine, IEEE*, 26(6):72–79, 2009. 17

[PPL+95] Thomas M Parks, Josù Luis Pino, Edward Lee, et al. A comparison of synchronous and cycle-static dataflow. In *Signals, Systems and Computers, 1995. 1995 Conference Record of the Twenty-Ninth Asilomar Conference on*, volume 1, pages 204–210. IEEE, 1995. 30

[PPW+09] Maxime Pelcat, Jonathan Piat, Matthieu Wipliez, Slaheddine Aridhi, and Jean-François Nezan. An open framework for rapid prototyping of signal processing applications. *EURASIP journal on embedded systems*, 2009:11, 2009. 50

[PRE15] PREESM website. http://preesm.sourceforge.net/website/, 2015. Accessed: 2015-07-02. 39

[RVdM14] J. Rivera and R. Van der Meulen. Gartner's 2014 Hype Cycle for Emerging Technologies Maps the Journey to Digital Business. *Retrieved March*, 31:2015, 2014. 9

[RVR10] Rubén Ruiz and José Antonio Vázquez-Rodríguez. The hybrid flow shop scheduling problem. *European Journal of Operational Research*, 205(1):1–18, 2010. 70, 72

[SB12] Sundararajan Sriram and Shuvra S. Bhattacharyya. *Embedded multiprocessors: Scheduling and synchronization*. CRC press, 2012. 46, 70

[SBA13] Jocelyn Sérot, François Berry, and Sameer Ahmed. Caph: a language for implementing stream-processing applications on fpgas. In *Embedded Systems Design with FPGAs*, pages 201–224. Springer, 2013. 23

[SGS10] John E Stone, David Gohara, and Guochun Shi. OpenCL: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66, 2010. 17

[SGTB11] Sander Stuijk, Marc Geilen, Bart Theelen, and Twan Basten. Scenario-aware dataflow: Modeling, analysis and implementation of dynamic applications. In *Embedded Computer Systems (SAMOS), 2011 International Conference on*, pages 404–411. IEEE, 2011. 36

[Sim15] Simulink Webpage. http://fr.mathworks.com/products/simulink/, 2015. Accessed: 2015-02-07. 18

[SOHW12] Gary J Sullivan, J-R Ohm, Woo-Jin Han, and Thomas Wiegand. Overview of the high efficiency video coding (hevc) standard. *Circuits and Systems for Video Technology, IEEE Transactions on*, 22(12):1649–1668, 2012. 119, 122, 134

[SPR14] SPRS691e - Multicore Fixed and Floating-Point Digital Signal Processor. http://www.ti.com/lit/pdf/sprs691e, 2014. Accessed: 2015-07. 14

[SSKH13] A.K. Singh, M. Shafique, A. Kumar, and J. Henkel. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, page 1. ACM, 2013. 61, 79, 131

[Ste90] Per Stenström. A survey of cache coherence schemes for multiprocessors. *Computer*, 23(6):12–24, 1990. 13

[Sys15] SystemC website. http://accellera.org/downloads/standards/systemc, 2015. Accessed: 2015-07-02. 18

[TBG+13] Stavros Tripakis, Dai Bui, Marc Geilen, Bert Rodiers, and Edward A. Lee. Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(3):83, 2013. 26, 30, 31

[Texa] Texas Instruments. *KeyStone Architecture Multicore Navigator*. (accessed 06/2014). 87

[Texb] Texas Instruments. *Multicore DSP+ARM KeyStone II System-on-Chip (SoC) - SPRS866E*. (accessed 08/2015). 88

[Texc] Texas Instruments. *Multicore Fixed and Floating-Point Digital Signal Processor - SPRS691E.* (accessed 05/2015). 87

[TGB+06] Bart D Theelen, Marc CW Geilen, Twan Basten, Jeroen PM Voeten, Stefan Valentin Gheorghita, and Sander Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Proceedings of the Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE'06. Proceedings.*, pages 185–194. IEEE Computer Society, 2006. 33

[TKA02] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002. 18

[Tur36] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *J. of Math*, 58(345-363):5, 1936. 22

[WG87] Min-You Wu and Daniel Gajski. A programming aid for message-passing systems. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, pages 328–332. Society for Industrial and Applied Mathematics, 1987. 61, 74, 131

[WSC+13] Matthieu Wipliez, Nicolas Siret, Nicola Carta, Francesca Palumbo, and Luigi Raffo. Design ip faster: Introducing the chigh-level language. *Design & Reuse*, 2013. 23

[YLJ+13] Hervé Yviquel, Antoine Lorence, Khaled Jerbi, Gildas Cocherel, Alexandre Sanchez, and Mickaël Raulet. Orcc: Multimedia development made easy. In *Proceedings of the 21st ACM international conference on Multimedia*, pages 863–866. ACM, 2013. 18

[Zed15] Zedboard website. www.zedboard.org, 2015. Accessed: 2015-07-02. 86

[ZPB+13] George F Zaki, William Plishker, Shuvra S Bhattacharyya, Charles Clancy, and John Kuykendall. Integration of Dataflow-Based Heterogeneous Multiprocessor Scheduling Techniques in GNU Radio. *Journal of Signal Processing Systems*, 70(2):177–191, 2013. 96

[ZPBF12] G.F. Zaki, W. Plishker, S.S. Bhattacharyya, and F. Fruth. Partial Expansion Graphs: Exposing Parallelism and Dynamic Scheduling Opportunities for DSP Applications. In *2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 86–93, July 2012. 50

[ZT98] MengChu Zhou and Edward Twiss. Design of industrial automated systems via relay ladder logic programming and petri nets. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 28(1):137–150, 1998. 21

INSA de RENNES
Service des Formations

AVIS DU JURY SUR LA REPRODUCTION DE LA THESE SOUTENUE

Titre de la thèse:
Techniques d'Ordonnancement en Ligne pour la Répartition d'Applications Flot de Données de Traitement de Signal et de l'Image sur Architectures Multi-coeur Hétérogène Embarquée

Nom Prénom de l'auteur : HEULOT JULIEN

Membres du jury :
- Monsieur SASSATELLI Gilles
- Monsieur GIRAULT Alain
- Monsieur LILIUS Johan
- Monsieur McALLISTER John
- Monsieur ARIDHI Slaheddine
- Monsieur NEZAN Jean-François
- Monsieur PELCAT Maxime
- Madame MUNIER Alix

Président du jury : Alix Munier

Date de la soutenance : 24 Novembre 2015

Reproduction de la these soutenue

Thèse pouvant être reproduite en l'état
~~Thèse pouvant être reproduite après corrections suggérées~~

Fait à Rennes, le 24 Novembre 2015

Signature du président de jury

Le Directeur,

M'hamed DRISSI

A. MUNIER